Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004

Zbral in uredil Janez Brank

Institut Jožef Stefan
Ljubljana, 2006

And then my heart with pleasure fills,
And dances with the daffodils.

WORDSWORTH

Rešene naloge s srednješolskih računalniških tekmovanj, 1988–2004
Institut Jožef Stefan, 2006

Uredil Janez Brank

Tisk: Present d. o. o.
Naklada: 500 izvodov

Ta različica zbirke je bila pripravljena 14. 12. 2006.

CIP — Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

371.27:004(497.4)
004.42(079.1)

REŠENE naloge s srednješolskih računalniških tekmovanj 1988–2004 [Elektronski vir] / [zbral in uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2006

Način dostopa (URL): `http://rtk.ijs.si/naloge/rtknj.pdf`

ISBN-10 961-6303-88-0
ISBN-13 978-961-6303-88-0
1. Brank, Janez, 1979– 230267136

Kazalo

Leto		Naloge				Rešitve			
	Skupina	1	2	3	Z/U	1	2	3	Z/U
1988	Stran	22	23	26		28	32	37	
1989		45	46	47		48	54	67	
1990		79	81	82		84	89	95	
1991		107	110	112		115	120	128	
1992		135	136	138		140	142	146	
1993		155	157	159		161	165	173	
1994		181	182	184		187	192	200	
1995		211	213	216	219	222	227	232	247
1996		259	260	263	268	271	275	281	288
1997		292	294	297	299	303	308	317	323
1998		340	344	346		348	352	363	
1999		367	368	372	684	376	378	385	694
2000		395	397	401	685	403	413	416	701
2001		429	431	437	687	442	458	464	706
2002		483	485	491	688	500	505	509	710
2003		527	529	535	690	542	551	557	713
2004		579	582	590	692	596	599	605	718

Z = zaključno tekmovanje (reševanje nalog na računalnikih). Taka tekmovanja so potekala v letih 1995, 1996 in 1997.

U = tekmovanje v poznavanju Unixa. Taka tekmovanja so potekala v letih 1999–2004.

Predgovor

Računalniška tekmovanja imajo v Sloveniji spoštljivo dolgo zgodovino — segajo v sedemdeseta leta dvajsetega stoletja, kar samo po sebi pove veliko o tradiciji in kulturi gojenja odličnosti v računalniški znanosti v Sloveniji. Tekmovanja so namreč le pokazatelj širšega dogajanja, ki pa ima mnogo izrazov. Zagotovo lahko rečemo, da je najpomembnejši izraz in tudi glavni motiv, da računalniška tekmovanja približajo mladim tekmovalcem aktualne znanstvene in tehnološke tematike skozi preprost jezik, zapisan v kratkih nalogah.

V treh desetletjih se je na slovenskih srednješolskih računalniških tekmovanjih prekalilo veliko generacij dijakov, ki dandanes po večini predstavljajo okostje slovenske računalniške skupnosti. Bivši tekmovalci so dandanes profesorji na univerzah, raziskovalci na institutih, predvsem pa si brez njih ni mogoče predstavljati slovenske računalniške industrije. Mnogo jih je tudi odšlo v tujino in uspelo na najuglednejših institucijah. Po vseh teh letih lahko opazimo, da je večina uspešnejših tekmovalcev dosegala uspehe tudi kasneje v študijskem in poklicnem življenju.

Računalniška tekmovanja potekajo že tri desetletja. Segajo v čase, ko so se ljudje še spraševali, kako npr. urejati podatke ali kako početi mnogo preprostih tehničnih opravil, ki jih danes lahko izvemo v prvih urah učenja računalništva in se nam zdijo povsem samoumevna. Segajo tudi v čase, ko so bili računalniki še zelo veliki, ko so bili v uporabi programski jeziki, ki bi se jim dandanes mnogi samo čudili, ko še ni bilo sedaj vsem znanih operacijskih sistemov in ko večina ljudi pravzaprav sploh ni vedela, kaj je to računalnik. Za tisti čas so veljale povsem druge „verske vojne“ — če se dandanes nekateri prepirajo, ali je boljši Linux ali Windowsi oz. Java ali C++ oz. ta ali oni spletni brskalnik, lahko le rečemo, da takrat ni bilo še nobene od naštetih zadev. Takrat so bile vojne na danes nerazumljive tematike, kot npr.: ali je boljši IBM ali DEC oz. Fortran ali Pascal oz. se je bolje ukvarjati s „softverom“ ali „hardverom“. Od začetkov računalniških tekmovanj se je dogodilo veliko — znanost si je odgovorila na mnoga vprašanja, razvila se je močna svetovna računalniška industrija, pojavil se je internet, menjali so se programski jeziki, razvili so se operacijski sistemi, pojavilo se je mobilno komuniciranje in še mnogo drugega, kar uporabljamo za tematike pri tekmovalnih nalogah.

Pričujoča zbirka nalog v dveh zvezkih je nastajala ravno v obdobju največjega razcveta računalništva. Naloge vsebujejo mnogo iskric in intuicij, ki so vgrajene v naprave, ki jih uporabljamo vsakodnevno. Zbirka je nastajala postopoma in je rezultat dela nekaj ducatov članov komisije za računalniška tekmovanja. Vsa imena je skoraj nemogoče našteti, ne da bi koga izpustili. Posebej bi izpostavil le Janeza Branka, ki je zbirki dal zaključni ton in je iz množice drobnih prispevkov sestavil urejeno in uravnoteženo zbirko nalog

in ji pridal mnogo dodatnega materiala, ki podrobneje pojasnjuje posamezne tematike in izdatno bogati zbirko.

Komu je namenjena zbirka? Pravprav vsem, ki jih računalništvo zanima — od osnovnošolcev in dijakov do akademikov, računalnikarjev iz industrije in ljubiteljskih entuziastov. V zbirko lahko pogledamo, ko bi radi vadili za tekmovanje ali izpit ali pa, da se npr. izobrazimo v tematikah, ki nam niso najbližje. Naloge so predstavljene na raznih ravneh težavnosti in so tako primerne za širok spekter bralcev. Tematsko zajemajo večino področij računalništva in področij, ki so z računalništvom povezana. Poudarek je na algoritemskih in programerskih nalogah, ki pa pogosto v sebi skrivajo probleme in rešitve, ki še zdaleč niso samo programerske narave.

Vsem bralcem zbirke želimo veliko zadovoljstva ob prebiranju nalog in mnogo novih uvidov.

Marko Grobelnik
Kranj, 8. decembra 2006

Statistika

Tekmovanje	Datum	Število tekmovalcev					
		Skupaj	Po skupinah				
		$(1+2+3)$	1	2	3	Z	U
12.	21. 5. 1988	205	126	56	23		
13.	20. 5. 1989	?	?	?	?		
14.	19. 5. 1990	?	?	?	?		
15.	18. 5. 1991	131	54	47	20		
16.	16. 5. 1992	?	?	?	?		
17.	15. 5. 1993	142	66	47	29		
18.	14. 5. 1994	126	64	38	24		
19.	13. 5. 1995	155	67	53	35	14, 10	
20.	10.–11. 5. 1996	121	58	37	27	19	
21.	9.–10. 5. 1997	112	64	30	18	27	
22.	8.–9. 5. 1998	111	56	34	21		
23.	24. 4. 1999	105	61	27	16		13
24.	1. 4. 2000	94	40	28	26		22
25.	7. 4. 2001	94	28	45	21		14
26.	6. 4. 2002	81	25	26	30		13
27.	5. 4. 2003	67	24	17	26		5
28.	3. 4. 2004	94	44	30	20		7

Z = zaključno tekmovanje, U = tekmovanje v poznavanju Unixa.

Kraj tekmovanj: 1988 na Odseku za matematiko in mehaniko na FNT, v letih 1989–2004 na FER oz. FRI, razen tekmovanja za tretjo skupino v letih 2001–4, ki je bilo na FMF. Organizator tekmovanj je bila ZOTKS (Zveza organizacij za tehnično kulturo Slovenije oz. kasneje Zveza za tehnično kulturo Slovenije).

Kjer sta navedena dva dneva, je prvi petek, ko je bilo tekmovanje za tretjo skupino, drugi pa sobota, ko je bilo tekmovanje za prvo in drugo skupino. To je bilo uvedeno zato, ker je veliko tekmovalcev iz 3. skupine pisalo tudi maturo in bi se jim tekmovanje v soboto prekrivalo s pisanjem eseja za maturo pri slovenščini.

Datumi zaključnih tekmovanj: 14. in 27. 5. 1995, 2. 6. 1996, 25. 10. 1997.

Podatkov o številu tekmovalcev za leta 1989, 1990 in 1992 ni niti v biltenih tistih let. Podatek o številu tekmovalcev za leto 1995 v gornji tabeli temelji na prijavah, ne pa na dejanski udeležbi (kot ostala leta); pravo število udeležencev je verjetno okoli 120, podobno kot na prejšnjem in naslednjem tekmovanju.

Zahvale

Pri nastajanju te zbirke je sodelovalo veliko ljudi. Še posebej bi se rad zahvalil Marjani Plukavec, Mojci Miklavec, Marku Martincu, Poloni Novak, Blažu Novaku, Andražu Toriju, Alešu Koširju, Andražu Bežku, Gorazdu Božiču, Blažu Fortuni, Borisu Gašperinu, Marku Grobelniku, Urošu Jovanoviču, Mitji Lasiču, Juretu Leskovcu, Dunji Mladenić, Branetu Sotošku, Marjanu Šterku, Mihi Vuku in Anžetu Žagarju. Hvala tudi vsem, ki so v teh letih prispevali naloge in rešitve ali kako drugače pomagali pri izvedbi tekmovanja. Za zadnjih nekaj let so predlagatelji nalog znani in so navedeni na koncu rešitev za posamezno leto. Hvala tudi tekmovalcem ter njihovim mentorjem, saj brez njih ne bi bilo tekmovanja in tudi te zbirke nalog ne.

*

Bralce, ki opazijo v zbirki še kakšno napako ali imajo kakšen drug komentar, predlog, vprašanje, izboljšavo ali kaj podobnega, vabim, naj mi svoje pripombe pošljejo po elektronski pošti na naslov `janez@brank.org`.

Uvod

Nekaj uredniških opomb

Besedilo nalog je v tej zbirki objavljeno približno takšno, kakršnega so dobili udeleženci tekmovanja, le z nekaj drobnimi popravki in občasno kakšnim pojasnilom; vse opombe pod črto so urednikove. Rešitve nalog pa so v mnogih primerih močno predelane in razširjene (v primerjavi s tistimi, ki so za nekatera leta že bile objavljene, npr. v biltenih tekmovanj). Mnoge rešitve so zato tudi obsežnejše, kot bi jih na tekmovanjih pričakovali od tekmovalcev; pri tem nas je vodil namen, da bi se lahko bralec iz rešitev v naši zbirki še česa naučil, mogoče videl več možnih pristopov k reševanju problema ipd. Želeli smo tudi poudariti, da je koristno in pomembno, če človek o svoji rešitvi tudi malo razmisli in se poskuša prepričati, če je res pravilna, ter dobiti občutek za njeno časovno in prostorsko zahtevnost. Posledica tega so v tej zbirki mnogi razmisleki, izpeljave in dokazi, ki bodo bralca mogoče dolgočasili ali pa presegali njegovo znanje matematike; v tem primeru je vsekakor bolje, da jih preskoči, kot pa da bi ga pretiravanje s temi rečmi zanašalo v pedantnost ali mu zbijalo veselje do algoritmov in programiranja.

Na začetku besedila vsake naloge je na zunanjem robu strani navedeno, na kateri strani se začne rešitev te naloge, in obratno.

Izvorna koda rešitev nalog je objavljena na spletni strani `rtk.ijs.si`. Tam se dobi tudi elektronsko različico te zbirke ter testne podatke za tiste naloge, ki so jih tekmovalci reševali na računalnikih (zaključna tekmovanja v letih 1995–7 ter tretja skupina v letih 2001–4).

O tekmovanju

V letih 1988–2004 je potekal glavni del tekmovanja v treh težavnostnih skupinah. Naloge za prvo skupino so najlažje in od tekmovalcev ne zahtevajo veliko znanja programiranja, le nekaj občutka za sistematično razmišljanje. Naloge za drugo skupino so namenjene tekmovalcem, ki so se učili programirati že kakšno leto ali dve. Naloge za tretjo skupino so najtežje in predpostavljajo znanje programiranja in nekaj malega znanja o enostavnejših algoritmih (ali pa vsaj sposobnost razmišljanja o njih).

Praviloma so naloge v vsaki skupini štiri in na tekmovanju so imeli reševalci zanje dve uri in pol (150 minut) časa. Reševalci so pisali svoje odgovore na papir, po tekmovanju pa jih je ocenila tekmovalna komisija (izjema so zaključna tekmovanja v letih 1995–7 in tekmovanje za tretjo skupino v letih 2001–4). Pri ocenjevanju rešitev je glavni kriterij to, da je predlagana rešitev pravilna, zaželeno pa je tudi, da je uporabljeni postopek čim bolj učinkovit. Sintaktična pravilnost rešitev pa ni tako zelo pomembna; manjkajoča podpičja in podobne

zadeve bi nam v praksi hitro pomagal popraviti prevajalnik, dobrega postopka pa ne bo znal sestaviti namesto nas.

V letih 2001–4 je potekalo tekmovanje za tretjo skupino na računalnikih; število nalog je bilo različno, časa za reševanje pa je bilo načeloma tri ure. Vsaka naloga zahteva od tekmovalca, da napiše program, ki prebere neko vhodno datoteko, izračuna zahtevani rezultat in ga izpiše v izhodno datoteko. Programe se je preverjalo avtomatsko: tekmovalci jih pošljejo na ocenjevalni strežnik, ki jih prevede, požene na več testnih primerih in preveri, če so vrnili pravilne odgovore. Tekmovalec takoj ob oddaji programa izve, koliko testnih primerov je pravilno rešil; med tekmovanjem lahko za posamezno nalogo odda tudi več programov in na koncu se upošteva rezultat najboljšega izmed njih.

Čeprav je mnogim tekmovalcem tak tip tekmovanja zelo všeč, pa je v nekaterih pogledih v bistvu težji od tekmovanja, pri katerem se rešuje naloge na papir in odgovore nato oceni komisija. Pri tekmovanju na računalnikih in z avtomatskim ocenjevanjem morajo udeleženci izpiliti svojo rešitev vse do pravilno delujočega programa (oz. takega, ki pravilno reši vsaj nekaj testnih primerov) — če prehodimo pri neki nalogi le del poti do rešitve, če imamo le do polovice napisan program ali pa nekaj idej v pravi smeri, ne bomo dobili pri tisti nalogi nobenih točk (v časih pisanja na papir in ročnega ocenjevanja pa bi takšna delna rešitev dobila vsaj nekaj točk). Priti do dobrega rezultata je torej zdaj pri marsikateri nalogi težje in bolj zamudno kot v časih reševanja na papir, saj moramo zdaj rešiti nalogo (skoraj) v celoti in poloviti dovolj velik delež napak v našem programu. Da tekmovanje ne bi postalo pretežko, so naloge v tretji skupini v teh letih mogoče za odtenek lažje, kot so bile pred uvedbo tekmovanja na računalnikih (predvsem pa nekatere vrste nalog zdaj ne pridejo več v poštev za v tretjo skupino).

Nalog je bilo v tretji skupini v teh letih ponavadi več, kot jih lahko tekmovalec reši v razpoložljivem času (slabih treh urah), zato je del izziva pri tekmovanju tudi to, da se mora tekmovalec odločiti, katerih nalog se bo lotil (in v kakšnem vrstnem redu) — načeloma se je pametno najprej lotiti tistih nalog, za katere se mu zdi, da jih bo najlaže in najhitreje rešil. Da bi tekmovalcem ta razmislek malo olajšali, smo že mi uredili in oštevilčili naloge od lažjih proti težjim, je pa seveda mogoče, da imajo tekmovalci o tem, katere naloge so lažje in katere težje, drugačno mnenje kot komisija.

V letih 1995–97 so bila organizirana tudi zaključna tekmovanja (playoffi), leta 1995 celo v dveh krogih. Namen teh tekmovanj je bil predvsem izbrati ekipo, ki je zastopala Slovenijo na mednarodni računalniški olimpijadi (IOI), in izbrati tekmovalce, ki so kot nagrado dobili možnost poletne prakse v tujini. Na zaključna tekmovanja so bili povabljeni najboljši tekmovalci iz druge in tretje skupine ter najboljši udeleženci srečanja mladih raziskovalcev s področja računalništva. V letih 1995 in 1996 je bila na zaključnem tekmovanju ena naloga, leta 1997 pa tri naloge. Čas reševanja je bil iz leta v leto različen (leta

1995 tri ure, 1996 štiri, 1997 pa pet). Tudi tu se je tekmovalo na računalnikih na podoben način kot v tretji skupini v letih 2001–4; razlika je predvsem ta, da je preverjanje rešitev potekalo šele na koncu tekmovanja, ne pa že med njim.

Vrste nalog

Največ nalog je tipa „napiši (pod)program“, torej zahtevajo, naj reševalec v nekem konkretnem programskem jeziku napiše program (ali podprogram), ki reši dani problem. Pri nekaterih nalogah tega tipa lahko reševalec tudi predpostavi, da so mu že dani na voljo določeni podprogrami, s katerimi si lahko pomaga. Zelo zaželeno pri teh programskih rešitvah pa je, da vsebujejo tudi nekakšen komentar ali razlago, iz katere je razvidno, kako naj bi program deloval; če drugega ne, je pri ocenjevanju rešitev potem precej lažje razumeti, kaj je imel reševalec v mislih. (Tako ali tako pa je komentiranje programov koristno tudi na splošno, če s tem ne pretiravamo.)

Druga zelo pogosta vrsta nalog pa je „opiši postopek“, kjer je poudarek bolj na tem, da bi reševalec razmislil o postopku za reševanje nekega problema, ne pa toliko na tem, da bi ga izrazil v nekem konkretnem programskem jeziku in se obremenjeval s tehnikalijami, ki se jim pri tem skoraj ni mogoče izogniti. Poleg rešitve v obliki programa bi bil zato sprejemljiv tudi opis z besedami (da je le dovolj jasen in natančen), s psevdokodo, diagramom poteka ali čim podobnim. Pri nekaterih nalogah tega tipa ne gre toliko za sestavljanje postopka, pač pa bolj za razmislek o tem, kako naj bi nek sistem ali program deloval oz. kako bi se lotil reševanja določenega problema.

„Realnočasovne“ naloge zahtevajo program ali postopek, ki naj bi se sproti (v realnem času) odzival na neke zunanje dogodke. Reševalec mora na primer napisati podprogram, ki bi ga sistem poklical ob nekem dogodku, podprogram pa naj bi nanj odreagiral. Običajno je tudi natančno določeno, kakšen vmesnik (nekaj podprogramov) je na voljo za sporazumevanje z zunanjim svetom.

Pri nalogah vrste „kaj dela program“ je podan nek program ali podprogram, reševalec pa naj bi ugotovil, kaj (oz. kako) ta program dela; ali pa je v programu kakšna napaka, ki jo je treba najti in predlagati popravek. V zadnjih letih je bilo takih nalog manj in smo se jim bolj izogibali, ker se je pri njih prepogosto dogajalo, da so imeli tekmovalci ali skoraj nič točk ali pa skoraj vse, malo pa je bilo takih z delno pravilnimi rešitvami; preveč točk je bilo torej odvisnih le od tega, ali se bo reševalcu slučajno utrnila ravno prava ideja za tisto nalogo ali ne.

Tekmovanje v poznavanju Unixa

Tekmovanje iz znanja računalništva je bilo leta 1999 dopolnjeno še z dodatno disciplino poznavanja Unixa, saj se je pokazala priložnost preveriti, koliko tekmovalcev ob splošnem algoritemskem razmišljanju pozna tudi nove računalniške koncepte, kakršne ponuja operacijski sistem Unix. Ime Unix je tu

uporabljeno v posplošenem pomenu in zajema vso družino operacijskih sistemov, ki so združljivi s standardom POSIX (Portable Operating System Interface for UniX), vanjo pa od bolj znanih in pri nas razširjenih predstavnikov sodijo na primer GNU/Linux, BSD in FreeBSD, HP-UX, Solaris in AIX. To tekmovanje je bilo na sporedu po običajnem tekmovanju v znanju računalništva in pred razglasitvijo rezultatov, tako da so imeli na njem možnost sodelovati vsi tekmovalci. Trajalo je uro in pol, reševanje pa je potekalo na računalnikih, tako da so lahko tekmovalci delovanje svojih zamisli tudi preizkusili, kar daje tekmovanju dodatno privlačnost. Za preizkušanje pravilnosti svojih rešitev so lahko tekmovalci uporabljali sistem za preverjanje, ki na nekaj vhodnih primerih preizkusi, kako se vedejo rešitve, in rezultat preizkušanja sporoči tekmovalcu. Oddane rešitve pregleda tudi komisija za ocenjevanje in med funkcionalno enakovrednimi rešitvami v primeru enakega skupnega števila točk tekmovalcev te razvrsti še po lepoti in jasnosti njihovih rešitev.

Pokazalo se je, da je splošno znanje tekmovalcev visoko in razmeroma stalno. Praviloma dosegajo najboljše rezultate tisti, ki so uspešni tudi pri drugih skupinah, kar kaže na tesno prepletenost obeh znanj in na to, da imajo zmagovalci radi Unix.

Pri tekmovanju iz znanja Unixa so zastavljene štiri naloge, ki jih morajo tekmovalci rešiti na računalniku. Naloge so izbrane tako, da lahko tekmovalec pri vsaki pokaže eno od spretnosti:

1. učinkovita uporaba regularnih izrazov,
2. spretna uporaba cevovodov v ukazni lupini,
3. optimalna izvedba algoritmov, zapisanih v skriptnih jezikih,
4. prepoznava in uporaba nekega temeljnega koncepta Unixa.

Regularni izrazi so tako močno orodje, a na nekaterih sistemih premalo razširjeno, da se nam ga zdi vredno popularizirati tudi na ta način. Koncept cevovodov in povezovanja programov z njimi v učinkovite zveze je tudi presenetljivo zmogljiv način, s katerim moremo na računalnikih dnevno reševati probleme.

Komisija zastavi naloge po vrsti od lažjih proti težjim, s čimer želi namigniti tekmovalcem, katere naloge naj se najprej lotijo. Vse naloge imajo enako število možnih točk.

Tekmovalci morajo na računalniku pripraviti čimbolj pravilno rešitev in jo oddati v sistem za preverjanje. Glede na različne vhodne podatke mora program tekmovalca izpisati pravilne odgovore. Vhodni podatki so pripravljeni tako, da preverijo, ali so tekmovalci s pravilno metodo upoštevali tudi različne mejne primere, na katere lahko naletimo.

Tekmovalci naj orodje, s katerim se bodo lotili reševanja, prilagodijo naravi problema. Če naloga zahteva uporabo regularnega izraza, se tega lotimo z orodjem, ki jih pozna. Naravna izbira so v tem primeru orodja sed, grep, awk,

perl ali python. Če znamo večji problem razbiti na zaporedje manjših in jih povezati v cevovod, bo to verjetno kar prava rešitev.

Izvorna koda

Pri slogovni podobi programov smo se v splošnem zgledovali po *Enajsti šoli računalništva*. Rezervirane besede in imena standardnih tipov pišemo z malo začetnico, ostala imena pa večinoma z veliko, razen včasih pri enočrkovnih imenih spremenljivk. V resnici se nam tudi imena `I`, `J` in podobna ne zdijo problematična, saj pri pisavah, v kakršnih običajno vidimo izvorno kodo na zaslonu (torej takih, kjer so vse črke enako široke), običajno ni velikega tveganja, da bi `I` zamenjali z `l` ali `1` ali čim podobnim. Pač pa je v pisavi, ki smo jo za prikaz izvorne kode uporabili v tej zbirki, res težko ločiti `I` od `l`, zato smo pri spremenljivkah `i` in `j` raje uporabljali male črke, pri `L` pa veliko.

Rešitve so večinoma napisane v (kolikor toliko standardnem) pascalu, saj smo tudi na zadnjih tekmovanjih še vedno opažali, da večina tekmovalcev dela v pascalu. Poleg tega so verjetno programi v pascalu kolikor toliko razumljivi tudi mnogim izmed tistih, ki večinoma sicer ne delajo v pascalu; če bi bile rešitve v C-ju ali katerem izmed njegovih bližnjih sorodnikov, bi imelo najbrž z razumevanjem teh rešitev težave več ljudi kot pa z razumevanjem pascalskih. Pri nekaj nalogah je poleg pascalske rešitve še rešitev v kakšnem drugem jeziku, če so jo sestavljalci prispevali (npr. zato, ker je krajša ali bolj elegantna od pascalske). Pri eni nalogi (1995.3.2) je rešitev le v pythonu, ker je že ta program precej dolg, v pascalu pa bi bil verjetno še znatno daljši; upamo, da bo kljub temu vsaj za silo razumljiv, saj je python znan po razmeroma priljudni sintaksi.

Nismo pa v vseh pogledih vztrajali na nekaterih tradicionalnih značilnostih pascala, na primer pri tem, da bi morale biti vse spremenljivke deklarirane pred podprogrami. Konec koncev je bolje, če podprogrami tistih globalnih spremenljivk, ki jih ne bodo potrebovali, tudi ne vidijo, saj se tako zmanjšajo možnosti, da bi jih uporabili pomotoma namesto kakšne istoimenske lokalne spremenljivke, če je programer le-to pozabil deklarirati. Tudi stavkov `exit` (skok iz trenutnega podprograma), `break` (skok iz trenutno najbolj notranje zanke) in `continue` (skok na konec iteracije trenutno najbolj notranje zanke), ki sicer niso standardni, se nismo branili uporabljati, saj lahko poenostavijo marsikatero zanko ali podprogram.

Druga odstopanja od standardnega pascala so še naslednja:

- Za zgornje in spodnje meje indeksov v tabelah ne uporabljamo vedno konstant, pač pa včasih tudi izraze, ki imajo konstantno vrednost.
  Primer: `const n = 123; var a: array [1..n * n] of integer;`

- Logični operator **xor** (izključni ali): `a xor b` je `true` natanko tedaj, ko je eden od `a` in `b` enak `true`, eden pa `false`.

- Operatorja **shl** in **shr**, ki zamakneta levi operand za določeno število bitov v levo oz. desno; desni operand pove, za koliko bitov. Tako dobimo pri a **shl** b rezultat $a \cdot 2^b$ (če ne pride do prekoračitve obsega celih števil), pri a **shr** b pa $\lfloor a/2^b \rfloor$. Primer: 12 **shl** 6 = 768; 12 **shr** 3 = 1.

- Logične operatorje **and**, **or** in **xor** uporabljamo tudi kot aritmetične operatorje nad celimi števili. Takrat ti operatorji izračunajo svojo logično operacijo nad pari istoležnih bitov v danih operandih; pri tem se prižgan bit obnaša kot vrednost true, ugasnjen pa kot false. Nekaj primerov: 12 **or** 6 = 14; 12 **and** 6 = 4; 12 **xor** 6 = 10.

V primerih, ko bi radi kakšni spremenljivki že ob deklaraciji določili začetno vrednost, pridejo prav deklaracije oblike **var** x: integer **value** 7, ki jih standardni pascal sicer nima, so pa v standardu za razširjeni pascal (extended pascal). (V nekaterih narečjih pascala je mogoče namesto **value** uporabiti tudi „:=“ ali „=“. V Turbo Pascalu pa bi morali napisati **const** x: integer = 7, pri čemer bi se x potem vseeno obnašal kot spremenljivka, ne kot konstanta.)

Standardni pascal je pri delu z nizi precej okoren, zato smo za delo z nizi večinoma uporabljali nestandardni tip **string**, ki ga imajo mnoga narečja pascala. Spremenljivke tipa **string** lahko hranijo različno dolge nize, do posameznih znakov pa pridemo tako kot pri tabelah: s[1] je prvi znak, s[2] je drugi in tako naprej; funkcija Length(s) vrne dolžino niza s; nize lahko stikamo z operatorjem +. Nekaj nalog, predvsem starejših, pa uporablja tudi bolj standardni pristop — tabele oblike **packed array** [1..MaxDolzina] **of** char.

Zančne invariante in dokazovanje pravilnosti

Ko imamo pred seboj nek algoritem za reševanje določenega problema, je načeloma koristno, če se uspemo res trdno in zanesljivo prepričati o tem, da je ta algoritem sploh pravilen (torej: da res naredi to, kar od njega pričakujemo). Res je sicer, da se zdi človeku včasih kar nekako „očitno“, da je nek postopek pravilen (sploh če ga je sestavil sam), vendar pa nas lahko taki občutki tudi varajo, poleg tega pa ni nujno, da bi bila ista stvar očitna tudi drugim. Zato smo se trudili pri mnogih rešitvah podati tudi nekakšen razmislek ali dokaz, s katerim naj bi se lahko zatrdno prepričali o pravilnosti uporabljenega postopka.

Pri takšnih dokazih pride včasih prav matematična *indukcija*. Recimo, da imamo neko trditev, ki govori o nekem naravnem številu n; označimo jo s $P(n)$; in recimo, da bi radi dokazali, da velja $P(n)$ za vsako naravno število n. Načelo matematične indukcije pravi, da je za tak dokaz dovolj, če pokažemo naslednje: da velja $P(1)$ (temu pravimo *baza indukcije*) in da za vsako naravno število n velja $P(n) \Rightarrow P(n+1)$, torej: če velja naša trditev pri n, velja tudi pri $n+1$ (temu pravimo *induktivni korak;* ko se ukvarjamo z njim, imenujemo trditev $P(n)$ tudi *induktivna predpostavka*). Možne so tudi drugačne različice, npr. da dokažemo $P(1), P(2), \ldots, P(k)$ in nato še $P(n) \wedge P(n+1) \wedge \ldots \wedge P(n+k-1) \Rightarrow$

$P(n+k)$ za vsako naravno število n; ali pa, da dokažemo $P(1)$ in nato še (za vsak n) $P(1) \wedge \ldots \wedge P(n) \Rightarrow P(n+1)$. Pri dokazovanju pravilnosti bi lahko indukcijo uporabili na primer tako, da bi za $P(n)$ vzeli izjavo, da naš algoritem pravilno reši vse primerke problema, ki so veliki n (karkoli že velikost pri našem problemu pač pomeni). Če nam uspe dokazati, da velja $P(n)$ za vsak n, smo s tem tudi dokazali, da reši naš algoritem pravilno vse primerke tega problema, ne glede na njihovo velikost. Včasih je pri izboru trditve $P(n)$, ki naj bi jo poskušali dokazati z indukcijo, potrebne nekaj iznajdljivosti: če je P „premočna“ (torej: če zatrjuje preveč), bo težko dokazati $P(n) \Rightarrow P(n+1)$, ker $P(n+1)$ zahteva preveč; po drugi strani, če je P „prešibka“ (zatrjuje premalo), bo spet težko dokazati $P(n) \Rightarrow P(n+1)$, ker nam $P(n)$ pove premalo in nam daje prešibko oporo za sklepanje na $P(n+1)$. Primera dokazov z indukcijo sta pri rešitvah nalog 1988.2.1 in 2003.3.3.

V zvezi s preverjanjem pravilnosti algoritmov velja posebej omeniti še en prijem, ki pride velikokrat prav: *zančne invariante*. Zančna invarianta je načeloma nek pogoj, ki naj bi veljal na začetku vsake ponovitve (iteracije) neke zanke. O tem, da je nek pogoj res invarianta, se prepričamo tako, da dokažemo naslednje: (1) da ta pogoj res velja ob začetku prve iteracije naše zanke; in (2) da, če velja na začetku neke iteracije, zagotovo velja tudi na začetku naslednje (pri tem moramo seveda upoštevati, kaj vse se lahko v programu zgodi ali spremeni med trenutno iteracijo; na primer, pogosto se poveča vrednost kakšnega števca). Iz tega dvojega lahko z indukcijo (glej prejšnji odstavek) pokažemo, da velja naša invarianta res na začetku vsake ponovitve opazovane zanke.

```
ZacetekZanke:  { na tem mestu velja invarianta }
               if UstavitveniPogoj then goto KonecZanke;
               Stavek1;
               Stavek2;
               ...
               StavekN;
               goto ZacetekZanke;
KonecZanke:
```

Iz zgoraj omenjenih lastnosti (1) in (2) že sledi, da velja invarianta tudi po koncu izvajanja cele zanke; za povrhu tega pa velja takrat tudi ustavitveni pogoj, zaradi katerega se je zanka sploh ustavila. Invarianto si skušamo izbrati tako, da bomo lahko iz tega dvojega (invarianta + ustavitveni pogoj) že dokazali, da je zanka opravila svoje delo. Tako je edini del celega razmisleka, ki še utegne biti problematičen, iskanje primerne invariante. Tu je včasih potrebno nekaj domiselnosti, koristne pa so tudi izkušnje, saj gredo mnogi takšni dokazi po podobnem kopitu. Primere uporabe zančnih invariant lahko vidimo v rešitvah naslednjih nalog: 1990.2.2, 1993.2.3, 1997.Z.2, 1998.3.3, 1999.1.4, 2001.1.4, 2003.3.2, 2003.3.3.

Še ena stvar, ki jo zahtevamo od algoritma, je ta, da se, ne glede na to, na kako velikem primerku problema ga poženemo, vedno zanesljivo ustavi po končnem številu korakov (čeprav je to število seveda lahko odvisno od velikosti problema). Z drugimi besedami: nočemo, da bi se nam algoritem zaciklal. To je največkrat čisto očitno, včasih pa pri kakšni zanki le ni tako zelo jasno, ali bo njen ustavitveni pogoj nekoč res zagotovo izpolnjen. V takih primerih običajno poskušamo poiskati neko vrednost (ki je npr. odvisna od vrednosti tistih spremenljivk, ki se med izvajanjem zanke spreminjajo), ki se ob vsaki ponovitvi zanke zmanjša, obenem pa se ne more zmanjšati neskončno mnogokrat (npr. ker je vedno pozitivna in se vsakič zmanjša za 1). Če nam uspe pokazati kaj takega, iz tega že sledi, da se opazovana zanka ne more izvesti več kot končno mnogokrat. Na ta način lahko včasih dobimo tudi neko koristno zgornjo mejo števila izvajanj zanke (tako da vemo, da se ne bo izvedla več kot tolikokrat; gl. npr. rešitev naloge 2003.3.3, str. 567).

Časovna in prostorska zahtevnost

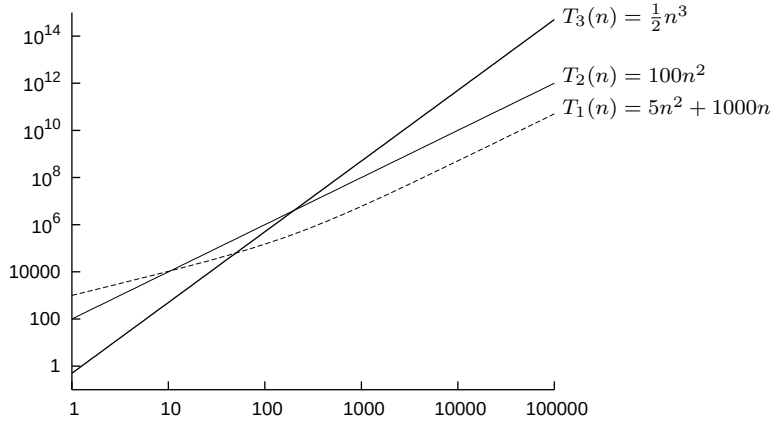
Poleg pravilnosti algoritma nas pogosto zanima tudi njegova učinkovitost; največkrat nas zanima, kako dolgo bi se izvajal, pogosto pa tudi to, koliko pomnilnika bi porabil. Nepraktično bi bilo, če bi skušali porabo časa in pomnilnika napovedovati čisto natančno, saj sta odvisni od preveč podrobnosti, ki jih pravzaprav ne poznamo (npr. od delovanja prevajalnika in od arhitekture procesorja, na katerem bo program tekel). Zato se zadovoljimo že z bolj grobimi ocenami zahtevnosti algoritma.

Običajno je naš algoritem namenjen reševanju nekega splošnega problema, ki ima veliko konkretnih primerkov. (Za primer recimo, da imamo algoritem za urejanje, ki zna urediti poljubno tabelo celih števil. Konkretni primerki tega problema so torej posamezna zaporedja celih števil, ki bi jih lahko podali našemu algoritmu kot vhodne podatke, da bi nam jih uredil.) Čas izvajanja (podobno pa je tudi s porabo pomnilnika) najbrž ne bo enak za vse primerke problema. (Na primer, urejanje večje tabele števil predvidoma traja dlje kot urejanje manjše tabele.) Zato bi si lahko časovno zahtevnost algoritma predstavljali kot funkcijo $T(x)$, ki pove, koliko časa porabi naš algoritem, če mu kot vhodni podatek podamo primerek x. Vendar pa je to še vedno nerodna in nepraktična mera, saj je lahko zaloga možnih x precej zapletena zadeva (pri našem primeru z urejanjem bi bil lahko x poljubno zaporedje celih števil). Zato raje definirajmo neko mero velikosti primerka, ki nam bo za vsak x vedela povedati njegovo velikost (recimo $d(x)$). Potem se lahko zanimamo za časovno zahtevnost kot funkcijo velikosti problema: $T(n) = \max\{T(x) : d(x) = n\}$. Tako definirana $T(n)$ nam torej pove, koliko časa bomo porabili v najslabšem primeru, če vemo, da imamo na vhodu problem velikosti n.[1] Pri našem pri-

[1] To je lahko pesimistično; mogoče je takih x zelo malo, pri večini pa bi bila poraba časa manjša. Zato bi lahko namesto funkcije max uporabili povprečje ali kaj podobnega, vendar

meru algoritma za urejanje bi lahko na primer definirali $d(x)$ kot število členov zaporedja x; potem bi nam $T(n)$ povedala, koliko časa potrebuje naš algoritem (v najslabšem primeru) za urejanje n števil.

Pri funkciji $T(n)$ nas bo najbolj zanimalo to, kako hitro narašča v odvisnosti od n-ja; druge podrobnosti pogosto kar zanemarimo. Oglejmo si na primer tri funkcije, ki jih prikazuje naslednji graf.



Čeprav je funkcija $T_2(n) = 100n^2$ pri majhnih n manjša od $T_1(n) = 5n^2 + 1000n$, pa jo sčasoma prehiti in razmerje $T_2(n) : T_1(n)$ se približuje vrednosti 20, če gledamo vse večje n. Lahko torej rečemo, da v nekem smislu obe naraščata enako hitro (namreč tako hitro kot kvadrat n-ja), le da je T_2 (pri dovolj velikih n) približno dvajsetkrat večja od T_1.

Če pa primerjamo $T_3(n) = \frac{1}{2}n^3$ s funkcijo T_2 ali T_1, spet vidimo, da je T_3 pri majhnih n sicer manjša od njiju, kasneje pa ju prehiti; vendar pa se tokrat tudi razmerje $T_3(n) : T_1(n)$ ne ustali, pač pa narašča prek vseh meja. Funkcija T_3 torej v nekem smislu narašča bistveno hitreje od T_1 in T_2, namreč s kubom n-ja, onidve pa s kvadratom.

Iz takih primerjav vidimo, da vpliva na hitrost naraščanja funkcije pri velikih n predvsem njen vodilni člen (torej tisti z najvišjo stopnjo); vpliv ostalih členov se lahko pozna le pri dovolj majhnih n. Pri vodilnem členu pa na hitrost naraščanja funkcije daleč najbolj vpliva njegova stopnja; če pa primerjamo več funkcij z isto stopnjo, naraščajo vse nekako enako hitro, razmerje med njimi pa se približuje razmerju med vodilnimi koeficienti.

Za lažje izražanje o zahtevnosti algoritmov se je uveljavil zapis z velikim O. Pravimo, da je funkcija $T(n)$ reda $O(f(n))$ (kjer je $f(n)$ tudi neka funkcija),

se s tem analiza algoritmov marsikje zelo zaplete, poleg tega pa bi se morali začeti ubadati tudi s tem, ali so res vsi x enako verjetni ali pa naj vplivajo na povprečje nekateri bolj kot drugi.

če obstaja taka konstanta c, da od nekega n naprej vedno velja neenakost $T(n) \leq c \cdot f(n)$. Z besedami bi lahko rekli, da sme $T(n)$ naraščati največ tako hitro kot $f(n)$, razen mogoče za nek konstantni faktor. Iz te definicije vidimo, da je $5n^2 + 1000n$ reda $O(n^2)$, pa tudi reda $O(n^3)$; to velja tudi za funkcijo $100n^2$; po drugi strani pa je funkcija $\frac{1}{2}n^3$ reda $O(n^3)$, ne pa tudi reda $O(n^2)$. Ta zapis nam pomaga, da se pri razmišljanju in primerjanju zahtevnosti algoritmov osredotočimo predvsem na najpomembnejše, torej na to, kako hitro narašča zahtevnost z večanjem problema, ki ga rešujemo.

Ta razmislek nas tudi napeljuje k ugotovitvi, da ni nič hudega, če s funkcijo $T(n)$ ne bomo poskušali izraziti natančne porabe časa (to bi bilo tako ali tako prezapleteno); raje si izberimo nekaj osnovnih korakov ali operacij in potem definirajmo $T(n)$ kot število takih korakov. Če bi zdaj pomnožili $T(n)$ s časom izvajanja najpočasnejšega od teh osnovnih korakov, bi dobili neko zgornjo (torej pesimistično) mejo za pravi čas izvajanja našega algoritma. Toda ker smo $T(n)$ pomnožili le z neko pozitivno konstanto, nismo na njen red zahtevnosti nič vplivali. Zato nam že poenostavljena $T(n)$, torej taka, ki le šteje osnovne operacije, pove dovolj o redu časovne zahtevnosti algoritma.[2]

V zadnjih nekaj odstavkih smo se zanimali predvsem za to, kaj se dogaja s časovno zahtevnostjo algoritma, če mu dajemo vse večje vhodne primere (vse večje n). Če bi nas zanimali majhni n, so seveda medsebojna razmerja med algoritmi lahko precej drugačna. Običajno že pri ne tako zelo velikih n prevladajo asimptotične lastnosti, vendar pa ni vedno tako (glej na primer rešitev naloge 2001.1.4, str. 457, in naloge 1999.U.1, str. 696).

[2]To, kaj si izberemo za osnovne korake, je lahko odvisno od algoritma in od problema, s katerim se ukvarjamo. Na primer, za seštevanje dveh števil običajno vzamemo, kot da je to nekak atomaren osnovni korak, za katerega porabimo vedno le konstantno mnogo časa. Po drugi strani pa je le res, da bi, če bi hoteli delati s poljubno velikimi števili, prej ali slej morali takšna števila predstaviti s tabelami in potem na njih izvajati postopek, ki bi posnemal ročno seštevanje; to pa ne traja več konstantno mnogo časa, ampak tem dlje, čim več števk imajo števila, s katerimi računamo. Zato, če bi vedeli, da bo naš postopek delal z res zelo velikimi števili (velikimi tudi v primerjavi z n-jem) in da se cene teh računskih operacij ne bodo utopile v ceni česa drugega, kar bo naš algoritem tudi še počel, bi bilo bolj pošteno vzeti za osnovni korak le seštevanje dveh števk, ne pa seštevanje dveh poljubno velikih števil.

Matematične oznake

$\lfloor x \rfloor$	x, zaokrožen navzdol: največje celo število, manjše ali enako x
$\lceil x \rceil$	x, zaokrožen navzgor: najmanjše celo število, večje ali enako x
$\lg x$	dvojiški logaritem x: $\lg x = (\ln x)/(\ln 2)$ in $2^{\lg x} = x$
$n \operatorname{div} d$	celi del količnika pri deljenju: $\lfloor n/d \rfloor$
$\sum_{i=m}^{n} a_i$	vsota $a_m + a_{m+1} + \ldots + a_{n-1} + a_n$
$\sum_{i:P(i)} a_i$	vsota vrednosti a_i za vse take i, ki ustrezajo pogoju $P(i)$
$\prod_{i=m}^{n} a_i$	produkt $a_m \cdot a_{m+1} \cdot \ldots \cdot a_{n-1} \cdot a_n$
$n!$	„n fakulteta“, $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$
$\binom{n}{k}$	binomski koeficient, „n nad k“: $\binom{n}{k} = n!/(k!(n-k)!)$, množica z n elementi ima $\binom{n}{k}$ podmnožic s k elementi
H_n	n-to harmonično število, $H_n = \sum_{k=1}^{n} 1/k$; $H_n \approx \ln n + \gamma + O(1/n)$, pri čemer je $\gamma = 0{,}577215\ldots$ za vsak n velja: $\ln n + \frac{1}{2} + \frac{1}{2n} \leq H_n \leq \ln n + 1$

Literatura

Nekaj knjig o **algoritmih in podatkovnih strukturah**:

R. Sedgewick: *Algorithms in {C, C++, Java}*. Addison-Wesley, 1997–2003. Algoritmi so tu opisani kar z izvorno kodo v konkretnih programskih jezikih (zato tudi obstaja več izdaj z različnimi programskimi jeziki).

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: *Introduction to Algorithms*. MIT Press, 2001. Izčrpna in temeljita, čeprav se zna občasno komu zdeti tudi malo puščobna in preveč pedantna. Prva izdaja, brez četrtega avtorja, je iz leta 1990 in je tudi še čisto uporabna. Algoritmi so opisani v lepo berljivi psevdokodi.

F. P. Preparata, M. I. Shamos: *Computational Geometry*. Springer, 1991. Klasična knjiga o računski geometriji, čeprav verjetno obstaja tudi že kaj novejšega.

D. E. Knuth: *The Art of Computer Programming*. V treh zvezkih, Addison-Wesley, 1997, 1998. Zelo obsežno delo; če v kakšni tanjši knjigi ne najdemo tistega, kar nas zanima, ne bo škodilo, če poskusimo srečo še pri Knuthu.

T. W. Parsons: *Introduction to Algorithms in Pascal*. John Wiley & Sons, 1994. Prijazen uvod v algoritme; koda v pascalu; ne pretirava s teorijo, dokazi, formulami in podobnim.

R. Neapolitan, K. Naimipour: *Foundations of Algorithms using C++ Pseudocode*. Jones and Bartlett, 1998. Poudarek na algoritmih. Psevdokoda, temelječa bolj na C++ kot na pascalu. Knjiga se načeloma trudi izogibati se pretiravanju z matematiko, formalnostmi, formulami.

I. Kononenko: *Načrtovanje podatkovnih struktur in algoritmov*. 2. izdaja, FRI, Ljubljana 1999. V prvi izdaji iz leta 1996 je precej napak. Algoritmi so opisani s pascaloidno psevdokodo. Ne pretirava s formalnostmi. Poudarek na podatkovnih strukturah.

I. Kononenko, M. Robnik Šikonja: *Algoritmi in podatkovne strukture 1*. FRI, Ljubljana 2003. Prenovljena različica prejšnje knjige. Algoritmi so opisani v javi.

B. Vilfan: *Osnovni algoritmi.* 2. izdaja, FRI, Ljubljana 2002. Prva izdaja iz 1998 ima precej napak. Poudarek na algoritmih. Koda v oberonu (sorodnik pascala). Občasno precej matematična.

N. Wirth: *Računalniško programiranje.* DMFA, Ljubljana 1979 (prvi del), 1983 (drugi del). Prevedel B. Vilfan. Prevod Wirthovih knjig *Systematic Programming — An Introduction* in *Algorithms + Data Structures = Programs.* Tako govorita knjigi po eni strani o algoritmih in podatkovnih strukturah, po drugi strani pa sta tudi uvod v strukturirano programiranje oz. v to, kako se programiranja in reševanja problemov lotiti čim bolj sistematično.

Zbirke nalog (kjer ni navedeno drugače, so naloge po težavnosti večinoma primerljive s prvo in drugo skupino pričujoče zbirke):

V. Batagelj *et al.: Enajsta šola računalništva.* DMFA, Ljubljana 1988. Zbirka nalog s prvih enajstih republiških tekmovanj (1977–1987). Naloge pokrivajo vse tri težavnostne skupine. Vse so rešene v pascalu in dobro razložene. Odlično uvodno poglavje o reševanju problemov, lepem programiranju itd.

M. Azarov Domajnko, M. Kastelic: *Algoritmi in programski jeziki: zbirka rešenih nalog za 1. letnik srednjih šol.* Tehniška založba Slovenije, Ljubljana, 1997, 1999, 2001, 2002.

T. Lončarič: *Algoritmi in programski jeziki: zbirka rešenih nalog za 2. letnik srednjih šol.* Tehniška založba Slovenije, Ljubljana, 1997, 1999, 2002.

J. Kozak, M. Lokar: *Naloge iz računalništva.* DMFA, Ljubljana 1988. Tu je tudi precej težjih in bolj teoretičnih nalog. Ta zbirka se lepo poda k učbenikom algoritmov in podatkovnih struktur, naštetim v prejšnjem razdelku.

M. Gradišar: *Programiranje v pascalu: zbirka vaj.* DMFA in Visoka šola za organizacijo dela, Ljubljana in Kranj 1983.

M. Juvan, M. Zaveršnik: *Vaje iz programiranja: C, C++ in Mathematica.* Študentska založba, Ljubljana 2000.

M. Juvan, M. Lokar: *121 nalog iz Pascala.* DMFA, Ljubljana 1992.

M. Juvan, M. Zaveršnik: *C naj bo: zbirka rešenih nalog.* DMFA, Ljubljana 1999.

I. Fajfar: *Praktično programiranje v jeziku C.* FE, Ljubljana 1998.

Knjige tipa „**uvod v programiranje**“:

I. Bratko, V. Rajkovič: *Računalništvo s programskim jezikom pascal.* DZS, Ljubljana 1989. Pregleden uvod v programiranje (od 8. poglavja naprej; pred tem so še poglavja o tem, kaj so računalniki, kako delujejo itd.).

B. Mohar, E. Zakrajšek: *Uvod v programiranje.* DMFA, Ljubljana 1982. Sistematičen uvod v programiranje s pascalom.

M. Lokar: *Prvi koraki v programski jezik C.* DMFA, Ljubljana 2000. Kratek, razumljiv uvod v C.

R. P. Halpern: *C for Yourself: Learning C using Experiments*. OUP, 1997. Prijazen uvod v C z veliko primeri in „eksperimenti“ tipa „poskusi sam“.

B. W. Kernighan, D. M. Ritchie: *Programski jezik C.* FER, Ljubljana 1991. Prevedel L. Mlakar. Klasičen in temeljit pregled jezika C (ki se je medtem v nekaterih podrobnostih sicer že malo spremenil).

Ž. Turk: *Programski jezik C.* ZOTKS, Ljubljana 1987.
F. Bratkovič: *Uvod v C.* FER, Ljubljana 1994.
V. Žumer, J. Brest: *Strukturirano programiranje v C++.* FERI, Maribor 2001.
V. Žumer, J. Brest: *Uvod v programiranje in programski jezik C++.* FERI, Maribor 2002.

Nekaj knjig o tematikah, ki jih pokriva tekmovanje v poznavanju **Unix**a:

W. R. Stevens: *Advanced Programming in the* UNIX *Environment.* Addison-Wesley, 1992.
A. Košir, R. Maurer, R. Papež, P. Peterlin, M. Tomšič: *Linux z namizjem* KDE*: priročnik za delo z operacijskim sistemom Linux.* 2. izdaja, Pasadena, 2003. `http://www.pingo.org/knjiga/web/`
B. W. Kernighan, R. Pike: *The Unix Programming Environment.* Prentice Hall, 1984.
J. Peek, T. O'Reilly, M. Loukides: *Unix Power Tools.* 3. izdaja, O'Reilly, 2002.
L. Wall, T. Christiansen, J. Orwant: *Programming Perl.* 3. izdaja, O'Reilly, 2000.
D. Dougherty, A. Robbins: *sed and awk.* 2. izdaja, O'Reilly, 1997.

Nekaj koristnih **spletnih naslovov**:

`http://online-judge.uva.es/` — arhiv nalog s študentskih tekmovanj v programiranju, ki jih organizira združenje ACM. Strežnik ponuja tudi avtomatsko preverjanje rešitev (uporabnik pošlje izvorno kodo, strežnik pa jo prevede in preizkusi na neznanih testnih primerih).

`http://olympiads.win.tue.nl/ioi/` — domača stran mednarodnih srednješolskih računalniških olimpijad. Tu je tudi arhiv starih nalog in testnih podatkov.

`http://ace.delos.com/usacogate/` — več kot le strežnik z nalogami: tu so tudi razlage in opisi mnogih tehnik, ki pridejo prav pri reševanju nalog in načrtovanju algoritmov.

N. J. A. Sloane (ur.): *The On-Line Encyclopedia of Integer Sequences* (OEIS), `http://www.research.att.com/~njas/sequences/`. Zbirka celoštevilskih zaporedij; za mnoga zaporedja so navedene razlage, formule, koristna literatura ipd.

E. W. Weisstein (ur.): *Eric Weisstein's World of Mathematics*, `http://mathworld.wolfram.com/`. Zbirka definicij raznih matematičnih pojmov; veliko formul, grafov, kazalcev na literaturo.

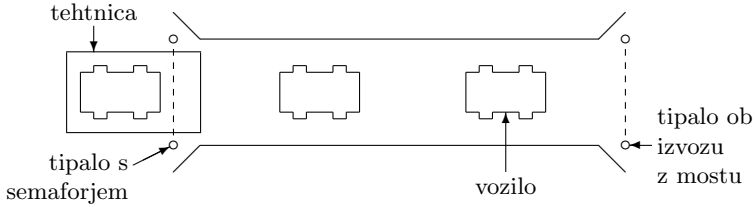
12. republiško tekmovanje v znanju računalništva (1988)

NALOGE ZA PRVO SKUPINO

R: 28

1988.1.1

Dolg ozek most, po katerem teče promet le v eno smer, zdrži skupno obremenitev 20 ton. Na mostu je lahko hkrati več vozil, vendar prehitevanje ni možno. Tik pred mostom je tehtnica, ki meri težo naslednjega vozila, ki bo zapeljalo na most. Ob tehtnici je tudi semafor, s katerim lahko prepovemo ali dovolimo, da bi vozilo s tehtnice zapeljalo na most. Na obeh straneh mostu sta tipali, ki povesta, kdaj kakšno vozilo zapelje z mostu oziroma s tehtnice nanj. Predpostaviš lahko, da je most na začetku prazen.



Za upravljanje semaforja želimo uporabiti računalnik. **Napiši algoritem**, po katerem bo računalnik upravljal semafor. **Opiši podatkovno strukturo**, ki jo potrebujemo za opis stanja na mostu.

R: 29

1988.1.2

Napiši program, ki prebere vrstico z nekaj besedami, ki so ločene z enim ali več presledki, in besede izpiše v obratnem vrstnem redu.[3] Primer:

```
Prijatlji! odrodile so trte vince nam sladkó
```

naj program prepiše v

```
sladkó nam vince trte so odrodile Prijatlji!
```

R: 30

1988.1.3

Iz majhnega računalnika, številčnega zaslona in tipke želimo sestaviti štoparico. Na zaslonu je prostor za prikaz ur, minut, sekund in stotink sekunde. S tipko upravljamo delovanje štoparice takole:

[3] Podobna naloga je tudi 2000.U.1 (str. 685, rešitev na str. 701).

- dolg pritisk na tipko (daljši od pol sekunde) postavi čas na 0 in ustavi štoparico, ne glede na to, ali je prej tekla ali ni;
- krajši pritisk na tipko požene ustavljeno ali pa ustavi tekočo štoparico.

Napiši program, ki bo krmilil štoparico. Na razpolago imaš naslednje podprograme:

function Tipka vrne true, če je tipka pritisnjena, sicer vrne false;

function Impulz vrne true, če je bila od prejšnjega klica te funkcije dopolnjena nova stotinka sekunde;

procedure Zaslon(h, m, s, cs) zapiše novo vrednost na zaslon; h, m, s, cs so cela števila — ure, minute, sekunde in stotinke sekunde.

Predpostaviš lahko, da je računalnik dovolj hiter, da ob rednem klicanju podprograma Impulz ne spregleda nobene stotinke sekunde.

1988.1.4

Imamo naslednji program: R: 31

```
program Naloga(Input, Output);
var a, d, k, p, q: integer;
begin
  d := 0; ReadLn(a);
  for k := 2 to a do begin
    p := 2; q := k div p;
    while p < q do begin p := p + 1; q := k div p end;
    if (p = q) and ((k mod p) = 0) then d := d + 1;
  end; {for}
  WriteLn(d);
end. {Naloga}
```

(a) Kaj navedeni program izpiše, če za a podamo vrednost 99? Kaj, če podamo 100?

(b) Kaj navedeni program dela? Utemelji!

NALOGE ZA DRUGO SKUPINO

1988.2.1

Datoteka na disku je organizirana kot seznam blokov, ki so poljubno razmetani po disku. Zato vsak blok poleg vsebine datoteke vsebuje še naslednja podatka: R: 32

- naslov naslednjega bloka datoteke na disku; ter

- naslov bloka z dvakratno zaporedno številko.

Do nekega bloka datoteke lahko tako pridemo po več poteh. Primer:

$$1 \to 2 \to 3 \to 4 \to \cdots \to 16 \to 17 \text{ (zaporedoma) ali pa}$$
$$1 \to 2 \to 3 \to 6 \to 7 \to 8 \to 16 \to 17$$

in še kako drugače.

Vsaka puščica seveda zahteva dostop na disk in branje bloka podatkov. **Napiši algoritem**, ki za dano številko bloka poišče najkrajšo možno pot od začetka datoteke do zahtevanega bloka v njej! **Kako** bi to pot čim krajše opisal? **Najmanj koliko** bitov je potrebnih za opis poti do blokov s številkami, manjšimi ali enakimi 2^n?

R: 34

1988.2.2

Uporabniki nekega programa so se pritožili, da so ukazne besede predolge. Zato želimo vpeljati možnost okrajševanja ukazov. Ukaz `dodaj` naj bi bilo na primer mogoče okrajšati na `dod`, `doda` ali `dodaj`; ne pa na `do` (prekratka okrajšava) ali `dodamo` (napačni znaki v ukazu). Okrajšavo opišemo tako, da z zvezdico v modelu ukazne besede označimo, do kam sega obvezni del ukaza; v našem primeru `dod*aj`. Okrajšani ukaz mora biti dolg najmanj en znak (v modelu ukazne besede zvezdica nikoli ne stoji na prvem mestu).

Napiši podprogram, ki ugotovi, ali nek niz ustreza na zgornji način podanemu modelu ukazne besede. Model ukazne besede in preverjani niz sta v ustreznih tabelah znakov in ju zaključuje vsaj en presledek.

R: 35

1988.2.3

Dve ločeni osi, med katerima je sklopka, se vrtita z različnima hitrostma v isto smer. Pri tem lahko krmilimo prvo os s pomočjo podprograma:

```
procedure Pogon(Pospesek: real);
```

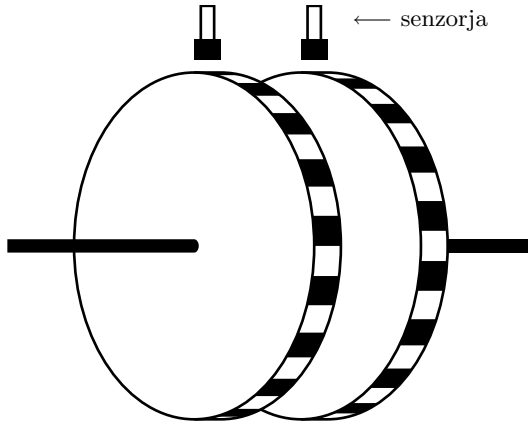
ki določa pospešek (Pospesek > 0) oziroma pojemek (Pospesek < 0) vrtenja. Upoštevaj, da motor, ki ga upravlja program, ne zmore večjega pospeška kot $+1$ in ne pojemka, večjega od -1. Hitrost vrtenja druge osi, ki se lahko počasi spreminja, lahko le opazujemo, nanjo pa ne moremo vplivati.

Vsaka os je po obodu označena z enakomerno debelimi črnimi in belimi pasovi. Za odčitavanje oznak imamo na voljo funkcijo:

```
function Oznaka(Os: integer): boolean;
```

ki pove za zahtevano os (število 1 ali 2), ali je ob senzorju črn pas ali ne (glej sliko na str. 25).

Za svoje delo imamo na voljo še funkcijo



Ilustracija k nalogi 1988.2.3.

---

function Cas: integer;

ki vrne trenutni čas v milisekundah od zagona programa; pri tem zanemarimo možnost prekoračitve obsega celih števil.

Napiši program, ki bo krmilil vrtenje prve osi tako, da se bo vrtela enako hitro kot druga os in bo na koncu s pomočjo podprograma Sklopi spojil obe osi, da se bosta vrteli skupaj. Osi je dovoljeno sklopiti šele pri dovolj majhni razliki hitrosti vrtenja obeh osi. Pri tem upoštevaj, da je računalnik zelo hiter v primerjavi z vrtenjem osi ter da so spremembe hitrosti majhne v primerjavi s hitrostjo vrtenja.

1988.2.4

Imamo naslednji program:

R: 36

```
program Naloga(Output);
const
  mCif = 4;
  mStv = 9999;
  pStv = 2;
var
  cif, stv, rbo, obr, vst: integer;
begin
  vst := 0;
  for stv := pStv to mStv do begin
    rbo := stv; obr := 0;
    for cif := 1 to mCif do
      begin obr := 10 * obr + (rbo mod 10); rbo := rbo div 10 end;
    vst := vst + (stv − obr);
```

```
  end; {for}
  WriteLn(vst);
end. {Naloga}
```

(a) **Kaj izpiše** podani program? Kaj bi program izpisal, če bi bila konstanta pStv enaka 1 namesto 2?

(b) **Kaj dela** podani program? Utemelji!

NALOGE ZA TRETJO SKUPINO

R: 37

1988.3.1

Na voljo imamo računalniški sistem z 10 enakimi procesorji. Vsak procesor lahko uporablja skupen pomnilnik na varen način, ne da bi ga pri tem motili ostali procesorji. **Napiši algoritem**, ki bo na tem sistemu kar najhitreje poiskal vsa praštevila med 1 in 100000! **Koliko hitreje** se izvede tvoj program (na vsakem procesorju teče svoja kopija) na tem sistemu kot na enoprocesorskem sistemu? Za usklajevanje dela med procesorji imaš na razpolago naslednje podprograme:

MojaOznaka(Oznaka) vrne v spremenljivki Oznaka številko procesorja (med 0 in 9), na katerem teče program;

Zaseden označi, da je procesor, ki izvaja ta klic, zaseden;

Prost označi, da je procesor, ki izvaja ta klic, prost;

VsiProsti je funkcijski podprogram, ki vrne vrednost true, ko so vsi procesorji prosti (nobeden še ni s klicem podprograma Zaseden označil, da je zaseden, ali pa so vsi, ki so bili zasedeni, s klicem podprograma Prost označili, da so prosti); če je vsaj en procesor zaseden, je vrednost funkcije false. Ob zagonu sistema ima funkcija vrednost true.

VsiZasedeni je funkcijski podprogram, ki vrne vrednost true, ko so vsi procesorji zasedeni (vsak je že vsaj enkrat klical Zaseden in po svojem zadnjem klicu podprograma Zaseden še ni poklical podprograma Prost), sicer pa vrne False.[4]

R: 41

1988.3.2

Sestavi program, ki prešteje, kolikokrat se v nekem nizu znakov pojavi nek drug niz. Pri tem ni nujno, da znaki drugega niza v prvem stoje zaporedoma, ujemati se mora le vrstni red.

[4] V besedilu te naloge, kakršno se nam je ohranilo v biltenu tekmovanja za leto 1988, funkcije VsiZasedeni ni (pač pa le MojaOznaka, Zaseden, Prost in VsiProsti). Izkaže se, da si je samo s temi ostalimi funkcijami težko pomagati pri sinhronizaciji naših paralelno delujočih procesorjev (tudi rešitev v biltenu za leto 1988 ima pri sinhronizaciji napake). Zato smo zdaj dodali še funkcijo VsiZasedeni, bralec pa lahko poskusi razmisliti tudi o tem, kako bi skrbel za sinhronizacijo, če ne bi imel na voljo tu opisanih funkcij (razen MojaOznaka).

```
Primer:  MATI   v  MATEMATIKA     ali     SENO   v  SOSEDNOST
     1           MAT    I                        S  E NO
     2           MA     TI                         SE NO
     3           M     ATI
     4                MATI
```

1988.3.3

R: 42

Na laserski gramofonski plošči je glasbena informacija zapisana v digitalni obliki tako, da je za vsako sekundo podanih 44100 vzorcev zvočne amplitude (vzorec je celo število med 0 in 65535). Na ta način je možno digitalizirati poljuben zvok iz človekovega slušnega območja. Pri zapisovanju in kasnejšem branju lahko pride do napak v zapisu, ki jih mora ustrezno vezje čimbolje zakriti (seveda čudeži niso možni, če je izgubljene preveč informacije). To dela takole:

(a) Če je poškodovanih do največ 9 zaporednih vzorcev, jih nadomesti z navadno linearno funkcijo od zadnjega pravilnega vzorca pred napako do prvega pravilnega po napaki.

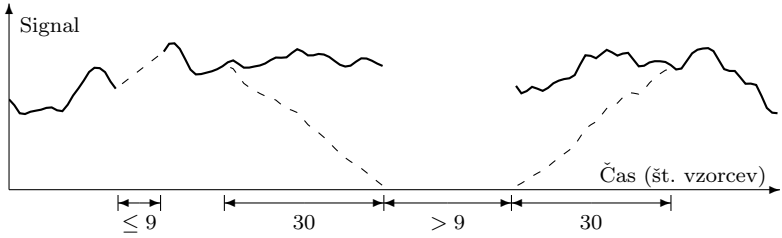
(b) Če je poškodovanih več kot 9 zaporednih vzorcev, se ti nadomestijo z ničlami, razen tega se zadnjih trideset vzorcev pred začetkom napak linearno utiša in prvih trideset po napaki primerno ojača.

Napiši program, ki je skrit v opisanem vezju. Na voljo imaš podprograma:

DobiVzorec(Vzorec, Pravilen), ki dobi s plošče vzorec in podatek o pravilnosti vzorca (je/ni pravilen);

PosljiVzorec(Vzorec), ki pošlje vzorec digitalno/analognemu pretvorniku.

Predpostavi, da je delovanje programa dovolj hitro za sprotno branje in obdelavo vzorcev.



1988.3.4

R: 44

Pri prenosu sporočil prek javnih sredstev (pošta, telegram, elektronska pošta itd.) želimo zagotoviti tajnost. To storimo tako, da pošiljatelj sporočilo zakodira (predela svoje besedilo x v drugo besedilo $y = f(x)$, pri tem pa funkciji f pravimo *kodirna funkcija*), prejemnik pa nad zakodiranim besedilom y uporabi obratno funkcijo g (*razkodirna funkcija*), ki predela prejeto besedilo y v prvotno besedilo $x = g(y) = g(f(x))$.

Da ima skupina ljudi zagotovljeno tajnost sporočil pri njihovi medsebojni izmenjavi, vsakemu človeku določimo njemu lasten par funkcij f in g, za kateri velja:

$$f(g(x)) = g(f(x)) = x.$$

Funkciji f in g morata biti seveda izbrani tako, da je kljub poznavanju ene od funkcij ter nekaj primerkov nekodiranega in zakodiranega besedila v sprejemljivem času nemogoče izračunati njej obratno funkcijo.

Vsakdo eno od svojih funkcij zadrži kot svojo skrivnost (denimo f), drugo pa (denimo g) uvrsti v javni kodirni imenik. Če torej želi Aleš poslati sporočilo Bojanu, poišče v kodirnem imeniku Bojanovo javno kodirno funkcijo in z njo zakodira sporočilo. Razkodira ga lahko le Bojan s svojo skrivno funkcijo.

Javnost kodirnih funkcij odpira nov problem: vsakdo lahko napiše sporočilo Bojanu in se v njem izdaja za Aleša (vsakdo lahko zakodira sporočilo z Bojanovo javno kodirno funkcijo ter mu ga pošlje). Ta se ne more na noben zanesljiv način prepričati, da sporočilo, zakodirano po zgornjem postopku, res prihaja od Aleša in ne od koga drugega, ki se šali na račun obeh.

Ali lahko samo z uporabo opisanih parov kodirnih funkcij posameznih članov skupine (po ene javne in ene skrivne) Aleš sporočilo zakodira tako, da bo Bojan povsem prepričan, da sporočilo prihaja zares od njega? **Razloži** rešitev!

REŠITVE NALOG ZA PRVO SKUPINO

N: 22

R1988.1.1

Ker je tehtnica postavljena le pri vstopu na most, si moramo težo vozila zapomniti, da jo bomo lahko, ko bo vozilo odpeljalo z mostu, odšteli od skupne obremenitve mostu. Ker je na mostu lahko hkrati več vozil, med seboj pa se ne morejo prehitevati, bo primerna podatkovna struktura vrsta, katere elementi so teže posameznih vozil. Vrsto lahko realiziramo s tabelo, indeksoma najstarejšega in najnovejšega elementa v njej in s spremenljivko, ki hrani trenutno število elementov v vrsti. Če pa nimamo nobene pametne zgornje meje za število vozil na mostu in zato ne vemo, kolikšno tabelo pripraviti, bi lahko vrsto realizirali tudi s seznamom, v katerem so posamezni elementi povezani s kazalci.

```
SkupnaTeza := 0; IzprazniVrsto;
while true do begin
  Tehtaj(TezaVozila); { odčitamo težo na tehtnici; lahko je tudi 0 }
  if SkupnaTeza + TezaVozila <= DovoljenaObremenitev
    then prižgi zeleno luč else prižgi rdečo luč;
  if vozilo zapeljalo na most then begin
    SkupnaTeza := SkupnaTeza + TezaVozila
    DodajVVrsto(TezaVozila);
  end;
```

```
  if vozilo zapeljalo z mostu then begin
    TezaVozila := najstarejši element v vrsti; SkrajsajVrsto;
    SkupnaTeza := SkupnaTeza − TezaVozila;
  end; {if}
end; {while}
```

Ta rešitev predpostavlja, da tehtnica v trenutku, ko tipalo na mostu zazna vozilo, še vedno odčituje celotno težo vozila (tako kaže tudi slika pri besedilu naloge). Predpostavlja tudi, da tipali sporočita mimovozeče vozilo le trenutno, torej je za vsako vozilo odčitana vrednost true le ob prvem odčitavanju, ne pa ob vsakokratnem odčitavanju, dokler se vozilo pelje mimo tipala.

R1988.1.2

N: 22

Ko beremo niz, si lahko v neki globalni spremenljivki (vBesedi v spodnjem programu) zapomnimo, ali smo trenutno znotraj besede ali ne. Tako lahko opazimo začetek besede (če doslej nismo bili v besedi, trenutni znak pa je nekaj drugega kot presledek). Začetke besed si zapomnimo v neki tabeli (Zacetek). Potem se lahko z zanko **for** sprehodimo po tej tabeli od konca proti začetku in tako izpišemo obrnjeno zaporedje besed.

```
program ObrniVrstniRed(Input, Output);
const
  mVrsta = 200;          { največje število znakov v vrsti }
  mBesed = 100;          { največje možno število besed v vrsti }
type
  VrstaT = array [1..mVrsta + 1] of char;
  ZacetekT = array [1..mBesed] of integer;
var
  Zacetek: ZacetekT;     { indeksi začetkov besed }
  BesL: integer;         { število shranjenih začetkov besed }
  Vrsta: VrstaT;         { vrstica z besedami }
  VrstaL: integer;       { dolžina vrstice z besedami }
  vBesedi: boolean;      { pove, ali smo pri branju v besedi }
  Znak: char;
  j, Bes: integer;
begin
  VrstaL := 0; BesL := 0; vBesedi := false;
  { Preberemo vrstico z besedami in shranimo indekse začetkov besed. }
  while not Eoln do begin
    Read(Znak); VrstaL := VrstaL + 1; Vrsta[VrstaL] := Znak;
    if Znak = ' ' then vBesedi := false
    else if not vBesedi then
      begin vBesedi := true; BesL := BesL + 1; Zacetek[BesL] := VrstaL end;
  end; {while}
  { Na konec vrste dodamo presledek za stražarja. }
  Vrsta[VrstaL + 1] := ' ';
  { Izpišemo besede v obratnem vrstnem redu. }
```

```
  for Bes := BesL downto 1 do begin
    j := Zacetek[Bes]; if Bes < BesL then Write(' ');    {⋆}
    while Vrsta[j] <> ' ' do begin Write(Vrsta[j]); j := j + 1 end;
  end; {for}
  WriteLn;
end. {ObrniVrstniRed}
```

Da bi lahko gornji program varno uporabljali v praksi, bi mu morali dodati še preverjanje prekoračitve največje dolžine vrstice in največjega števila besed v vrstici.

Gornji program med dvema zaporednima besedama vedno napiše po en presledek, čeprav je bilo v vhodnih podatkih tam mogoče več zaporednih presledkov. Če bi hoteli ohranjati vse presledke, bi lahko vrstico {⋆} zamenjali z nečim takšnim:

```
    if Bes < BesL then begin { Izpišemo presledke med besedama Bes in Bes + 1. }
      j := Zacetek[Bes + 1]; while Vrsta[j − 1] = ' ' do j := j − 1;
      while Vrsta[j] = ' ' do begin Write(Vrsta[j]); j := j + 1 end;
    end; {if}
    j := Zacetek[Bes];
```

Pa še primer rešitve v pythonu, ki kaže, da lahko problem včasih rešimo lažje, če se ga lotimo s primernim orodjem:

```
import sys
besede = sys.stdin.readline().split()  # preberemo vrstico in jo razbijemo na besede
besede.reverse()                       # obrnemo seznam besed
print " ".join(besede)  # staknemo jih skupaj (vmes postavimo presledke) in izpišemo
```

Šlo bi celo z enim samim stavkom, čeprav to verjetno že ni več primer lepega programiranja:

```
print " ".join(reversed(__import__("sys").stdin.readline().split()))
```

Funkcija `__import__` naloži zahtevani modul in vrne referenco nanj. Za obračanje seznama besed smo uporabili funkcijo `reversed`, ki vrne objekt (iterator), ki zna naštevati elemente danega seznama v obrnjenem vrstnem redu. Tak iterator lahko podamo kot parameter metodi `join`, da besede spet staknemo skupaj v en sam niz.

N: 22

R1988.1.3

Prejšnje stanje tipke si zapomnimo v spremenljivki `tPrej`; z njeno pomočjo lahko ugotovimo, kdaj je uporabnik pritisnil ali spustil tipko. Ko uporabnik pritisne tipko, spremenimo stanje ure (spremenljivka `Tece`) in začnemo odštevati čas (spremenljivka `Brisi`): če ostane pritisnjena dovolj dolgo, bomo postavili (ko bo padla `Brisi` na 0) čas na 0. Če pa tipko prej spusti, postavimo `Brisi` takoj na 0, kar nam zagotavlja, da se do naslednjega pritiska na tipko s to spremenljivko ne bomo več ukvarjali.

```
program Stoparica;
var
  Tece: boolean;          { ura teče }
  Brisi: integer;         { 0 ali čas v stotinkah do brisanja časa }
  t, tPrej: boolean;      { trenutno in prejšnje stanje tipke }
  h, m, s, cs: integer;   { ure, minute, sekunde, stotinke }

  function Tipka: boolean; external;
  function Impulz: boolean; external;
  procedure Zaslon(h, m, s, cs: integer); external;

begin {Stoparica}
  Tece := false; Brisi := 0; tPrej := false;
  h := 0; m := 0; s := 0; cs := 0; Zaslon(h, m, s, cs);
  repeat
    t := Tipka;
    if t and not tPrej then begin Tece := not Tece; Brisi := 50 end
    else if tPrej and not t then Brisi := 0;
    tPrej := t;
    if Impulz then begin
      if Brisi > 0 then begin
        Brisi := Brisi - 1;          { merimo čas do brisanja časa }
        if Brisi = 0 then begin      { čas je za brisanje zaslona }
          h := 0; m := 0; s := 0; cs := 0; Zaslon(h, m, s, cs);
          Tece := false;             { po brisanju naj ura stoji }
        end; {if}
      end; {if}
      if Tece then begin             { če ura teče, popravimo zaslon }
        cs := cs + 1;
        if cs >= 100 then begin
          s := s + 1; cs := cs - 100;
          if s >= 60 then begin m := m + 1; s := s - 60 end;
          if m >= 60 then begin h := h + 1; m := m - 60 end;
          if h >= 24 then h := h - 24;
        end; {if}
        Zaslon(h, m, s, cs);
      end; {if Tece}
    end; {if Impulz}
  until false;
end. {Stoparica}
```

R1988.1.4

Program prešteje, koliko je popolnih kvadratov od 2 do prebranega števila a. Za štetje uporablja spremenljivko d. Ta se poveča za ena natančno tedaj, ko veljata hkrati pogoja: N: 23

$$p = q \qquad \text{in} \qquad (k \bmod p) = 0.$$

Ker je $q = k \operatorname{div} p$ in je ostanek tega deljenja takrat 0, velja $p \cdot q = k$; in ker je $p = q$, velja takrat tudi $p \cdot p = k$, torej se d poveča natanko tedaj, ko je k popoln kvadrat.

Popolnih kvadratov med 2 in vključno 99 je osem, med 2 in 100 pa devet. V prvem primeru program torej izpiše 8, v drugem pa 9. V splošnem torej program izpiše $\lfloor \sqrt{a} \rfloor - 1$.

REŠITVE NALOG ZA DRUGO SKUPINO

N: 23

R1988.2.1

Pot lahko opišemo s tem, da pri vsakem prebranem bloku povemo, katerega od nadaljevalnih naslovov bomo uporabili: tistega, ki kaže na naslednji blok, ali tistega, ki kaže na blok z dvakrat večjo številko. Za to nam za vsak korak na poti zadostuje po en bit.

Najkrajša pot je tista, pri kateri kar največkrat uporabimo skok na blok z dvakratno številko, saj z njim pridemo najdlje.[5] Najlaže to pot določimo v smeri od zadnjega bloka proti začetku datoteke. Če je številka bloka sodo število, do tega bloka najhitreje pridemo iz bloka s pol manjšo številko; do bloka z liho številko pa ne moremo drugače kot iz predhodnega bloka. Ostale podrobnosti so razvidne iz programa.

```
const MaxPot = 32;                { največja dolžina poti }
type SmerT = (Naprej, Skok);      { tip koraka do naslednjega bloka }
     PotT = packed array [1..MaxPot] of SmerT;        { opis poti }

procedure PotDoBloka(StZap: integer; var Pot: PotT);
{ Določi najkrajšo pot do bloka datoteke na disku. }
var
  Korak: integer;      { števec korakov na poti }
  Blok: integer;       { števec blokov, skozi katere gre pot }
  Zadnji: integer;     { zadnji korak }
  t: SmerT;
begin
  { Najprej določimo pot v smeri od bloka nazaj proti začetku datoteke. }
  Korak := 0; Blok := StZap;
  while Blok > 1 do begin
```

[5] Natančneje povedano: nobena rešitev nima strogo več skokov kot najkrajša. O tem se bomo prepričali kasneje, ko bomo videli, da ima najkrajša pot do bloka B natanko $\lfloor \lg B \rfloor$ skokov. Če ima neka druga pot do tega bloka recimo s skokov, velja $2^s \leq B$, saj nam vsak skok podvoji številko trenutnega bloka, ta pa ne sme preseči B; torej je $s \leq \lg B$, ker pa je s celo število, pomeni to tudi $s \leq \lfloor \lg B \rfloor$.

Ni pa nujno, da je vsaka rešitev s tem maksimalnim številom skokov že tudi najkrajša — skoke moramo izvesti ob pravem času. Na primer, najkrajša pot do 15 je $1 \Rightarrow 2 \rightarrow 3 \Rightarrow 6 \rightarrow 7 \Rightarrow 14 \rightarrow 15$ (puščice $\rightarrow$ ponazarjajo korake na naslednji blok, $\Rightarrow$ pa skoke na dvakratno številko) in ima tri skoke (ali dva, odvisno od tega, kako štejemo korak od 1 do 2). Toda poti s tremi skoki je še več, niso pa vse najkrajše; $1 \Rightarrow 2 \Rightarrow 4 \Rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow \cdots \rightarrow 15$ ima na primer tudi tri skoke, pa je precej daljša od najkrajše poti.

```
      if Odd(Blok) then
        begin Korak := Korak + 1; Pot[Korak] := Naprej; Blok := Blok − 1 end;
      Korak := Korak + 1; Pot[Korak] := Skok; Blok := Blok div 2;
    end; {while}
    { sedaj pot obrnemo }
    Zadnji := Korak;
    for Korak := 1 to Zadnji div 2 do begin
      t := Pot[Korak];
      Pot[Korak] := Pot[Zadnji − Korak];
      Pot[Zadnji − Korak] := t;
    end; {for}
end; {PotDoBloka}
```

Do bloka s številko 2^n pridemo zelo hitro: v n korakih, pri čemer vedno uporabimo naslov dvakrat bolj oddaljenega bloka. Največ dela imamo, da preberemo blok s številko $2^n - 1$. To vidimo takole: da do tega bloka pridemo iz bloka s številko $2^{n-1} - 1$, porabimo dva koraka. Od začetka datoteke tako opravimo $2n - 2$ korakov, kar je mogoče enostavno dokazati z matematično indukcijo. Podobno dokažemo tudi, da do tega bloka ni krajše poti. Za zapis vseh poti do blokov s številkami, manjšimi ali enakimi 2^n, torej potrebujemo najmanj $2n - 2$ bitov.

Prepričajmo se z indukcijo, da naš postopek res vedno daje najboljšo rešitev. Naj bo $\#_0(B)$ število ničel v dvojiškem zapisu števila B, $\#_1(B)$ pa število enic. Iz glavne zanke **while** našega programa se takoj vidi, da je pot, ki jo ta program najde do bloka B, dolga $f(B) := \#_0(B) + 2(\#_1(B) - 1)$ korakov. Označimo z $g(B)$ dolžino najkrajše poti do B; očitno je $g(1) = 0$ (da pridemo do bloka 1, nam ni treba narediti ničesar, ker že vnaprej vemo, kje se nahaja), $g(2B + 1) = g(2B) + 1$ (ker se vse poti do bloka z liho številko $2B + 1$ končajo s korakom od $2B$ do $2B + 1$ in je zato tudi najkrajša pot do $2B + 1$ lahko le podaljšek najkrajše poti do $2B$) in $g(2B) = \min\{g(B), g(2B - 1)\} + 1$ (vse poti do bloka s sodo številko $2B$ se končajo ali s korakom od $2B - 1$ do $2B$ ali pa s skokom od B do $2B$, najkrajša pa bo pač tista med njimi, ki je podaljšek najkrajše poti do $2B - 1$ ali pa do B).

O tem, da vrača naš program vedno najboljšo rešitev, se lahko prepričamo, če pokažemo, da je $g(B) = f(B)$ pri vseh B. Za $B = 1$ je to očitno res, saj je $f(B) = g(B) = 0$. Recimo zdaj, da velja $f(b) = g(b)$ za vse $b = 1, 2, \ldots, B - 1$. Če je B lih, se njegov dvojiški zapis od $B - 1$ razlikuje le v tem, da ima $B - 1$ v najnižjem bitu ničlo, B pa enico; zato je $f(B) = f(B - 1) + 1$; po induktivni predpostavki je $f(B - 1) = g(B - 1)$, po definiciji g pa je $g(B) = g(B - 1) + 1$ (saj je B lih), tako da sledi $f(B) = g(B)$.

Recimo zdaj, da je B sod; v dvojiškem zapisu ima na najnižjih nekaj mestih ničle, prej ali slej pa tudi neko enico: recimo, da ima na najnižjih k bitih ničle, na naslednjem pa enico. Ker je B sod, se števili B in $B/2$ v dvojiškem zapisu razlikujeta le po tem, da ima $B/2$ na koncu eno ničlo manj, tako da je

$f(B/2) = f(B) - 1$ (po definiciji f); število $B-1$ pa se od B razlikuje po tem, da ima v dvojiškem zapisu na spodnjih k mestih enice, na naslednjem pa ničlo (le tako je namreč mogoče, da s prištevanjem 1 dobimo v vsoti na spodnjih k mestih same ničle: drugače bi se prenos pri seštevanju že prej ustavil) in iz definicije f sledi $f(B-1) = f(B) + k - 1$ ($+k$ zato, ker ima $B-1$ enice na spodnjih k mestih, B pa ima tam ničle, -1 pa zato, ker ima $B-1$ na naslednjem mestu ničlo, B pa enico). Izjema je le, če je dvojiški zapis števila $B-1$ za eno mesto krajši od zapisa števila B, torej če je $B-1$ sestavljen iz samih enic ($B - 1 = 2^{k+1} - 1$ je iz k enic); tedaj $B-1$ nima tiste ničle levo od svojih enic in velja $f(B-1) = f(B) + k - 2$. Ker nas zanimajo $B > 1$, je $k \geq 1$, tako da iz $f(B-1) \geq f(B) + k - 2$ sledi $f(B-1) \geq f(B) - 1 = f(B/2)$.

Po induktivni predpostavki je $f(B/2) = g(B/2)$ in $f(B-1) = g(B-1)$ in po definiciji g je $g(B) = \min\{g(B/2), g(B-1)\} + 1$, kar je naprej enako $\min\{f(B/2), f(B-1)\}+1$, to pa je zaradi $f(B-1) \geq f(B/2)$ enako $f(B/2)+1$, za kar smo zgoraj ugotovili, da je enako $f(B)$. Torej velja $f(B) = g(B)$ tudi v tem primeru.

N: 24

R1988.2.2

Spodnji podprogram v zanki preverja istoležne znake modela in niza. Čim odkrijemo kakšno neujemanje med trenutnima črkama, lahko takoj nehamo in že vemo, da se niz ne ujema z modelom. Zapomniti si moramo tudi, če smo v modelu že prišli do zvezdice (spremenljivka VseObv); tako lahko preverimo, če ni bil ukaz preveč okrajšan.

```
const MaxDolz = 20;
type BesedaT = packed array [1..MaxDolz] of char;

function AliJeUkaz(Model, Niz: BesedaT): boolean;
var
  iNiz: integer;      { števec znakov v nizu }
  iModel: integer;    { števec znakov v modelu ključne besede }
  VseObv: boolean;    { ali smo obdelali obvezni del ukaza }
  SeUjema: boolean;   { ali se obdelani del niza ujema z modelom }
  Konec: boolean;     { ali smo prišli do konca niza }
begin
  iNiz := 1; iModel := 1; VseObv := false; SeUjema := true; Konec := false;
  while SeUjema and not Konec do begin
    if Model[iModel] = '*' then
      begin VseObv := true; iModel := iModel + 1 end
    else if Niz[iNiz] = ' ' then Konec := true
    else if Niz[iNiz] <> Model[iModel] then SeUjema := false
    else begin iNiz := iNiz + 1; iModel := iModel + 1 end;
  end; {while}
  AliJeUkaz := VseObv and SeUjema;
end; {AliJeUkaz}
```

Če bi prišli do konca modela prej kot do konca niza, bi podprogram še vedno pravilno deloval, ker bi primerjal trenutni znak niza s presledkom, na katerega bi naletel na koncu modela, in ugotovil, da se pač ne ujemata.

R1988.2.3

N: 24

Če v zanki opazujemo stanje senzorjev (Ozn1, Ozn2) in ga primerjamo s stanjem ob prejšnji meritvi (PrejOzn1, PrejOzn2), lahko opazimo, kdaj smo prišli do začetka naslednje oznake. Če si zapomnimo še čas, ko smo opazili začetek prejšnje oznake (t1, t2), lahko izračunamo hitrost vrtenja: ena oznaka v dt1 oz. dt2 milisekundah. S primerjavo teh dveh hitrosti lahko usmerjamo pogon (v nasprotno smer): če je $1/\mathsf{dt1} > 1/\mathsf{dt2}$, moramo vrtenje prve osi malo upočasniti, sicer pa pospešiti. Ker naloga pravi, da zmore motor le pospeške med -1 in 1, delimo razliko $1/\mathsf{dt1} - 1/\mathsf{dt2}$ s hitrostjo druge osi, tako da pospeški ali pojemki, ki jih bomo zahtevali, ne bodo preveliki (razen če se ne vrti prva os res zelo hitro). Ko na plošči prvič zagledamo začetek oznake, hitrosti še ne moremo oceniti, ker ne vemo, kdaj se je začela prejšnja oznaka; ta primer si zapomnimo s spremenljivkama Uvod1 in Uvod2.

```
program Sklopka;
type
  CasT = integer;
  OsT = 1..2;

  function Cas: CasT; external;
  function Oznaka(Os: OsT): boolean; external;
  procedure Pogon(Pospesek: real); external;
  procedure Sklopi; external;

var
  Uvod1, Uvod2: integer;        { koraki, potrebni za začetek meritve }
  Ozn1, Ozn2, OznPrej1, OznPrej2: boolean;    { za iskanje začetka oznake }
  t, t1, t2: CasT;              { čas zadnje oznake }
  dt1, dt2: CasT;               { čas med dvema oznakama }
  RelNapaka: real;              { relativna razlika hitrosti osi }
  Izenaceno: boolean;           { je razlika hitrosti osi dovolj majhna? }
begin {Sklopka}
  Uvod1 := 2; Uvod2 := 2; Izenaceno := false;
  OznPrej1 := Oznaka(1); OznPrej2 := Oznaka(2); t1 := 0; t2 := 0;
  repeat
    Ozn1 := Oznaka(1); Ozn2 := Oznaka(2); t := Cas;
    if Ozn1 and not OznPrej1 then begin
      dt1 := t - t1; t1 := t; if Uvod1 > 0 then Uvod1 := Uvod1 - 1;
    end; {if}
    if Ozn2 and not OznPrej2 then begin
      dt2 := t - t2; t2 := t; if Uvod2 > 0 then Uvod2 := Uvod2 - 1;
    end; {if}
    OznPrej1 := Ozn1; OznPrej2 := Ozn2;
```

```
      if (Uvod1 = 0) and (Uvod2 = 0) then begin
        RelNapaka := (1/dt1 − 1/dt2) / (1/dt2); Pogon(−RelNapaka);
        Izenaceno := Abs(RelNapaka) < 0.01;
      end; {if}
    until Izenaceno;
    Sklopi;
end. {Sklopka}
```

Naloga pravi, da funkcija Cas šteje milisekunde; če se plošči vrtita dovolj hitro in je na njiju dovolj veliko število belih in črnih oznak, zna biti takšno merjenje časa premalo natančno. Pri n oznakah vsake barve in k obratih na sekundo bi morala naša spremenljivka dt izmeriti $1000/(nk)$ milisekund, v resnici pa bo namerila $\lfloor 1000/(nk) \rfloor$ ali pa $\lceil 1000/(nk) \rceil$. Če torej namestimo d milisekund, utegne biti prava vrednost karkoli med $d-1$ in $d+1$, pravo število obratov n pa zato karkoli med $(d-1)n/1000$ in $(d+1)n/1000$. Krajše ko so oznake (večji n) in hitreje ko se vrti opazovana os (manjši d), bolj je ta naša meritev nenatančna, zato pa je tudi razlika v hitrosti osi, ko ju na koncu poskušamo sklopiti, lahko večja. Natančnejše meritve bi lahko dobili, če bi merili čas v manjših enotah (s tem bi v zadnjih dveh formulah namesto 1000 dobili nek večji imenovalec) ali pa bi namesto časa od začetka ene do začetka naslednje črne oznake gledali po več oznak zaporedoma (kar ima tak učinek, kot če bi se zmanjšal n); pri slednjem moramo seveda upati, da se hitrost vrtenja ne spreminja prehitro, da bi to preveč pokvarilo naše meritve.

Krmiljenje pogona v tej rešitvi je tudi sicer le zasilno; potreben čas za izenačitev hitrosti ni najmanjši. Za optimalno krmiljenje bi bilo potrebno poznavanje odziva mehanizma in upoštevanje metod krmiljenja procesov s povratnimi zankami.

N: 25

R1988.2.4

Program v notranji zanki **for** določi obrat obr štirimestnega števila stv; obrat 1234 je 4321, obrat 1 je 1000. V zunanji zanki zato uravnoteženo računa razliko naslednjih dveh vsot:

$$\begin{aligned} S_1 &= 2+3+\ldots+9999 \\ &= (1+2+\ldots+9999)-1 \\ &= S-1 \end{aligned}$$

$$\begin{aligned} \text{in } S_2 &= o(2)+o(3)+\ldots+o(9999) \\ &= (o(1)+o(2)+\ldots+o(9999))-o(1) \\ &= S-1000. \end{aligned}$$

Tu smo z $o(n)$ označili obrat štirimestnega števila n. V vsoti S_1 nastopajo vsa štirimestna števila razen 1 (med štirimestna števila štejemo vsa števila, ki jih lahko zapišemo z največ štirimi števkami), v drugi vsoti pa, čeprav v

premešanem vrstnem redu, ravno tako vsa štirimestna števila, razen obrata 1, ki je enak 1000. Program izpiše razliko obeh vsot, to je

$$S_1 - S_2 = (S - 1) - (S - 1000) = 999.$$

Če bi vrednost pStv spremenili v 1, bi program izračunal razliko $S - S$, zato bi izpisal 0.

REŠITVE NALOG ZA TRETJO SKUPINO

R1988.3.1

N: 26

Eden od najhitrejših postopkov za iskanje praštevil na enoprocesorskem sistemu je Eratostenovo sito. Vzemimo ga tudi za osnovo postopka na večprocesorskem sistemu. Da bo postopek učinkovit, mora biti delo enakomerno porazdeljeno med vse procesorje (vsak bo rešetal svoj približno enako velik del tabele števil).

Ob zagonu vsi procesorji opravijo inicializacijo: kot prvo praštevilo izberejo 2. Vsi procesorji potem delajo po istem postopku: vsak si najprej določi del sita (tabele), v katerem rešeta in iz svojega dela tabele odstrani vse mnogokratnike trenutnega praštevila. Nato vsak procesor v svojem delu sita poišče najmanjše še ne odstranjeno število. To število je kandidat za novo praštevilo. Vsak procesor nato počaka, da tudi ostali izberejo vsak svojega kandidata; kot novo praštevilo vsi opisani procesorji izberejo najmanjšega med kandidati. Nato vsi ponove opisani postopek.

Ker je v vsakem delu tabele približno enako mnogo mnogokratnikov vsakokratnega praštevila, vsi procesorji končajo z delom v približno enakem času. Zato lahko ocenimo, da z vzporednim postopkom praštevila določimo približno desetkrat hitreje kot z navadnim. Nekaj časa več je potrebnega le zaradi delitve dela.

```
program VzporednoSejanje;
const
  MaxStev = 100000;          { Gornja meja območja iskanih praštevil. }
  StProc = 10;               { Število procesorjev. }
var
  { Skupne spremenljivke. V tabeli Kandidat vsak procesor predlaga,
    s katerim praštevilom bi se v nadaljevanju ukvarjali (obvelja najmanjše). }
  Kandidat: shared array [1..StProc] of integer;
  JePrast: shared packed array [1..MaxStev] of boolean;    { Sito. }

  { Lokalne spremenljivke. }
  Stev: integer;             { Trenutno število. }
  Prast: integer;            { Trenutno praštevilo, čigar večkratnike črtamo. }
  SpMeja, ZgMeja: integer;   { Meji obdelovanega dela sita. }
  Oznaka: 0..StProc − 1;     { Oznaka procesorja }
```

```
  procedure MojaOznaka(var Oznaka: integer); external;
  procedure Zaseden; external;
  procedure Prost; external;
  function VsiProsti: boolean; external;
  function VsiZasedeni: boolean; external;

begin {VzporednoSejanje}
  { Inicializacija. }
  MojaOznaka(Oznaka); Zaseden; Prast := 2;
  { Vsak procesor počisti svoj del sita. }
  for Stev := 1 + MaxStev * Oznaka div StProc to
              MaxStev * (Oznaka + 1) div StProc do JePrast[Stev] := true;
  { Počakajmo na ostale. }
  repeat until VsiZasedeni; Prost; repeat until VsiProsti;
  { Začnimo z rešetanjem. }
  while Prast <= MaxStev do begin
    Zaseden;
    { Pregledali bomo števila od Prast do MaxStev in prečrtali večkratnike
      števila Prast. Razdelimo si ta kos tabele na StProc delov; naš
      procesor bo pregledal indekse SpMeja..ZgMeja. }
    SpMeja := Prast + (MaxStev - Prast + 1) * Oznaka div StProc;
    ZgMeja := Prast + ((MaxStev - Prast + 1) * (Oznaka + 1) div StProc) - 1;
    { Poiščimo najmanjši večkratnik števila Prast v svojem delu tabele. }
    Stev := ((SpMeja + Prast - 1) div Prast) * Prast;
    if Stev <= Prast then Stev := Stev + Prast;
    { Presejmo svoj del tabele. }
    while Stev <= ZgMeja do
      begin JePrast[Stev] := false; Stev := Stev + Prast end;
    { Poiščimo kandidata za naslednje praštevilo. }
    Stev := SpMeja; if Stev <= Prast then Stev := Prast + 1;
    if Stev <= SpMeja then
      while (Stev < ZgMeja) and not JePrast[Stev] do Stev := Stev + 1;
    if Stev >= SpMeja then Kandidat[Oznaka] := MaxStev + 1
    else Kandidat[Oznaka] := Stev;
    { Počakajmo na ostale. }
    repeat until VsiZasedeni; Prost; repeat until VsiProsti;
    { Določimo naslednje praštevilo. }
    Zaseden;
    Prast := MaxStev + 1;
    for Stev := 0 to StProc - 1 do
      if Kandidat[Stev] < Prast then Prast := Kandidat[Stev];
    { Počakajmo na ostale. }
    repeat until VsiZasedeni; Prost; repeat until VsiProsti;
  end; {while}
```

end. {*VzporednoSejanje*}

Tehnika, ki jo gornji program uporablja za usklajevanje (sinhronizacijo) dela posameznih procesorjev, se imenuje „sinhronizacija z oviro“ (*barrier synchronization*): ko pride nek procesor do ovire, mora počakati, dokler ne pridejo do nje še vsi ostali.[6] Preden označi procesor sebe za prostega, mora počakati, da so vsaj za hip vsi označeni kot zasedeni — to je znak, da so že zapustili prejšnjo oviro. Zato je dobro, da imamo na razpolago podprogram VsiZasedeni. Brez tega bi se lahko zgodilo naslednje: ko procesorji čakajo na oviri in jo doseže še zadnji med njimi, bi zdaj eden pač prvi opazil, da so vsi prosti, in bi oddrvel naprej; in zdaj bi se lahko zgodilo, da bi se ta razglasil za zasedenega, še preden bi drugi uspeli opaziti, da so bili nekaj časa vsi hkrati prosti. Zato bi drugi še naprej čakali na prejšnji oviri, medtem pa bi naš procesor tekel naprej in se ustavil šele na naslednji oviri; ko bi se takrat razglasil za prostega, bi se lahko ponovila ista težava. Spet bi lahko eden od procesorjev oddrvel naprej, mogoče celo isti kot prej. Naš program bi po vsem tem lahko dajal čisto napačne rezultate.

Razmislimo še o tem, kako bi lahko naredili oviro, če ne bi imeli funkcij, kot sta VsiZasedeni in VsiProsti. Če bi imeli na voljo kakšen zanesljiv mehanizem za zaklepanje (na primer inštrukcijo, ki prebere in nato zapiše neko pomnilniško celico in pri tem zagotavlja, da je med branjem in pisanjem ne bo prekinil noben drug procesor — spodaj tako operacijo predstavlja funkcija BeriInPisi), bi lahko oviro izvedli s števcem. Prvotna vrednost števca naj bo 0; vsak procesor, ki pride do ovire, mora števec povečati za 1, nato pa v zanki opazuje vrednost števca in ko ta doseže StProc, zapusti oviro. Z zaklepanjem lahko zagotovimo, da pri povečevanju števca procesorji ne bodo hodili drug drugemu v zelje.

```
  var Kljucavnica: shared integer value 0; { za dostop do števca }
      Stevec: shared integer value 0;  { koliko jih čaka na trenutni oviri }
      TrenutnaOvira: integer value 0; { ovira, ki jo je naš procesor nazadnje zapustil }
      KoncanaOvira: shared integer value 0; { ovira, ki se je nazadnje sprostila
                                              (ker so jo dosegli vsi procesorji) }

  { Vpiše N v S in vrne staro vrednost S-ja. To opravi kot atomarno (nedeljivo)
    operacijo, torej med našim branjem in pisanjem ne more nihče drug do S. }
  function BeriInPisi(var S: integer; N: integer): integer; external;

procedure Ovira;
var St: integer;
begin
  TrenutnaOvira := 1 - TrenutnaOvira;        { Gremo na naslednjo oviro. }
  repeat until BeriInPisi(Kljucavnica, 1) = 0;  { Zasezimo števec. }
  St := Stevec + 1; Stevec := St;   { Povečajmo števec procesorjev na tej oviri. }
```

[6] Gl. npr. J. L. Hennessy, D. A. Patterson: *Computer Architecture: A Quantitative Approach*, 2. izd., 1996, str. 700–703.

```
  Kljucavnica := 0;                          { Sprostimo števec. }
  if St = StProc then    { Mi smo zadnji na tej oviri — dajmo znak za konec. }
    begin St := 0; KoncanaOvira := TrenutnaOvira end
  else    { Počakajmo, da pridejo še drugi do ovire in jo eden od njih sprosti. }
    repeat until KoncanaOvira = TrenutnaOvira;
end; {Ovira}
```

Ker naša naloga ne omenja, da bi imeli na voljo zaklepanje, lahko oviro izvedemo takole:

```
var NaOviri: shared array [1..StProc] of integer;

procedure Ovira;
var i, TaOvira, NaslednjaOvira: integer; Ok: boolean;
begin
  TaOvira := (NaOviri[Oznaka] + 1) mod 3;
  NaOviri[Oznaka] := TaOvira;
  NaslednjaOvira := (TaOvira + 1) mod 3;
  repeat
    { Preverimo, če so že vsi prišli do trenutne ovire. }
    i := 0; Ok := true;
    while Ok and (i <= StProc) do begin
      Ok := (NaOviri[i] = TaOvira) or (NaOviri[i] = NaslednjaOvira);
      i := i + 1;
    end; {while}
  until Ok;
end; {Ovira}
```

Vrednost NaOviri[i] torej pove, katera je zadnja ovira, ki jo je procesor i že dosegel (in mogoče tudi zapustil). Na oviri moramo čakati, dokler ne ugotovimo, da so vsi že na trenutni ali pa na naslednji oviri. Možnost, da je kakšen procesor A že na naslednji oviri, se lahko zgodi le v primeru, da je A že prej opazil, da so vsi na trenutni oviri, in oddrvel naprej do naslednje ovire, še preden smo mi sploh ponovno preverili, ali so vsi na trenutni oviri ali ne. Pomembno pa je, da A ne bo mogel oddrveti dlje kot do naslednje ovire, saj on zdaj tam čaka, da bodo tudi vsi ostali prišli vsaj do tiste ovire — dokler je kdo še na trenutni oviri, se A ne more premakniti naprej. Zato tudi štejemo ovire s števili 0, 1, 2 in ne le 0, 1: v slednjem primeru ne bi mogli ločiti procesorjev, ki čakajo še na prejšnji oviri, od tistih, ki čakajo že na naslednji.

Priznati je treba, da bi bil podprogram Ovira verjetno razmeroma neučinkovit: ko nek procesor spremeni svojo celico NaOviri[Oznaka], jo bodo tisti, ki trenutno še čakajo v zanki **repeat**, kmalu spet prebrali; ker se je spremenila, jo bodo morali na novo potegniti iz skupnega pomnilnika. To bi utegnilo povzročiti veliko konfliktov na vodilu.

Pri inicializaciji moramo zagotoviti, da dobijo vsi elementi tabele NaOviri vrednosti 0, še preden bo kakšen od procesorjev prvič poklical podprogram

Ovira. V nasprotnem primeru bi se namreč lahko zgodilo, da bi Ovira že poskušala postaviti nek element tabele NaOviri na neko novo vrednost, ki bi označevala, da je ta procesor prišel do te ovire, nato pa bi nek drug procesor v okviru inicializacije povozil celo tabelo z ničlami. To, da bi vsak procesor postavil na nič le svojo celico tabele NaOviri, ni dovolj, ker potem pri prvem klicu podprograma Ovira ne vemo, katere celice že vsebujejo smiselne vrednosti, katere pa še ne. S podprogramom VsiProsti si ne moremo pomagati, kajti tudi če se med inicializacijo vsi procesorji razglašajo za zasedene, to, da so vsi prosti, še ne pomeni, da so res že opravili z inicializacijo — mogoče je sploh še niso začeli. Res zanesljiva rešitev bi bila, da bi nek kontrolni program postavil tabelo NaOviri na 0, še preden se naši paralelni programi sploh začnejo izvajati.

Mimogrede, Eratostenovo sito lahko z nekaj preprostimi prijemi še pospešimo, tako v enoprocesorski kot v tu opisani paralelni različici. Na primer: prva stvar, ki jo bo program naredil, je to, da bo prečrtal v situ vsa soda števila (razen 2); torej lahko 2 obravnavamo kot poseben primer, v situ pa hranimo le liha števila; tako prihranimo pol pomnilnika. Ko brišemo iz rešeta večkratnike praštevila p, nima smisla začeti pri p ali $2p$. Vsi p-jevi večkratniki do vključno $(p-1)p$ imajo očitno tudi nek faktor, manjši od p; in ker obravnavamo praštevila v naraščajočem vrstnem redu, smo jih morali prečrtati že prej. Torej lahko začnemo črtati p-jeve večkratnike šele od p^2 naprej, postopek pa lahko ustavimo, čim velja $p^2 >$ MaxStev (saj odtlej ne bi prečrtali ničesar več). Poleg tega tudi ni treba gledati vseh p-jevih večkratnikov: eno od števil kp in $(k+1)p$ je gotovo sodo in ga torej v tabeli ni; tako je dovolj, če pri praštevilu p gledamo večkratnike p^2, $(p+2)p$, $(p+4)p$ in tako naprej.

Možne so tudi bolj zapletene izboljšave Eratostenovega sita; glej opombo na koncu rešitve naloge 2001.1.4 (str. 454).

Tudi na `online-judge.uva.es` je nekaj nalog, kjer pride prav Eratostenovo sito (npr. #367, #10311).

R1988.3.2

N: 26

Nalogo najlaže rešimo z rekurzivnim sestopom; v nizu poiščemo prvi znak, ki se ujema s prvim znakom podniza, nato pa iskanje ponovimo s preostankom podniza v preostanku niza. Če pri tem sestopu podniz izpraznimo, smo našli pojavitev; če ne najdemo poti naprej, pa se vrnemo na prejšnji nivo. Ta metoda je eden od osnovnih načinov za reševanje problemov v umetni inteligenci.[7]

```
program IsciPodniz(Input, Output); { Ugotovi, kolikokrat podniz nastopa v nizu. }

  const MaxDolz = 20;
  type NizT = packed array [1..MaxDolz] of char;
```

[7] Obstajajo pa tudi precej učinkovitejše rešitve od tu opisane, npr. z dinamičnim programiranjem; gl. nalogo 1997.2.3 (str. 295, rešitev na str. 311), ki je praktično povsem enaka tej nalogi.

```
  var Niz, Podniz: NizT;
      Stevilo: integer;    { število pojavitev niza v podnizu }

  { Primerja del niza od ZacNiz dalje s podnizom od ZacPodniz dalje. }
  procedure Primerjaj(ZacNiz, ZacPodniz: integer; var Stevilo: integer);
  begin
    if (ZacNiz > MaxDolz) or (ZacPodniz > MaxDolz) then begin end
    else if Podniz[ZacPodniz] = ' ' then Stevilo := Stevilo + 1
    else begin
      while ZacNiz <= MaxDolz do begin
        if Niz[ZacNiz] = Podniz[ZacPodniz] then
          Primerjaj(ZacNiz + 1, ZacPodniz + 1, Stevilo);
        ZacNiz := ZacNiz + 1;
      end; {while}
    end; {if}
  end; {Primerjaj}

begin
  Write('Vnesi niz: '); ReadLn(Niz);
  Write('Vnesi podniz: '); ReadLn(Podniz);
  Stevilo := 0; Primerjaj(1, 1, Stevilo);
  WriteLn;
  WriteLn('Podniz "', Podniz, '" se ', Stevilo, '-krat ',
          'pojavi v nizu "', Niz, '".');
end. {IsciPodniz}
```

Če bi na začetku izmerili dolžino niza in podniza, bi lahko pogoj v zanki **while** zamenjali z malo strožjim ZacNiz <= DolzinaNiza − DolzinaPodniza + ZacPodniz. S tem se izognemo brezplodnemu pregledovanju primerov, ko je v nizu ostalo manj znakov kot v podnizu. Vendar je postopek kljub temu še vedno zelo neučinkovit.

N: 27

R1988.3.3

Spodnji program hrani v tabeli Vrsta zadnjih 40 vzorcev. Toliko jih potrebujemo zaradi možnosti, da pride do daljše skupine napačnih vzorcev: ko opazimo, da je trenutni vzorec in še prejšnjih devet napačnih, moramo trideset pravilnih pred njimi utišati, tako da potrebujemo v vrsti vsaj zadnjih 40 vzorcev. Z drugimi besedami, šele ko pride za nekim vzorcem še 40 drugih, smo lahko prepričani, da že poznamo dokončno vrednost tistega vzorca. V spremenljivki ZadnjiPrav si bomo zapomnili, kateri je zadnji pravilni vzorec pred trenutnim; tako lahko opazimo, če pride do krajše napake in moramo vmesne vzorce interpolirati, opazimo pa lahko tudi, kdaj dobimo deseto zaporedno napako in moramo vzorce pred tem utišati. V slednjem primeru tudi postavimo Utisaj na 30; to vrednost ohrani, dokler opažamo napačne vzorce, nato pa jo začnemo zmanjševati in s tem nadziramo postopno linearno ojačanje zvoka. Naloga ne predpisuje, kaj storiti v primeru,

če med dvema dolgima zaporedjema napak nastopi peščica (59 ali manj) dobrih vzorcev: pri nekem konkretnem dobrem vzorcu lahko zahteva prejšnja skupina napak oslabitev s faktorjem $a/31$, naslednja skupina napak pa z $b/31$. Ena možnost bi bila, da bi ga pomnožili kar z $\min\{a,b\}/31$, naš spodnji program pa ga v takem primeru oslabi z vsakim faktorjem posebej, kar ustreza množenju vzorca z $(a/31)(b/31)$.

```
program CD;

  const m = 40;                { dolžina zakasnilne vrste }
  type VzorecT = 0..65535;     { vzorčena amplituda signala }

  procedure PosljiVzorec(Vzorec: VzorecT); external;
  procedure DobiVzorec(var Vzorec: VzorecT; var Pravilen: boolean); external;

var
  Vrsta: array [1..m] of VzorecT;   { zakasnilna vrsta }
  Pravilen: boolean;                { ali je vzorec pravilen }
  ZadnjiPrav: integer;              { indeks zadnjega dobrega vzorca v vrsti }
  Utisaj: integer;                  { utišati je treba toliko naslednjih vzorcev }
  i: integer;
  k: real;
begin
  ZadnjiPrav := m; Utisaj := 0;
  for i := 1 to m do Vrsta[i] := 0;
  repeat
    PosljiVzorec(Vrsta[1]);                      { pošlji najstarejši vzorec }
    for i := 2 to m do Vrsta[i - 1] := Vrsta[i]; { pomakni vrsto }
    DobiVzorec(Vrsta[m], Pravilen);
    if ZadnjiPrav > 1 then ZadnjiPrav := ZadnjiPrav - 1;
    if Pravilen then begin
      if Utisaj > 0 then begin
        { zaradi prejšnje daljše napake postopno povečujemo jakost }
        Vrsta[m] := Vrsta[m] * (30 - Utisaj + 1) div 31;
        Utisaj := Utisaj - 1;
      end; {if}
      if m - ZadnjiPrav > 1 then begin           { interpoliraj }
        k := (Vrsta[m] - Vrsta[ZadnjiPrav]) / (m - ZadnjiPrav);
        for i := ZadnjiPrav + 1 to m - 1 do
          Vrsta[i] := Vrsta[ZadnjiPrav] + Round(k * (i - ZadnjiPrav));
      end; {if}
      ZadnjiPrav := m;
    end else if Utisaj = 30 then begin           { daljša napaka še vedno traja }
      Vrsta[m] := 0; ZadnjiPrav := m;
    end else if m - ZadnjiPrav > 9 then begin    { zadnjih 10 vzorcev je napačnih }
      { postopno utišaj 30 vzorcev pred napačnimi }
      for i := ZadnjiPrav - 30 + 1 to ZadnjiPrav do
```

```
          Vrsta[i] := Vrsta[i] * (ZadnjiPrav − i + 1) div 31;
        for i := ZadnjiPrav + 1 to m do Vrsta[i] := 0;
        { treba bo postopno dvigniti jakost 30-tim pravilnim vzorcem }
        Utisaj := 30; ZadnjiPrav := m;
      end; {if}
    until false;
end. {CD}
```

N: 27

R1988.3.4

Postopek kodiranja je naslednji: Aleš najprej zakodira vsebino sporočila s svojo skrivno funkcijo f_A. Potem sporočilu doda svoj podpis (da Bojanu olajša iskanje pošiljatelja) in celotno sporočilo zakodira še z Bojanovo javno funkcijo g_B. Tako zakodirano sporočilo lahko Bojan (in le Bojan) razkodira najprej z uporabo svoje skrivne funkcije f_B. Tako zve za podpisnika sporočila (vendar v avtentičnost še ne more biti prepričan). Preostali del sporočila sedaj razkodira z Aleševo javno funkcijo g_A. Če je sporočilo čitljivo, je s tem preveril, da mu je sporočilo res poslal Aleš.

Zapišimo postopek še drugače (funkcija f je skrivna, g pa javna):

$$x \xrightarrow[f_A]{\text{Aleš}} f_A(x) \xrightarrow[g_B]{\text{Aleš}} g_B(f_A(x)) \xrightarrow[f_B]{\text{Bojan}} f_B(g_B(f_A(x))) \parallel f_A(x) \xrightarrow[g_A]{\text{Bojan}} g_A(f_A(x)) = x.$$

Problem nastopi le tedaj, ko sta naslovnik in pošiljatelj ista oseba, saj tedaj vsebina sporočila potuje nezakodirana. Možna rešitev bi bila, da bi imel vsakdo dva različna para ključev: en par bi se uporabljal, ko je ta človek v vlogi pošiljatelja, drugi par pa, ko je ta človek prejemnik. Prvi par bi bil torej namenjen podpisovanju (ugotavljanju istovetnosti pošiljatelja), drugi pa šifriranju (da sporočila ne more prebrati nihče razen prejemnika).

13. republiško tekmovanje v znanju računalništva (1989)

NALOGE ZA PRVO SKUPINO

1989.1.1

V tabeli velikosti 10×10 znakov nastavi vsem elementom začetne vrednosti `'.'` (pika). Nato v tabelo na naključna mesta shrani števila od 1 do 9 tako, da številke niso v sosednjih elementih. Uporabi deklaracije: R: 48

```
type Tabela = array [0..9, 0..9] of char;
var t: Tabela;
function Random(Obmocje: integer): integer; external;
```

Funkcija Random vrne naključne vrednosti v mejah $0 \leq x <$ Obmocje.

1989.1.2

Program naj prebere sto celih in sto realnih števil. Za tem naj prebere število stolpcev, v katerih naj nato ta števila izpiše urejena po velikosti od najmanjšega do največjega. (Izpisuje naj jih po vrsticah in v vsaki vrstici od leve proti desni, ne najprej po stolpcih in pri vsakem stolpcu od zgoraj navzdol.) Števila, ki jih je prebral kot realna, naj izpiše na dve decimalki natančno. Če se neko število pojavi večkrat, naj ga izpiše le enkrat; če se pojavi tako med celimi kot med realnimi, naj ga izpiše kot celo število (torej brez decimalne vejice in ničel za njo). R: 49

Primer (s tremi celimi in štirimi realnimi števili):

```
  10   5   10   -15.5   -10.4   5.0   20.6
```

izpiše v treh stolpcih

```
-15.50  -10.40        5
    10   20.60
```

1989.1.3

Na voljo imaš niz N^2 znakov ter matriko znakov $N \times N$. **Napiši program**, ki znake iz niza prepiše v matriko tako, da bodo tvorili spiralo z začetkom v gornjem levem kotu. Spirala mora teči v smeri urinega kazalca. R: 52

Primer: `qwertyuiopasdfgh` prepiši v

```
q w e r
s d f t
a h g y
p o i u
```

R: 53 **1989.1.4** **Napiši proceduro**, ki bo z besedami izpisala števila od vključno 0 do 999.

NALOGE ZA DRUGO SKUPINO

R: 54 **1989.2.1** **Napiši proceduro**, ki iz datoteke prebere največ deset trojic ⟨*ime, rezultat, nivo*⟩. Trojice so urejene po padajočem vrstnem redu, ključ urejanja je *rezultat*. V zaporedje na pravo mesto dodaj novo trojico, podano kot argument procedure. Novo zaporedje zapiši nazaj v isto datoteko. Če vsebuje novo zaporedje več kot deset trojic, zapiši le prvih deset. Uporabi naslednje deklaracije:

```
type Top = record
             Ime: array [1..20] of char;
             Rezultat, Nivo: integer;
           end;
var f: file of Top;
```

R: 55 **1989.2.2** Podana je dovolj velika količina šifriranega besedila. Za šifriranje je bila uporabljena metoda, pri kateri vsako črko zamenjamo z istoležno črko iz ključa. Ključ vsebuje vse črke abecede, vsako natanko enkrat. Primer šifriranja:

Besedilo:	`to besedilo bo sifrirano s spodnjim kljucem`
Ključ:	`abcdefghijklmnopqrstuvwxyz` `qazwsxedcrfvtgbyhnujmikolp`
Šifrirano besedilo:	`jb asuswcvb ab ucxncnqgb u uybwgrct fvrmzst`

Napiši proceduro Desifriraj, ki bo brez poznavanja šifrirnega ključa po najboljših močeh dešifrirala to besedilo. Šifrirano besedilo vsebuje samo male črke brez ostalih znakov! Besedilo je napisano v angleščini, katere abeceda vsebuje 26 črk!

```
type Ver = array [1..26] of record
             c: char;      { črka }
             v: real;      { pogostost črke v angleškem besedilu, }
           end;                              { podana v odstotkih }
procedure Desifriraj(var f: text;    { datoteka s šifriranim besedilom }
                         v: Ver);    { tabela pogostosti črk }
```

Tabela Ver vsebuje podatke o pogostosti črk v tipičnem angleškem besedilu. Črke so urejene po padajoči pogostosti, pogostosti pa so izražene v odstotkih (od 0 do 100). Dešifrirano besedilo izpisuj na zaslon.

1989.2.3

Napiši podprogram, ki dano vrednost, zapisano v danem številskem sistemu, pretvori v drug številski sistem. Najvišja številska osnova je 36; za števke uporabljamo poleg običajnih števk `0`, ..., `9` še velike črke angleške abecede, torej `A`, ..., `Z`. Tvoj podprogram naj ustreza naslednjim deklaracijam: R: 59

```
const MaxDolzina = ...;
type Niz = packed array [1..MaxDolzina] of char;
procedure Pretvori(var Stevilo: Niz; IzOsnove, VOsnovo: integer; var Dolzina: integer);
```

Ob klicu bo tvoj podprogram dobil število v tabeli Stevilo; dolgo je Dolzina števk in zapisano v številskem sistemu z osnovo IzOsnove. Ob vrnitvi iz podprograma naj bo v spremenljivki Stevilo isto število, zapisano v številskem sistemu z osnovo VOsnovo, v spremenljivki Dolzina pa število njegovih števk v tem novem zapisu.

1989.2.4

V tabeli dimenzij $N \times M$ naredi labirint brez vhoda in izhoda. Številka 0 v tabeli predstavlja zid, enica pa pot. Poti so lahko široke en element, labirint pa mora biti enostaven, poln[8] in acikličen (brez krožnih poti). Zato sta dimenziji N in M lihi števili. R: 60

NALOGE ZA TRETJO SKUPINO

1989.3.1

Napiši proceduro Isci, ki bo po datoteki iskala zaporedja zlogov (bytov). Iskana zaporedja (rečemo jim tudi „vzorci“) so lahko dolga največ 100 zlogov in jih je lahko največ 20. Procedura mora najti vsa zaporedja, vsebovana v datoteki, in izpisati, katero zaporedje se začne na najdenem mestu. Primer izpisa: R: 67

```
zaporedje 3 odmik 45312
zaporedje 5 odmik 51200
. . . . . .
```

```
type Vzorci = array [1..20] of record
                Zlog: array [1..100] of byte;      { iskani vzorec }
                Dolzina: integer;                  { dolžina vzorca }
```

[8] Takšno besedilo naloge se nam je ohranilo v biltenu s tekmovanja leta 1989. Ni čisto očitno, kaj je mišljeno z „enostaven“ in „poln“. Verjetno „enostaven“ pomeni, da se mora dati priti (po samih enicah) od vsake enice do vsake druge enice — z drugimi besedami, labirint ni sestavljen iz več nepovezanih delov. „Poln“ pa verjetno pomeni, da nočemo imeti kakšnih debelih zidov, npr. kvadrat 2×2 samih ničel, ampak mora biti labirint maksimalno prepreden s potmi. — Nerodno pri tej nalogi pa je tudi to, da je težko natančno definirati, kaj točno je labirint. Konec koncev bi lahko naredili eno samo dolgo kačasto pot, ki bi se vila cikcak gor in dol po celi tabeli, kar bi brez dvoma ustrezalo vsem zahtevam te naloge, obenem pa bi se vsakdo strinjal, da je to presneto klavrn labirint.

```
        end;
{ Podprogram naj v datoteki f poišče pojavitve vzorcev v[1], ..., v[n]. }
procedure Isci(var f: file of byte; v: Vzorci; n: integer);
```

R: 68

1989.3.2

V tabeli $N \times M$ so zapisane nepravilne površine. **Napiši proceduro**, ki bo označila notranji rob ene površine. Iskana površina je določena s parametroma x in y, ki označujeta koordinati začetka površine (najbolj leva gornja točka). Površine so označene s pozitivnimi vrednostmi, praznine pa z nič; robove površin naj tvoj podprogram označi z -1.

```
var Tabela: array [1..N, 1..M] of integer;
procedure Obrobi(x, y: integer);
```

R: 69

1989.3.3

Napiši program, ki prebere število točk v ravnini. Za tem prebere koordinate vseh točk in izpiše najmanjše število premic, ki jih potrebujemo, da vsako izmed danih točk pokrijemo z vsaj eno premico. Pri izračunu zaradi realnih števil upoštevamo točko kot krog s polmerom 10^{-4}.

R: 75

1989.3.4

Napiši funkcijo Eval, ki izračuna vrednost podanega aritmetičnega izraza. Izraz je zapisan v datoteki f kot zaporedje znakov, končano z znakom za konec vrstice (konstanta EOL). Izraz lahko vsebuje realne konstante, operatorje `*`, `/`, `+`, `-`, unarni minus (negativni predznak) ter oklepaje. Funkcija kot svojo vrednost vrne vrednost izraza. Realna števila so lahko zapisana v decimalni obliki (`13.4`, `.6`, `12`, `12.`).

Primer:

```
14*(0.5+.01)      --5
```

vrne vrednost

```
12.14
```

```
function Eval(var f: text): real;
```

REŠITVE NALOG ZA PRVO SKUPINO

N: 45

R1989.1.1

Števke lahko vpisujemo v tabelo eno za drugo; pri vsaki si naključno izberemo koordinati (x, y) celice, kamor jo bomo vpisali. Nato moramo še preveriti, če v tej ali sosednjih celicah (x1, y1) res ni še nobenega števila. Pri tem moramo paziti na možnost, da smo si izbrali celico na robu tabele in zato sosed v kakšnih smereh sploh nima. Bilo bi prikladno, če bi lahko tip Tabela deklarirali kot **array** [-1..10, -1..10] **of** char in v prvo ter zadnjo vrstico ter stolpec postavili pike, tako da bi tiste celice

delovale kot neke vrste stražarji. Ker pa naloga že določa, da naj bo Tabela le tabela 10×10, bomo morali pač posebej preverjati koordinate sosednjih celic, preden jih bomo poskusili pogledati v tabeli.

Po vsakem vpisu neke števke v tabelo postane največ devet celic neprimernih za vpis nadaljnjih števk. Ker imamo sto celic in devet števk, nam torej ni treba skrbeti, da bi v tabeli zmanjkalo primernih položajev, še preden bi nam uspelo vpisati vse števke.

```
program StevilkeVTabeli(Input, Output);
var Tabela: array [0..9, 0..9] of char;
    Stevka: char;
    x, y, x1, y1: integer;
    OK: boolean;
begin
  for x := 0 to 9 do
    for y := 0 to 9 do
      Tabela[x, y] := '.';
  for Stevka := '1' to '9' do
    repeat
      { Naključno izberimo nek položaj v tabeli. }
      x := Random(10); y := Random(10);
      { Preverimo, če so ta celica in njene sosede proste. }
      OK := true;
      for x1 := x - 1 to x + 1 do
        for y1 := y - 1 to y + 1 do
          if (x1 >= 0) and (x1 <= 9) and
            (y1 >= 0) and (y1 <= 9) then
            if Tabela[x1, y1] <> '.' then OK := false;
      { Če je vse v redu, vpišimo trenutno števko v to celico. }
      if OK then Tabela[x, y] := Stevka;
    until OK;
  for x := 0 to 9 do begin
    for y := 0 to 9 do Write(Tabela[x, y]:2);
    WriteLn;
  end; {for}
end. {StevilkeVTabeli}
```

R1989.1.2

N: 45

Naloga zahteva za števila, ki smo jih prebrali kot realna, drugačen izpis kot za tista, ki smo jih prebrali kot cela. Lahko bi prebrali vse v tabelo spremenljivk tipa real, jih uredili, odstranili duplikate in nato pred izpisom vsakega števila pogledali, če je celo ali ne; vendar bi se lahko zgodilo, da bi kakšno število, ki je po vrednosti sicer celo, prebrali le v okviru branja realnih (kjer je bilo npr. podano kot `12.0`) in bi ga morali zdaj (če se hočemo res natančno držati navodil naloge) izpisati na dve decimalki, saj je bilo pač prebrano kot realno. Torej bi si morali

pravzaprav že ob branju pri vsakem številu zapomniti, ali smo ga prebrali kot realno ali kot celo, in potem te podatke prenašati naokoli tudi pri urejanju; pri odstranjevanju duplikatov pa paziti na primere, ko se neko število pojavlja tako med celimi kot med realnimi; takšno število moramo obdržati kot celo, ker naloga v takem primeru zahteva, naj ga izpišemo kot celo. Da ne bomo zapletali na ta način, bomo števila raje hranili v dveh tabelah, eni za cela in eni za realna; vsako posebej bomo uredili in odstranili duplikate, nato pa ju pri izpisu „zlivali“ — v vsakem koraku pogledamo, katera od njiju bi nam dala na naslednjem mestu manjše število; tako lahko zagotovimo, da bo izpis kot celota pravilno urejen, pa še v primeru, da se neko število pojavi v obeh, lahko poskrbimo, da ga bomo izpisali kot celo število.

Za urejanje tabele števil obstaja veliko postopkov, ker pa bosta naši dve tabeli majhni, bomo uporabili kar enega od najpreprostejših — urejanje z izbiranjem (*selection sort*). Najprej poiščemo najmanjše število v tabeli in ga postavimo na prvo mesto; nato poiščemo najmanjše med preostalimi števili in ga postavimo na drugo mesto; tako nadaljujemo, dokler ne poiščemo na koncu manjšega od največjih dveh števil in ga postavimo na predzadnje mesto; ostane le še eno število, ki je največje in je že na pravem mestu. Za prestavljanje elementov po tabeli uporabljamo „zamenjave“, torej operacije tipa `t := x[i]; x[i] := x[j]; x[j] := t`, kjer je `t` pomožna spremenljivka. Po teh treh prirejanjih je v celici `x[i]` tista vrednost, ki je bila prej v `x[j]`, in obratno. Med urejanjem lahko tudi izločamo duplikate: če opazimo, da sta dve števili enaki, lahko eno od njiju zavržemo iz tabele (čezenj na primer napišemo tisto, ki je bilo prej v zadnji celici tabele, nato pa zmanjšamo števec elementov za 1). Zaradi izločanja duplikatov se lahko zaporedji števil skrajšata; namesto `StCelih` in `StRealnih` imamo le še `nc` celih in `nr` realnih števil.

Pri izpisu se pomikamo po tabeli celih števil `x` z indeksom `i`, po tabeli realnih števil `y` pa z indeksom `j`. Če ni še nobeden od teh dveh indeksov prišel do konca tabele, primerjamo naslednje celo število, `x[i]`, in naslednje realno, `y[j]`; če je celo manjše ali enako, bomo izpisali tega; če sta enaki, moramo `y[j]` kar preskočiti (takoj povečamo `j`). Če pa smo pri eni od tabel že prišli do konca (`i > nc` ali pa `j > nr`), moramo pač izpisati naslednje število iz druge tabele. Ko pridemo do konca obeh tabel, končamo. Z zastavico `IzpC` si zapomnimo, ali bo treba izpisati celo število ali ne. Števec `StIzp` pa pove, v katerem stolpcu smo; z njegovo pomočjo vemo, kdaj bo treba iti v novo vrstico.

```
program Urejanje(Input, Output);
const StCelih = 100;
      StRealnih = 100;
var x: array [1..StCelih] of integer;
    y: array [1..StRealnih] of real;
    i, j, nc, nr, StStolpcev, StIzp, x1: integer; y1: real;
    IzpC: boolean;
begin
```

```
{ Preberimo podatke. }
for i := 1 to StCelih do
  begin Write(i, '-to celo število je: '); ReadLn(x[i]) end;
for i := 1 to StRealnih do
  begin Write(i, '-to realno število je: '); ReadLn(y[i]) end;
ReadLn(StStolpcev);

{ Uredimo podatke — cele posebej in realne posebej. }
nc := StCelih; i := 1;
while i < nc do begin
  { V x[i] hočemo dobiti najmanjšega od elementov x[i..nc]. }
  j := i + 1;
  while j <= nc do
    { Spotoma še brišimo duplikate: če sta dve števili enaki,
      eno pobrišimo in skrajšajmo zaporedje. }
    if x[j] = x[i] then begin x[j] := x[nc]; nc := nc - 1 end
    else begin
      if x[j] < x[i] then begin x1 := x[j]; x[j] := x[i]; x[i] := x1 end;
      j := j + 1;
    end; {if}
  i := i + 1;
end; {while}
nr := StRealnih; i := 1;
while i < nr do begin
  { V y[i] hočemo dobiti najmanjšega od elementov y[i..nc]. }
  j := i + 1;
  while j <= nr do
    if y[j] = y[i] then begin y[j] := y[nr]; nr := nr - 1 end
    else begin
      if y[j] < y[i] then begin y1 := y[j]; y[j] := y[i]; y[i] := y1 end;
      j := j + 1;
    end; {if}
  i := i + 1;
end; {while}

{ Lotimo se izpisovanja. }
i := 1; j := 1; StIzp := 0;
while (i <= nc) or (j <= nr) do begin
  { Naj izpišemo naslednje celo število ali naslednje realno? }
  IzpC := false;
  if j > nr then IzpC := true { Če so ostala le še cela, bo treba izpisati celo. }
  else if i <= nc then
    { Imamo tako cela kot realna; poglejmo, katero je manjše. }
    if x[i] <= y[j] then begin
      IzpC := true;
      if x[i] = y[j] then j := j + 1;
    end; {if}
```

```
      if StIzp = StStolpcev { če je to potrebno, začnimo novo vrstico }
        then begin WriteLn; StIzp := 1 end
        else StIzp := StIzp + 1;
      if IzpC then { izpis celega števila }
        begin Write(x[i]:8); i := i + 1 end
      else { izpis realnega števila }
        begin Write(y[j]:8:2); j := j + 1 end;
    end; {while}
    WriteLn;
end. {Urejanje}
```

N: 45

R1989.1.3

Spiralo, v katero moramo postaviti črke, si lahko predstavljamo kot sestavljeno iz več kvadratnih okvirjev (z robom, širokim eno celico), ki so vloženi eden v drugega. Trenutni kvadrat pokriva vrstice in stolpce First..Last. Na začetku je First enak 1, Last pa n, nato pa First povečujemo in Last zmanjšujemo. V spremenljivki Smer si zapomnimo, katero stranico trenutnega kvadrata bomo risali v nadaljevanju (0 = zgornjo, 1 = desno, 2 = spodnjo, 3 = levo), v spremenljivki p pa indeks znaka, ki ga bomo naslednjega vpisali v matriko. Za oglišča kvadrata je pravzaprav vseeno, h kateri stranici jih štejemo, važno je le, da vsako oglišče le k eni stranici. Spodnji program šteje zgornji dve oglišči k zgornji, spodnji dve pa k spodnji stranici.

```
program Spirala(Output);
const n = 5;
var Niz: packed array [1..n * n] of char;
    Matrika: array [1..n, 1..n] of char;
    x, y, First, Last, p, Smer: integer;
begin
    Niz := 'ABCDEFGHIJKLMNOPQRSTUVWXY';
    x := 1; y := 1; p := 1;
    Smer := 0; First := 1; Last := n;
    repeat
      case Smer of
        0: for x := First to Last do
           begin Matrika[x, First] := Niz[p]; p := p + 1 end;
        1: for y := First + 1 to Last - 1 do
           begin Matrika[Last, y] := Niz[p]; p := p + 1 end;
        2: for x := Last downto First do
           begin Matrika[x, Last] := Niz[p]; p := p + 1 end;
        3: for y := Last - 1 downto First + 1 do
           begin Matrika[First, y] := Niz[p]; p := p + 1 end;
      end; {case}
      Smer := (Smer + 1) mod 4;
      if Smer = 0 then begin Last := Last - 1; First := First + 1 end;
    until p > n * n;
```

```
  { Izpišimo matriko na zaslon, čeprav naloga tega pravzaprav ne zahteva. }
  for y := 1 to n do begin
    for x := 1 to n do Write(Matrika[x, y]);
    WriteLn;
  end; {for}
end. {Spirala}
```

R1989.1.4

N: 46

Vsako število najprej razbijmo na posamezne števke (S za stotice, D za desetice in E za enice). Enice so ostanek po deljenju z 10; če pa nato vzamemo količnik po deljenju z 10 in še njega delimo z 10, so desetice ostanek po tem drugem deljenju, stotice pa količnik.

Koristno je imeti pri roki imena posameznih števk. To bo opravil program IzpisiStevko, ki mu lahko tudi povemo, ali naj 2 piše kot *dve* (npr. v 2, 102 ali 200) ali kot *dva* (npr. v 12, 20, 32), poleg tega pa zna za števko izpisati še dani niz ZaStevko.

Zapis števila je iz dveh delov: najprej stotice, nato desetice in enice (npr. *sto triindvajset*). Če sta prisotna oba dela, je vmes presledek. Pri izpisu stotic je *sto* poseben primer, ostala imena stotic pa lahko sestavimo iz imena števke D1 in niza `sto`. Pri izpisu desetic in enic je treba posebej obdelati primer, ko desetic sploh ni (izpišemo le enice), in primer, ko je D2 = 1 (takrat izpišemo enice in *najst*; izjemi sta *deset* in *enajst*); drugače pa izpišemo najprej enice, nato *in*, desetice in še *jset* (za 20..29) ali *deset* (za 30..99).

```
procedure IzpisiStevko(Stevka: integer; Dva: boolean; ZaStevko: string);
begin
  case Stevka of
  1: Write('ena');
  2: if Dva then Write('dva') else Write('dve');
  3: Write('tri');
  4: Write('štiri');
  5: Write('pet');
  6: Write('šest');
  7: Write('sedem');
  8: Write('osem');
  9: Write('devet');
  end; {case}
  Write(ZaStevko);
end; {IzpisiStevko}

procedure IzpisiStevilo(Stevilo: integer);
var S, D, E: integer;
begin
  if Stevilo = 0 then
    Write('nič')
  else begin
```

```
    E := Stevilo mod 10;
    Stevilo := Stevilo div 10;
    D := Stevilo mod 10;
    S := Stevilo div 10;
    case S of
      1: Write('sto');
      2..9: IzpisiStevko(S, false, 'sto');
    end; {case}
    if (S > 0) and (D + E > 0) then Write(' ');
    case D of
      0: if E > 0 then IzpisiStevko(E, false, '');
      1: case E of
           0: Write('deset');
           1: Write('enajst');
           2..9: IzpisiStevko(E, true, 'najst');
         end;
      2..9:
         begin
           if E > 0 then IzpisiStevko(E, true, 'in');
           IzpisiStevko(D, true, '');
           if D = 2 then Write('jset') else Write('deset');
         end;
    end; {case}
  end; {if}
  WriteLn;
end; {IzpisiStevilo}

procedure Izpisi1000Stevil;
var Stevilo: integer;
begin
  for Stevilo := 0 to 999 do IzpisiStevilo(Stevilo);
end; {Izpisi1000Stevil}
```

REŠITVE NALOG ZA DRUGO SKUPINO

N: 46

R1989.2.1

Sproti, ko beremo zapise iz datoteke, lahko tudi primerjamo njihove rezultate s tistim pri novem zapisu; slednjega vrinemo v zaporedje pred prvi tak zapis, ki ima manjši rezultat. Zastavica Vrinjen nam pove, če smo ga že vrinili v zaporedje — da ga ne bi slučajno še enkrat. Če ga ne uspemo vriniti v zaporedje, ga dodamo na konec — očitno ima slabši rezultat od vseh iz datoteke. Na koncu shranimo zapise v datoteko. Na koncu shranimo zapise v datoteko — vse, če jih je deset ali manj, sicer pa le prvih deset.

```
type Top = record
             Ime: array [1..20] of char;
```

```
            Rezultat, Nivo: integer;
          end;
    TopFile = file of Top;

procedure InTop10(var f: TopFile; Novi: Top);
var Zadnji, Tekoci: integer;
    Vrinjen: boolean;
    Top10: array [1..11] of Top;
begin
  Vrinjen := false;
  Reset(f);
  Zadnji := 0;
  while not Eof(f) and (Zadnji < 10) do begin
    Zadnji := Zadnji + 1;
    Read(f, Top10[Zadnji]);
    if (Top10[Zadnji].Rezultat < Novi.Rezultat) and not Vrinjen then begin
      { Novi zapis bo treba vriniti pred ravnokar prebranega. }
      Top10[Zadnji + 1] := Top10[Zadnji];
      Top10[Zadnji] := Novi;
      Zadnji := Zadnji + 1;
      Vrinjen := true;
    end; {if}
  end; {while}
  { Če novega zapisa še nismo vrinili v zaporedje, ga dajmo na konec.
    Če smo prebrali deset zapisov, se bo tam na koncu sicer izgubil,
    če pa smo jih prebrali manj, bo prišel še prav. }
  if not Vrinjen then begin
    Zadnji := Zadnji + 1;
    Top10[Zadnji] := Novi;
  end; {if}
  { Največ deset zapisov shranimo nazaj v datoteko. }
  if Zadnji > 10 then Zadnji := 10;
  Rewrite(f);
  for Tekoci := 1 to Zadnji do Write(f, Top10[Tekoci]);
end; {InTop10}
```

R1989.2.2

N: 46

Ker ključa nismo dobili, si bomo morali pomagati s pogostostmi črk. Pri šifriranju se vsaka črka c spremeni v neko drugo črko $f(c)$; različne črke se preslikajo v različne črke. To pomeni, da, če je bila neka c_1 najpogostejša črka v prvotnem besedilu, bo $f(c_1)$ najpogostejša črka v kodiranem besedilu. Podobno velja za drugo najpogostejšo in tako naprej. Če torej poiščemo najpogostejšo črko v kodiranem besedilu, bi morala biti to koda najpogostejše črke prvotnega besedila (torej tiste, ki jo dobimo v v[1].c — v praksi bi bila to črka e). Druga najpogostejša črka v kodiranem besedilu mora biti koda črke v[2].c in tako naprej. Recimo temu postopku postopek 1.

```
procedure Desifriraj1(var f: text; v: Ver);
var
  Pog: array ['a'..'z'] of integer;
  VrstniRed: array [1..26] of char;
  Kod: array ['a'..'z'] of char;
  c: char; i, j: integer;
begin
  { Preštejmo pogostosti črk v datoteki f. }
  for c := 'a' to 'z' do Pog[c] := 0;
  while not Eof(f) do begin
    Read(f, c);
    if (c >= 'a') and (c <= 'z') then
      Pog[c] := Pog[c] + 1;
  end; {while}
  { Uredimo črke po padajoči pogostosti. }
  for i := 1 to 26 do begin
    j := i - 1; c := Chr(Ord('a') + j);
    while j >= 1 do
      if Pog[VrstniRed[j]] >= Pog[c] then break
      else begin VrstniRed[j + 1] := VrstniRed[j]; j := j - 1 end;
    VrstniRed[j + 1] := c;
  end; {for}
  { i-ta najpogostejša črka iz f naj se dekodira v
    i-to najpogostejšo črko iz v. }
  for i := 1 to 26 do
    Kod[VrstniRed[i]] := v[i].c;
  { Dekodirajmo zdaj vsebino datoteke. }
  Reset(f);
  while not Eof(f) do begin
    Read(f, c);
    if (c >= 'a') and (c <= 'z') then
      c := Kod[c];
    Write(c);
  end; {while}
end; {Desifriraj1}
```

Spodnji program pa uporablja še malo drugačen postopek (recimo mu postopek 2). Za vsako črko kodiranega besedila izračuna njeno pogostost (v odstotkih glede na celotno dolžino kodiranega besedila) in nato poišče najbližjo pogostost med črkami nekodiranih besedil, torej najbližjo med vrednostmi v[i].v. V ta namen okoli vsake v[i].v ustanovi interval, ki sega pol poti do pogostosti prejšnje in naslednje črke, torej od (v[i − 1].v + v[i].v) / 2 do (v[i].v + v[i + 1].v) / 2. Izjema sta najpogostejša in najredkejša črka (pri prvi je zgornja meja kar 100 %, pri zadnji pa je spodnja meja 0 %). Za pogostost vsake črke kodiranega besedila torej pogledamo, v kateri interval pade, in to črko dekodiramo v črko, ki ji pripada najdeni interval. Nerodno pri tem postopku je, da se nam lahko

več črk dekodira v isto, če se pogostosti v obdelovanem besedilu dovolj razlikujejo od tistih v prvotnem; mogoče pa to niti ni tako slabo, če bo zato vsaj ena od teh več črk, ki se dekodirajo v isto, dekodirana pravilno.

```
procedure Desifriraj2(var f: text; v: Ver);
type Interval = record c: char; zg, sp: real end;
var Int: array [1..26] of Interval;
    Pog: array ['a'..'z'] of real;

  procedure IzracunajIntervale;
  var i: integer;
  begin
    for i := 2 to 25 do begin
      Int[i].c := v[i].c;
      Int[i].zg := v[i].v + (v[i -1].v - v[i].v) / 2;
      Int[i].sp := v[i].v - (v[i].v - v[i + 1].v) / 2;
    end; {for}
    Int[1].c := v[1].c; Int[1].sp := Int[2].zg; Int[1].zg := 100;
    Int[26].c := v[26].c; Int[26].zg := Int[25].sp; Int[26].sp := 0;
  end; {IzracunajIntervale}

  procedure PrestejCrke;
  var c: char; Vsota: real;
  begin
    for c := 'a' to 'z' do Pog[c] := 0;
    Vsota:= 0;
    while not Eof(f) do begin
      Read(f, c);
      if (c >= 'a') and (c <= 'z') then
        begin Pog[c] := Pog[c] + 1; Vsota := Vsota + 1 end;
    end; {while}
    if Vsota = 0 then Vsota := 1;    { Datoteka f je prazna. }
    for c := 'a' to 'z' do
      Pog[c] := 100 * Pog[c] / Vsota;
  end; {PrestejCrke}

  procedure Zamenjaj;
  var c: char;

    function VrniCrko(c: char): char;
    var i: integer; Nasel: boolean;
    begin
      Nasel := false;
      VrniCrko := '?';
      i := 1;
      while not Nasel and (i <= 26) do
        if (Pog[c] >= Int[i].sp) and (Pog[c] <= Int[i].zg) then
          begin Nasel := true; VrniCrko := Int[i].c end
```

```
        else i := i + 1;
    end; {VrniCrko}

  begin
    while not Eof(f) do begin
      Read(f, c);
      if (c <= 'z') and (c >= 'a') then
        Write(VrniCrko(c));
    end; {while}
  end; {Zamenjaj}

begin {Desifriraj}
  IzracunajIntervale;
  PrestejCrke;
  Reset(f);
  Zamenjaj;
end; {Desifriraj2};
```

Še ena preprosta, a koristna izboljšava te rešitve bi bila, da ne bi klicali VrniCrko za vsako črko posebej, ampak bi jo poklicali po enkrat za vsako črko od `'a'` do `'z'` in rezultate shranili v neko tabelo; odtlej pa bi za dekodiranje le uporabili ustrezni element te tabele in ne bi bilo treba več vsakič prečesavati celega seznama intervalov.

Naredili smo preprost poskus. Vzeli smo nekaj besedil, spremenili velike črke v male, odstranili vse nečrkovne znake, uporabili eno besedilo za merjenje pogostosti črk v tabeli `v` in nato pogledali, kako bi se odrezala gornja dva postopka, če bi bilo treba dekodirati neko drugo besedilo.

Rezultate prikazuje tabela na str. 59. V oklepajih so dolžine besedil (v milijonih črk). Odstotki napačno dekodiranih znakov upoštevajo tudi pogostost posamezne črke: na primer, če napačno dekodiramo tisto, v kar se je zakodiral *a*, nam da to precej več napak kot pa napačno dekodiranje tistega, v kar se je zakodiral *z*. Besedila so bila: Gibbonova *Zgodovina zatona in propada rimskega cesarstva* (v šestih zvezkih, skupaj 7 492 576 črk); nekaj Dickensovih del (skupaj 15 205 102 črk, ki smo jih razdelili v dve približno enako veliki skupini); in znana zbirka 806 791 Reutersovih člankov (ki smo jih razdelili na 8 delov s po 100 848 ali 100 849 zaporednimi članki). Po starosti si sledijo ta besedila v razmiku približno sto let: Gibbonova *Zgodovina* je bila objavljena v letih 1776–88, Dickensova dela sredi 19. stoletja, Reutersovi članki pa so iz obdobja od 20. avgusta 1996 do 19. avgusta 1997.

Vidimo, da je postopek 1 ponavadi malo boljši, vendar so v praksi napake še vedno kar precejšnje, tudi če imamo veliko besedil in bi torej človek pričakoval, da bi morale biti frekvence precej stabilne. Čeprav so vsa besedila v istem jeziku, se frekvence črk dovolj spreminjajo, da nam to resno otežkoča dekodiranje. Pogosto se na primer zgodi, da imata dve črki zelo podobni frekvenci, tako da je to, katera je malenkost pogostejša od druge, pravzaprav stvar

Besedilo za tabelo v	Besedilo za dekodiranje	% napačno dekodiranih znakov	
		Postopek 1	Postopek 2
Gibbon 1 (1,27 M)	Gibbon 2 (1,41 M)	49,258	39,600⋆
Gibbon 1–3 (3,89 M)	Gibbon 4–6 (3,60 M)	24,220⋆	31,781
David Copperfield (1,49 M)	Nicholas Nickleby (1,43 M)	27,707⋆	39,816
Dickens 1 (7,63 M)	Dickens 2 (7,56 M)	17,425⋆	29,140
Reuters 1 (114,0 M)	Reuters 2 (107,5 M)	0,000⋆	10,142
Reuters 1–2 (221,5 M)	Reuters 3–4 (223,0 M)	17,198⋆	20,347
Reuters 1–4 (444,5 M)	Reuters 5–8 (446,1 M)	3,201	1,862⋆
Gibbon 1–6 (7,49 M)	Dickens 1–2 (15,20 M)	53,536⋆	54,772
Gibbon 1–6 (7,49 M)	Reuters 1–8 (890,6 M)	63,381	61,505⋆
Dickens 1–2 (15,20 M)	Reuters 1–8 (890,6 M)	70,384	66,052⋆

⋆ = boljši

Primerjava dveh rešitev naloge 1989.2.2.

naključja, torej se metoda 1 pri teh dveh črkah čisto lahko zmoti; podobno pa se lahko frekvence hitro spremenijo vsaj za toliko, da se zmoti pri kakšni od teh črk tudi metoda 2. Levji delež napake pri dekodiranju Reuters 3–4 s frekvencami iz Reuters 1–2 izvira ravno od tega, ker je na Reuters 1–2 druga najpogostejša črka *t* (8,62 %), tretja pa *a* (8,59 %), na Reuters 3–4 pa je ravno obratno (*a* 8,64 %; *t* 8,55 %), tako da pri dekodiranju oba postopka pobrkljata *a* in *t*.

Deset najpogostejših črk (in njihove pogostosti v promilih):

Gibbon	*e* 130	*t* 94	*a* 79	*o* 76	*i* 75	*s* 67	*n* 67	*r* 63	*h* 57	*d* 41
Dickens	*e* 122	*t* 89	*a* 80	*o* 77	*n* 71	*i* 70	*s* 62	*h* 62	*r* 59	*d* 45
Reuters	*e* 119	*a* 86	*t* 86	*n* 76	*i* 75	*o* 73	*s* 70	*r* 68	*l* 42	*d* 41

Najbrž bi si bilo pri dekodiranju pametno pomagati še s slovarjem angleških besed — ko razmišljamo o tem, v kaj bi dekodirali neko črko, bi bilo dobro preveriti, da nam ne bo nastalo v dekodiranem besedilu ogromno besed, ki jih ni v slovarju. Zmedi med črkama *a* in *t* bi se lahko izognili s pomočjo dejstva, da je *the* v angleščini običajno daleč najpogostejša tročrkovna beseda (pravzaprav najpogostejša beseda sploh), tako da ni težko ugotoviti, v kaj se je zakodiral *t*.

R1989.2.3

V številskem sestavu z osnovo b imajo števke vrednosti $0, 1, \ldots, b-1$. Če številu z vrednostjo n na desni pripišemo števko c, se mu vrednost spremeni v $n \cdot b + c$. Zato ni težko izračunati, kakšno vrednost predstavlja neko število, zapisano v b-iškem sestavu: začnemo z 0, nato pa beremo števke od leve proti desni, število vsakič pomnožimo z b in mu prištejemo pravkar prebrano števko.

N: 47

Za pretvorbo v obratno smer, torej vrednosti v zaporedje števk, moramo izvajati obratne operacije. Zadnja (najbolj desna) števka števila n je (če je

$n \geq 0$) kar ostanek po deljenju n-ja z osnovo b, torej $n \bmod b$. Celi del količnika n/b pa je ravno število n brez te skrajno desne števke (saj za $n \operatorname{div} b$ res velja, da če mu pripišemo števko $n \bmod b$, dobimo n: $n = (n \operatorname{div} b) \cdot b + (n \bmod b)$). Tako lahko pulimo iz n-ja števke eno za drugo, od desne proti levi. Pred izpisom bomo morali njihov vrstni red še obrniti, saj moramo najprej izpisati bolj leve števke (tiste z največjo težo).

```
const MaxDolzina = 15;
type Niz = packed array [1..MaxDolzina] of char;

procedure Pretvori(var Stevilo: Niz; IzOsnove, VOsnovo: integer; var Dolzina: integer);
const Stevke = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
var i, T, Vrednost: integer;
begin
  { Izračunajmo vrednost, ki jo predstavlja niz Stevilo. }
  Vrednost := 0;
  for i := 1 to Dolzina do begin
    T := 0;   { Katero števko predstavlja znak Stevilo[i]? }
    while (T < IzOsnove) and (Stevke[T + 1] <> Stevilo[i]) do
      T := T + 1;
    if T >= IzOsnove then
      begin WriteLn('Neveljavna števka: ', Stevilo[i]); T := 0 end;
    Vrednost := Vrednost * IzOsnove + T;
  end; {for}
  { Koliko števk bo imelo to število v zapisu z osnovo VOsnovo? }
  Dolzina := 0; T := Vrednost;
  repeat
    T := T div VOsnovo;
    Dolzina := Dolzina + 1;
  until T = 0;
  { V niz Stevilo vpišimo predstavitev števila Vrednost z osnovo VOsnovo. }
  for i := Dolzina downto 1 do begin
    Stevilo[i] := Stevke[(Vrednost mod VOsnovo) + 1];
    Vrednost := Vrednost div VOsnovo;
  end; {for}
end; {Pretvori}
```

N: 47

R1989.2.4

Za začetek bomo ves labirint zazidali — v vse celice postavimo zidove. Potem se bomo postavili v nek začetni položaj, ga „odzidali“ in se začeli premikati iz njega v sosednje celice ter pri tem spreminjati zidove v poti. To bo zagotovilo, da bo labirint povezan. Na vsakem položaju naredimo naslednje: če sta v kakšni od štirih smeri (gor, dol, levo, desno) ob trenutni celici dve zaporedni zazidani celici, ju obe spremenimo v prosti celici in se premaknemo v drugo od njiju (na primer, če smo na (x, y)

in sta tako $(x, y-1)$ kot $(x, y-2)$ zazidani, ju spremenimo v prosti in se premaknimo v $(x, y-2)$); če je mogoč tak premik v več smeri, si smer izberemo naključno; če pa ni nobene take smeri, se umaknemo nazaj (po poti, po kateri smo prišli) na položaj, iz katerega smo prvikrat prišli v trenutni položaj. Da se ne bo dalo priti ven iz labirinta, morajo ostati prvi in zadnji stolpec in vrstica popolnoma zazidani; to zagotovimo, če se delamo, da so zunaj labirinta same poti.

	15	14	13		113		103	102	85	86	87	88	89	90	91	92	93		101	
	16		12		112		104		84								94		100	
	17		11		111	110	105		83	82	81	80	79		97	96	95	98	99	
	18		10				106						78							
	19		9	8	7		107	108	109		75	76	77		67	66	65	64	63	
	20				6						74				68				62	
	21	22	23		5	4	3	2	1		73	72	71	70	69	114	115		61	
			24																60	
	119		25		31	32	33	34	35		43	44	45		51	52	53	54	55	
	118		26		30				36		42		46		50				56	
	117	116	27	28	29		39	38	37	40	41		47	48	49		59	58	57	

Primer labirinta. Osenčene celice so zazidane, številke v prostih celicah pa povedo, v kakšnem vrstnem redu jih je obiskoval naš program.

Označimo x-koordinate celic z $0, \ldots, m-1$ in y-koordinate z $0, \ldots, n-1$; naj bo (x_0, y_0) naš začetni položaj (ki si ga izberemo naključno; omejitev je le ta, da morata biti, kot bomo videli, obe tidve koordinati lihi). Ker se vedno premaknemo za dve celici naenkrat v isto smer, bodo vsi naši kasnejši položaji oblike (x_0+2a, y_0+2b) za neki celi števili a in b. Navpični hodniki (torej skupine dveh ali več nezazidanih celic, ki se dotikajo z vodoravnimi stranicami) imajo torej vedno x-koordinato oblike $x_0 + 2a$ in pokrivajo y-koordinate od $y_0 + 2b$ do $y_0 + 2b'$ za neke a, b, b'. Podobno vodoravni hodniki pokrivajo x-koordinate od $x_0 + 2a$ do $x_0 + 2a'$ pri y-koordinati $y_0 + 2b$, za neke a, a', b.

No, rekli smo, da bo moral ostati zunanji rob (stolpca $x = 0$ in $x = m-1$ ter vrstici $y = 0$ in $y = n-1$) labirinta zazidan. Ker so $(0, y)$ sami zidovi, ne smeta biti nobeni dve celici $(1, y)$ in $(1, y+1)$ hkrati zazidani, ker bi imeli v tem primeru debel zid (kvadrat 2×2 samih zidov), tega pa nočemo. Torej bo vsaj vsaka druga celica tega stolpca prosta (nezazidana); vsaka prosta celica pa je del nekega vodoravnega in/ali nekega navpičnega hodnika. Če je del vodoravnega hodnika, se ta tu konča (saj na $x = 0$ ne more iti), torej je $x = 1$ oblike $x_0 + 2a$; če je del navpičnega hodnika, pa mora spet veljati (spomnimo se lastnosti hodnikov, ki smo jih omenili v prejšnjem odstavku), da je $x = 1$ oblike $x_0 + 2a$. V vsakem primeru sledi $x_0 = 1 - 2a$, torej je x_0 lih. Podoben razmislek bi lahko zdaj začeli tudi iz dejstva, da so v vrhnji vrstici sami zidovi, torej v vrstici $y = 1$ ne moreta biti po dve sosednji celici zazidani, itd., in prišli

bi do ugotovitve, da mora biti tudi y_0 lih. (Če bi tako razmišljali še za skrajno desni stolpec, $x = m-1$, in najnižjo vrstico, $y = n-1$, bi videli tudi, da mora biti x_0 po parnosti enak vrednosti $m-2$, y_0 pa vrednosti $n-2$. Torej morata biti $m-2$ in $n-2$ tudi liha, zato pa tudi m in n; no, besedilo naloge na srečo obljublja, da to zagotovo velja.) Kakorkoli že, zdaj smo ugotovili, da morata biti koordinati našega začetnega položaja obe lihi.

?	?	H	?	?
?	F	C	G	?
?	D	B	E	?
?	?	A	?	?

Slika 1.

?	?	?	?
?	V	W	U
?	X	Y	?
?	?	?	?

Slika 2.

Ali se lahko našemu algoritmu zgodi, da v labirintu dobi cikle? Da bi nastal cikel, bi morali v nekem trenutku podreti zid v celici, ki ima že dve nezazidani sosedi. Recimo, da je A naš trenutni položaj in podremo celici B in C (glej sliko 1). C je očitno pred tem še zazidana, sicer tega sploh ne bi počeli; torej je podiranje zidu B lahko problem le, če je od prej že prosta D ali pa E. Toda da bi bila katera od njiju podrta, bi morala biti del vodoravnega ali pa navpičnega hodnika, toda za vodoravni hodnik ima napačno parnost y-koordinate, za navpičnega pa napačno parnost x-koordinate (ker ima v obeh primerih nasprotno parnost kot ustrezna A-jeva koordinata, ta pa ima pravo, saj je A trenutni položaj). Torej podiranje zidu B ni problem. Kaj pa podiranje zidu C? To bi bil problem, če bi bila od prej prosta katera od F, G in H. F ne more biti del navpičnega hodnika (zaradi parnosti x-koordinate), če pa bi bila del vodoravnega, ne more biti njegov začetek ali konec; torej bi bila v tem primeru C vsekakor že tudi podrta, kar je protislovje. Enak razmislek velja za G, podoben pa tudi za H, ki zaradi parnosti svoje y-koordinate ne more biti del vodoravnega hodnika niti konec navpičnega, torej bi bila tudi C zagotovo že prosta, če bi bila prosta H. Torej tudi podiranje zidu C ni problem.

Ta razmislek nam zagotavlja tudi, da v labirintu ne bomo dobili velikih soban, npr. kvadrata 2×2 samih nezazidanih celic, saj je to tudi že cikel.

Ali se lahko našemu algoritmu zgodi, da v labirintu pusti debele zidove, torej nekje recimo kvadrat 2×2 zazidanih celic? Recimo, da obstaja tak kvadrat; gotovo si lahko izberemo takega, ki ima vsaj eno prosto sosedo, recimo U (glej sliko 2). Ker je U prosta, je del vodoravnega in/ali navpičnega hodnika. Če vodoravnega, se ta pri U očitno konča, torej je bil v U nekoč trenutni položaj; toda takrat bi opazili, da sta W in V zazidani, in bi ju prekopali; torej je to nemogoče. Torej je U del navpičnega hodnika; če se ta hodnik konča pri U, je bil nekoč tu trenutni položaj in imamo enak problem kot prej; če pa se ne konča pri U, se mora nadaljevati tudi pri U-jevi spodnji sosedi

in je vsaj ena od njiju bila nekoč trenutni položaj in bi morali takrat opaziti in izkopati ali V in W (če je bil trenutni položaj U) ali pa X in Y (če je bil trenutni položaj v U-jevi spodnji sosedi). — Analogno lahko razmišljamo v primeru, ko je U ob zgornji, levi ali desni stranici našega kvadrata.

Tako smo se prepričali, da naš algoritem poišče labirinte, ki ustrezajo vsem zahtevam naloge: povezani, aciklični, brez velikih sob in debelih zidov.

```
const XS = 77; YS = 21;
type Celica = (Zid, Pot);
var Tabela: array [0..XS - 1, 0..YS - 1] of Celica;

function JeZid(x, y: integer): boolean;
begin
  if (x < 0) or (x >= XS) or (y < 0) or (y >= YS) then JeZid := false
  else JeZid := Tabela[x, y] = Zid;
end; {JeZid}

procedure Labirint(x, y: integer);
var nx, ny: integer;
begin
  Tabela[x, y] := Pot;
  while JeZid(x - 2, y) or JeZid(x + 2, y) or
        JeZid(x, y - 2) or JeZid(x, y + 2) do begin
    nx := x; ny := y;
    case Random(4) of
      0: nx := x + 2;
      1: nx := x - 2;
      2: ny := y + 2;
      3: ny := y - 2;
    end; {case}
    if JeZid(nx, ny) then begin
      Tabela[(x + nx) div 2, (y + ny) div 2] := Pot;
      Labirint(nx, ny);
    end; {if}
  end; {while}
end; {Labirint}

procedure ZazidajTabelo;
var x, y: integer;
begin
  for x := 0 to XS - 1 do
    for y := 0 to YS - 1 do
      Tabela[x,y] := Zid;
end; {ZazidajTabelo}

procedure PrikaziTabelo;
var x, y: integer;
```

```
begin
  for y := 0 to YS − 1 do begin
    for x := 0 to XS − 1 do
      if Tabela[x, y] = Zid then Write('#') else Write('.');
    WriteLn;
  end; {for}
end; {PrikaziTabelo}

begin
  ZazidajTabelo;
  Labirint(2 * Random(XS div 2) + 1, 2 * Random(YS div 2) + 1);
  PrikaziTabelo;
end.
```

Zgoraj opisani postopek za speljevanje poti po labirintu se je pravzaprav zgledoval po preiskovanju grafov v globino, le da si naključno izbira, v kakšnem vrstnem redu bo preiskoval sosede trenutnega položaja. Lahko pa bi se namesto po tem zgledovali tudi po Primovem ali pa po Kruskalovem algoritmu za iskanje minimalnih vpetih dreves v grafu. Primov algoritem bi gradil labirint tako, da bi v vsakem koraku naključno izbral eno od prostih polj (x, y), eno od štirih možnih smeri in nato podrl zidova v dveh sosednjih poljih v tej smeri (če je to dopustno, torej če sta na obeh teh dveh poljih res zidova). Kruskalov algoritem pa bi za začetek podrl zidove na vseh poljih, ki imajo obe koordinati lihi, nato pa bi si naključno izbiral po neko polje in neko smer ter podrl zid v sosednjem polju v tisti smeri, če seveda ni mogoče do polja, ki je od trenutnega oddaljeno za dva koraka v tisti smeri, priti že zdaj po kakšni bolj oddaljeni poti). Pri tem algoritmu bi prišla prav znana podatkovna struktura za disjunktne množice,[9] s katero bi lahko učinkoviteje preverjali, ali sta neki dve polji že povezani s potjo ali ne; vendar pa, ker so naši labirinti bolj majhni, uporablja spodnji podprogram kar tabelo, v kateri za vsako točko piše, kateri povezani komponenti labirinta pripada.

```
procedure LabirintPrim(x0, y0: integer);
type TockaT = record x, y: integer end;
var ToDo: array [1..XS * YS] of TockaT;
    nToDo, x, y, nx, ny, i: integer;
begin
  nToDo := 1; with ToDo[1] do begin x := x0; y := y0 end;
  Tabela[x0, y0] := Pot;
  while nToDo > 0 do begin
    { Naključno izberimo kakšno izmed tistih celic, ki imajo še kakšno primerno
      zazidano sosedo. Seznam takih celic je v tabeli ToDo. }
    i := Random(nToDo) + 1; x := ToDo[i].x; y := ToDo[i].y;
```

[9] Glej npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 22.3 v prvi izdaji, 21.3 v drugi.

```
      if not (JeZid(x − 2, y) or JeZid(x + 2, y) or
              JeZid(x, y − 2) or JeZid(x, y + 2)) then begin
        { Vse njene sosede smo že pregledali — pobrišimo jo iz množice ToDo. }
        ToDo[i] := ToDo[nToDo]; nToDo := nToDo − 1;
      end
      { Izberimo eno od zazidanih sosed trenutne celice in tisti zid porušimo.
        Sosedo dodajmo na seznam ToDo, da bomo o priliki pregledali še njene }
      else while true do begin                                    { sosede. }
        nx := x; ny := y;
        case Random(4) of
          0: nx := x + 2;   1: nx := x − 2;   2: ny := y + 2;   3: ny := y − 2;
        end; {case}
        if JeZid(nx, ny) then begin
          Tabela[(x + nx) div 2, (y + ny) div 2] := Pot;
          Tabela[nx, ny] := Pot;
          nToDo := nToDo + 1; ToDo[nToDo].x := nx; ToDo[nToDo].y := ny;
          break;
        end; {if}
      end; {while}
    end; {while}
end; {LabirintPrim}

procedure LabirintKruskal;
type TockaT = record x, y: integer end;
var ToDo: array [1..XS * YS] of TockaT;
    nToDo, x, y, xt, yt, k, nk, nx, ny, i: integer;
    Komp: array [1..XS, 1..YS] of integer;

  function IstaKomp(x, y, KotKomp: integer): boolean;
  begin
    if (x < 0) or (x >= XS) or (y < 0) or (y >= YS) then IstaKomp := true
    else IstaKomp := Komp[x, y] = KotKomp;
  end; {IstaKomp}

begin
  { Labirint bo med gradnjo razdeljen na „komponente“ — dele labirinta,
    ki so z zidovi povsem ločeni drug od drugega. Na začetku je vsaka celica,
    ki ima lihi koordinati, sama svoja komponenta. }
  nToDo := 0;
  for x := 0 to (XS div 2) − 1 do for y := 0 to (YS div 2) − 1 do begin
    nToDo := nToDo + 1; Komp[2 * x + 1, 2 * y + 1] := nToDo;
    ToDo[nToDo].x := 2 * x + 1; ToDo[nToDo].y := 2 * y + 1;
    Tabela[2 * x + 1, 2 * y + 1] := Pot;
  end; {for}
  { Dokler obstaja več kot ena komponenta: izberimo naključno neko celico
    in neko njeno sosedo; če nista v isti komponenti, zid med njima porušimo
    (obe komponenti se s tem zlijeta v eno samo). }
```

Algoritem	Povprečno število korakov pri tavanju	Povprečna dolžina najkrajše poti
Iskanje v globino	$504{,}2 \pm 126{,}1$	$284{,}5 \pm 64{,}4$
Primov algoritem	$434{,}0 \pm 135{,}0$	$106{,}8 \pm 8{,}2$
Kruskalov algoritem	$453{,}2 \pm 95{,}1$	$149{,}0 \pm 19{,}0$

Primerjava rešitev naloge 1989.2.4.

```
  while nToDo > 0 do begin
    i := Random(nToDo) + 1; x := ToDo[i].x; y := ToDo[i].y; k := Komp[x, y];
    if IstaKomp(x − 2, y, k) and IstaKomp(x + 2, y, k) and
      IstaKomp(x, y − 2, k) and IstaKomp(x, y + 2, k) then begin
      ToDo[i] := ToDo[nToDo]; nToDo := nToDo − 1;
    end
    else while true do begin
      nx := x; ny := y;
      case Random(4) of
        0: nx := x + 2;   1: nx := x − 2;   2: ny := y + 2;   3: ny := y − 2;
      end; {case}
      { Porušimo zid med celicama in zlijmo njuni komponenti. }
      if not IstaKomp(nx, ny, k) then begin
        Tabela[(x + nx) div 2, (y + ny) div 2] := Pot;
        nk := Komp[nx, ny];
        for xt := 0 to (XS div 2) − 1 do for yt := 0 to (YS div 2) − 1 do
          if Komp[2 * xt + 1, 2 * yt + 1] = nk then
            Komp[2 * xt + 1, 2 * yt + 1] := k;
        break;
      end; {if}
    end; {while}
  end; {while}
end; {LabirintKruskal}
```

Po naših opažanjih imajo labirinti, ki jih pripravlja Primov algoritem, več daljših hodnikov kot tisti pri iskanju v globino; labirinti, dobljeni s Kruskalovim algoritmom, pa so še bolj zaviti kot tisti, dobljeni z iskanjem v globino. Poskusili smo tudi oceniti, kako težko je priti z enega konca labirinta do drugega, torej od polja $(1, 1)$ do $(m-2, n-2)$. Pri tem si mislimo, da se v vsakem koraku, če imamo več možnih smeri za nadaljevanje poti, naključno odločimo za eno od tistih, v katere še nismo šli; če pa pridemo v slepo ulico, gremo nazaj v smeri, od koder smo prišli, dokler ne pridemo do prvega takega križišča, iz katerega vodi kakšna še neprehojena smer. Tabela na vrhu te strani kaže povprečno število polj, ki smo jih morali pri takem tavanju prehoditi, da smo prišli od zgornjega levega do spodnjega desnega kota; kaže pa tudi število polj pri najkrajši poti. Vse številke so povprečja prek 10000 naključnih labirintov.

REŠITVE NALOG ZA TRETJO SKUPINO

R1989.3.1

N: 47

Ker vemo, da ni nobeden od vzorcev daljši od sto znakov, je dovolj, če si med branjem datoteke zapomnimo le zadnjih sto prebranih znakov. Po branju vsakega znaka pa pojdimo po vseh vzorcih in preglejmo, če se kakšen od njih mogoče pojavi v datoteki tako, da se pojavitev konča prav pri ravnokar prebranem znaku. Zadnjih sto znakov bomo hranili v tabeli Okno, ki jo bomo uporabljali kot krožni pomnilnik: stoprvi znak bomo zapisali v Okno[1] (in s tem povozili prvega, ki pa ga tako ali tako ne bomo več potrebovali), stodrugi znak v Okno[2], dvestodrugega spet v Okno[2] in tako naprej. Seveda moramo biti zato pri delu z indeksi previdni — delati se moramo, kot da pred znakom Okno[1] pride znak Okno[100].

```
const MaxDolz = 100;
type Vzorci = array [1..20] of record
                  Zlog: array [1..MaxDolz] of byte;
                  Dolzina: integer;
                end;
     Datoteka = file of byte;

procedure Isci(var F: Datoteka; V: Vzorci; N: integer);
var Okno: array [1..MaxDolz] of byte;
    Polozaj, Konec, i, j, Prebrano: integer;
begin
    Prebrano := 0; Konec := 0;
    while not Eof(F) do begin
     Konec := Konec + 1; if Konec > MaxDolz then Konec := 1;
     Read(F, Okno[Konec]);
     if Prebrano < MaxDolz then Prebrano := Prebrano + 1;
     for i := 1 to N do with V[i] do if Prebrano >= Dolzina then begin
       { S številom j se bomo sprehajali po trenutnem vzorcu, V[i].Zlog[...],
         s številom Polozaj pa po vsebini okna (zadnjih V[i].Dolzina
         prebranih zlogov). }
       Polozaj := Konec - Dolzina + 1; j := 1;
       if Polozaj <= 0 then Polozaj := Polozaj + MaxDolz;
       while j <= Dolzina do begin
         if Okno[Polozaj] <> Zlog[j] then break;
         j := j + 1; Polozaj := Polozaj + 1;
         if Polozaj > MaxDolz then Polozaj := 1;
       end; {while}
       if j > Dolzina then { Vse se ujema! }
         WriteLn('zaporedje ', i, ' odmik ', FilePos(F) - Dolzina);
     end; {if, with, for}
   end; {while}
end; {Isci}
```

Ta postopek bi lahko še izboljšali, če bi se zgledovali po kakšnem od znanih postopkov za iskanje podnizov v nizih, npr. Knuth-Morris-Prattovem ali Boyer-Moorovem.[10]

N: 48

R1989.3.2

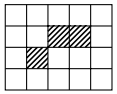
Naloga pravi, da že poznamo najbolj levo zgornjo točko površine, ki nas zanima. Ta torej že pripada notranjemu robu površine, od tu naprej pa bo najbolj učinkovito, če se bomo premikali ves čas po tem robu, recimo v smeri urinega kazalca. Vsako polje, ki ga obiščemo, označimo kot rob (vanj vpišemo -1), nato pa se moramo odločiti, v kateri smeri nadaljevati. Možnih smeri je le osem, saj ima vsako polje v karirasti mreži le osem sosedov. Smer naslednjega premika je lahko ista kot smer prejšnjega premika, lahko pa se glede na tisto smer tudi odkloni za 45, 90 ali 135 stopinj v levo ali desno. (Odklon za 180° pa nas ne zanima, saj bi ta že pomenil vrnitev nazaj na prejšnji položaj.) Ker se gibljemo v smeri urinega kazalca in hočemo ostati na robu površine (ne pa iti v notranjost), moramo ves čas siliti karseda v levo. Torej poskusimo smer premika najprej spremeniti za 135° v levo; če opazimo, da bi nas to vrglo iz površine, poskusimo le 90° v levo in tako naprej. Pred prvim premikom, torej ko smo še na začetnem položaju, se lahko delamo, kot da smo sem prišli s premikom v desno.

Spodnji program ima smerne vektorje za vseh osem smeri navedene v smeri urinega kazalca (če si mislimo, da os y kaže navzdol) v tabeli Okolica. Zasuku za 135° v levo tako ustreza premik za tri mesta nazaj po tej tabeli; ali pa, kar je isto, za pet mest naprej (saj je 135° v levo isto kot 225° v desno).

```
const N = ...; M = ...; var Polje: array [1..N, 1..M] of integer;

{ Pove, če je dana celica prosta. Zunaj polja si mislimo same proste celice. }
function Prosta(X, Y: integer): boolean;
begin
  if (X < 1) or (Y < 1) or (X > N) or (Y > M)
  then Prosta := false else Prosta := Polje[X, Y] = 0;
end; {Prosta}

procedure Meja(ZacX, ZacY: integer);
const Okolica: array [0..7] of record X, Y: integer end =
  ( (X: 0; Y: 1), (X: −1; Y: 1), (X: −1; Y: 0), (X: −1; Y: −1),
    (X: 0; Y: −1), (X: 1; Y: −1), (X: 1; Y: 0), (X: 1; Y: 1) );
var i, LD, X, Y, Xn, Yn, X1, Y1: integer; Prvi: boolean;
begin
  X := ZacX; Y := ZacY; Prvi := true;
  LD := 3; { Delajmo se, da smo na trenutni položaj prišli s premikom v desno. }
  repeat
    Polje[X, Y] := −1;
```

[10] Gl. npr. Cormen *et al.*, *Introduction to Algorithms*, 34.4–5 v prvi izdaji, 32.4 v drugi; Boyer-Moore v slednji žal ni opisan.

```
    { Poglejmo, v kateri smeri moramo nadaljevati. }
    i := 0; while (i < 8) and Prosta(X + Okolica[LD].X, Y + Okolica[LD].Y) do
      begin LD := (LD + 1) mod 8; i := i + 1 end;
    if i >= 8 then break; { Imamo „lik“ velikosti 1 × 1. }
    Xn := X + Okolica[LD].X; Yn := Y + Okolica[LD].Y;
    { Če smo korak (X, Y) → (Xn, Yn) nekoč že naredili, smo
      zdaj prišli že okoli in okoli lika in se moramo ustaviti. }
    if Prvi then begin X1 := Xn; Y1 := Yn; Prvi := false end
    else if (X = X0) and (Y = Y0) and (Xn = X1) and (Yn = Y1) then break;
    X := Xn; Y := Yn;
    { Na naslednjem položaju moramo začeti preizkušati smeri pri LD − 3, }
    LD := (LD + 5) mod 8;                { torej z zasukom 135° v levo. }
  until false;
end; {Meja}
```



Podprogram Meja si zapomni prvi opravljeni premik (od (X0, Y0) do (X1, Y1)) in se ustavi, ko se ta premik prvič ponovi. Če bi preverjali le to, kdaj se prvič vrnemo na začetni položaj (X0, Y0), bi imeli težave v primerih, ko je treba začetni položaj obiskati večkrat, ker leži istočasno na spodnji in na zgornji meji površine (glej sliko na desni).

Naša rešitev tudi predpostavlja, da površina, ki nas zanima, v notranjosti nima kakšnih lukenj, okoli katerih bi morali tudi označevati rob. Take luknje bi lahko odkrili s pregledovanjem notranjosti površine, nato pa bi za vsako odkrito luknjo po postopku, podobnem kot zgoraj, označili njen zunanji rob, ki je z vidika površine pač notranji rob.

R1989.3.3

N: 48

Preprost postopek bi bil naslednji: izberimo si dve točki in potegnimo skoznju premico. Z malo sreče pokriva ta premica še kakšno drugo točko, ne le tistih dveh, iz katerih smo jo dobili. Kakorkoli že, pokrite točke zdaj zbrišimo iz naše množice in se v nadaljevanju postopka na enak način lotimo preostalih točk. Tako pokrivamo točke, dokler ne pokrijemo vseh.

Pri tem nimamo nobenega zagotovila, da bo dobljena rešitev kaj prida. Če imamo n točk, bi se lahko (z nekaj smole) zgodilo, da bi pokrila vsaka naša premica le dve točki (torej jih potrebujemo $\lceil n/2 \rceil$), obenem pa bi obstajala neka rešitev s samo dvema premicama; torej bi bila naša rešitev vsaj $n/4$-krat slabša ob najboljše možne.

Ta postopek lahko izboljšamo, če vedno izberemo tako premico, ki pokrije največ doslej še nepokritih točk. Pokazati je mogoče, da na ta način porabimo v najslabšem primeru $(1+\ln n)$-krat toliko premic kot pri najboljši možni rešitvi (taki z najmanj premicami). Problem pokrivanja točk s premicami je NP-težak in zaenkrat menda ni znan noben učinkovit postopek s kakšnim boljšim zagotovilom o kakovosti svojih rešitev.[11] Lahko pa bi približne rešitve iskali

[11] Dokaz NP-težkosti: N. Megiddo, A. Tamir: *On the complexity of locating linear fa-*

tudi z raznimi naključnimi postopki, na primer s simuliranim ohlajanjem; to bi verjetno dajalo še boljše rešitve, le da zdaj ne bi imeli teoretičnega zagotovila o kakovosti dobljenih rešitev.

Požrešni postopek bi lahko izvedli takole: če se postavimo v neko točko T in uredimo ostale po kotu, iz katerega se jih vidi iz T, lahko hitro ugotovimo, katere ležijo na istih premicah.[12] To naredimo pri vsaki T in tako dobimo vse premice, za vsako pa tudi vidimo, katere točke ležijo na njej. Obenem si še za vsako točko zapomnimo, na katerih premicah leži. Ko nekaj točk pobrišemo, ker smo jih pokrili z eno od premic, moramo za vse preostale premice zmanjšati število točk, ki ležijo na njih. Da nam bo vedno pri roki podatek o tem, katera premica pokriva največ točk, imamo lahko premice v kopici ali pa za vsako možno število točk nek seznam premic, ki pokrivajo točno toliko točk, in potem ob brisanju točke preselimo ustrezne premice v nižji seznam. Iskanje vseh premic nam bo vzelo $O(n^2 \lg n)$ časa (zaradi urejanja pri vsaki točki), nato pa bomo morali ob brisanju vsake točke iti po vseh premicah, ki so jo pokrivale, teh pa je največ $O(n)$, tako da bo ta del postopka porabil $O(n^2)$ časa. No, če se hočemo dosledno držati navodila iz naloge, češ da naj vsako točko obravnavamo kot krog s polmerom 10^{-4}, si je pri ugotavljanju, kaj vse leži na isti premici, težko pomagati s koti, saj se kota dveh točk glede na T lahko razlikujeta tudi za $90°$, če je ena od njiju zelo blizu T-ja (bližje kot za 10^{-4} na primer). V tem primeru bi še vedno lahko lepo naivno za vsak par točk določili premico skoznju in nato eksplicitno pregledali vse ostale točke, da bi videli, katere še ležijo na premici; to bi nam vzelo $O(n^3)$ časa.

Kako bi preverili, če leži neka točka $\vec{r} = (x, y)$ na isti premici kot točki $\vec{r}_1 = (x_1, y_1)$ in $\vec{r}_2 = (x_2, y_2)$? Lahko se delamo, da imamo 3-d vektorje (z z-koordinato 0) in si pomagamo z vektorskim produktom: če leži $\vec{r}$ na isti premici kot $\vec{r}_1$ in $\vec{r}_2$, sta $\vec{r}_2 - \vec{r}_1$ in $\vec{r} - \vec{r}_1$ vzporedna, tedaj pa je njun vektorski produkt enak 0. Označimo $\vec{r}_2 - \vec{r}_1$ z $\vec{d} = (d_x, d_y)$, vektor $\vec{r} - \vec{r}_1$ pa z

---

cilities in the plane, Operations Research Letters, 1(5):194–197 (1982). Ta problem lahko prevedemo na pokrivanje množic (*set covering*) — ustanovimo po eno množico za vsako premico, v njej pa so tiste točke, ki ležijo na tej premici. Za pokrivanje množic pa je znano, da vrača požrešni postopek največ $(1 + \ln n)$-krat slabše rešitve od optimalnih (D. S. Johnson: *Approximation algorithms for combinatorial problems*, Journal of Computer and System Sciences, 9:256–287, 1974; in zgodnejša različica v Proc. STOC 1973, 38–49); pravzaprav smo lahko še natančnejši: če nobena premica ne pokrije več kot m točk, vrne požrešni algoritem rešitev, ki je največ $(\sum_{k=1}^{m} 1/k)$-krat slabša od optimalne (vsota v oklepajih se imenuje „m-to harmonično število“ in znaša približno $\ln m + 0{,}577$).

Če pa uvedemo pri pokrivanju točk s premicami še dodatno zahtevo, da morajo biti premice vzporedne s koordinatnima osema, postane problem lažji in lahko v polinomskem času dobimo optimalne rešitve (R. Hassin, N. Megiddo: *Approximation algorithms for hitting objects with straight lines*, Discrete Applied Mathematics, 30(1):29–42, 1989).

[12] Če se nam ne bi bilo treba ubadati z numeričnimi nenatančnostmi, bi lahko kote metali kar v razpršeno tabelo in tako še lažje ugotovili, ali vidimo več točk pod istim kotom. Če bi bile koordinate naših točk recimo celoštevilske (ali pa vsaj racionalne), bi lahko namesto kotov uporabljali kar smerne koeficiente premic.

$\vec{u} = (u_x, u_y)$. Vektorski produkt $(d_x, d_y, 0) \times (u_x, u_y, 0)$ je $(0, 0, d_x u_y - d_y u_x)$; preveriti moramo torej le, če je $d_x u_y - d_y u_x$ približno enako 0. Naša naloga pravi, naj gledamo točke kot kroge s polmerom $\varepsilon = 10^{-4}$; če bi premik $\vec{u}$ izrazili kot vsoto premika v smeri $\vec{d}$ in premika v smeri pravokotno na $\vec{d}$, bi morali torej zdaj v resnici preverjati, če je ta premik v smeri pravokotno na $\vec{d}$ krajši od ε. Ta premik je v resnici dolg $|\vec{u}| \cdot \sin\alpha$, če je α kót med vektorjema $\vec{u}$ in $\vec{d}$. Vektorski produkt $\vec{d} \times \vec{u}$ pa je po definiciji dolg $|\vec{d}| \cdot |\vec{u}| \cdot \sin\alpha$; torej bo dovolj, če preverimo, ali je $|d_x u_y - d_y u_x| / |\vec{d}| \le \varepsilon$ oz. $(d_x u_y - d_y u_x)^2 \le \varepsilon^2 (d_x^2 + d_y^2)$.

```
program Pokrivanje;

const MaxTock = 10;

type
  PPremica = ↑TPremica;
  PTockaNaPremici = ↑TTockaNaPremici;

  TTocka = record
    x, y: real;                 { koordinati }
    pp: PTockaNaPremici;        { prva premica, ki vsebuje to točko }
  end;

  TTockaNaPremici = record
    it: integer;                { indeks točke }
    p: PPremica;                { kazalec na premico }
    pt, nt: PTockaNaPremici;    { prejšnja in naslednja točka na tej premici }
    np: PTockaNaPremici;        { naslednja premica, ki vsebuje to točko }
  end;

  TPremica = record
    pt: PTockaNaPremici;        { prva točka na tej premici }
    n: integer;                 { število točk na tej premici }
    pp, np: PPremica;           { prejšnja in naslednja premica z n točkami }
  end;

var
  { Premice[k] je seznam premic, ki pokrivajo k točk. }
  Premice: array [1..MaxTock] of PPremica;
  t: array [1..MaxTock] of TTocka;    { točke }
  n: integer;                         { število točk }

{ Ali leži k na premici, ki jo določata i in j? }
function NaPremici(i, j, k: integer): boolean;
const eps = 1e-4;
var dx, dy, ux, uy, v, d2: real;
begin
  dx := t[j].x - t[i].x; dy := t[j].y - t[i].y;
  ux := t[k].x - t[i].x; uy := t[k].y - t[i].y;
  v := dx * uy - dy * ux; d2 := dx * dx + dy * dy;
  NaPremici := (d2 > 0) and (v * v <= eps * eps * d2);
```

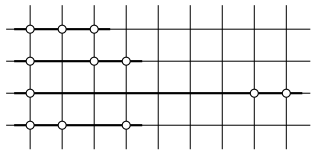
```
end; {NaPremici}

{ Doda v sezname podatek, da premica p pokriva točko it. }
procedure Dodaj(it: integer; p: PPremica);
var tp: PTockaNaPremici;
begin
  New(tp); tp↑.it := it; tp↑.p := p; p↑.n := p↑.n + 1;
  { Dodajmo jo v seznam točk, ki jih pokriva premica p. }
  tp↑.pt := nil; tp↑.nt := p↑.pt; p↑.pt := tp;
  if tp↑.nt <> nil then tp↑.nt↑.pt := tp;
  { Dodajmo jo v seznam premic, ki pokrivajo točko it. }
  tp↑.np := t[it].pp; t[it].pp := tp;
end; {Dodaj}

{ Zbriše točko it iz vseh premic, ki jo pokrivajo. }
procedure Brisi(it: integer);
var tp: PTockaNaPremici; p: PPremica; n: integer;
begin
  while t[it].pp <> nil do begin
    tp := t[it].pp; p := tp↑.p; t[it].pp := tp↑.np;
    { Zbrišimo točko it iz seznama točk, ki jih pokriva premica p. }
    if tp↑.pt = nil then p↑.pt := tp↑.nt else tp↑.pt↑.nt := tp↑.nt;
    if tp↑.nt <> nil then tp↑.nt↑.pt := tp↑.pt;
    Dispose(tp); n := p↑.n; p↑.n := p↑.n - 1;
    { Zbrišimo p iz seznama premic, ki pokrivajo n točk. }
    if p↑.pp = nil then Premice[n] := p↑.np else p↑.pp↑.np := p↑.np;
    if p↑.np <> nil then p↑.np↑.pp := p↑.pp;
    if n > 1 then begin { Dodajmo p v seznam premic, ki pokrivajo n - 1 točk. }
      p↑.pp := nil; p↑.np := Premice[n - 1];
      if p↑.np <> nil then p↑.np↑.pp := p;
      Premice[n - 1] := p;
    end; {if}
  end; {while}
end; {Brisi}

var i, j, k, pn, StPremic: integer; p: PPremica;
begin
  { Preberimo število točk in njihove koordinate. }
  Write('Število točk: '); ReadLn(n);
  for i := 1 to n do begin
    Write('X[', i, ']: '); ReadLn(t[i].x);
    Write('Y[', i, ']: '); ReadLn(t[i].y);
    t[i].pp := nil;
  end; {for}
  { Poiščimo vse premice. }
  for i := 1 to n do Premice[i] := nil;
```

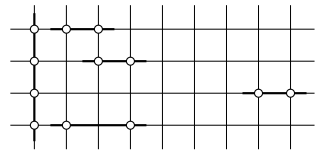
```
  for i := 1 to n − 1 do for j := i + 1 to n do begin
    New(p); p↑.n := 0; p↑.pt := 0; p↑.np := nil; p↑.pp := nil;
    { Poglejmo, katere točke pokriva premica skozi t[i] in t[j]. }
    for k := 1 to n do if NaPremici(i, j, k) then Dodaj(k, p);
    { Dodajmo premico v seznam premic, ki pokrivajo p↑.n točk. }
    if Premice[p↑.n] <> nil then Premice[p↑.n]↑.pp := p;
    p↑.np := Premice[p↑.n]; Premice[p↑.n] := p;
  end; {for}

  if n = 1 then begin { Imamo eno samo točko; pokrili jo bomo z eno premico. }
    New(p); p↑.n := 0; p↑.pt := nil; p↑.np := nil; p↑.pp := nil;
    Dodaj(1, p); Premice[p↑.n] := p;
  end; {if}

  { Požrešno izbirajmo premice. }
  pn := n; StPremic := 0;
  while pn > 0 do begin
    while Premice[pn] <> nil do begin
      { p je ena od premic, ki pokrivajo največ točk. }
      p := Premice[pn]; Write('Premica skozi');
      while p↑.pt <> nil do begin { Pobrišimo točke, ki jih p pokriva. }
        Write(' ', p↑.pt↑.it);
        Brisi(p↑.pt↑.it);
      end; {while}
      Dispose(p); WriteLn; StPremic := StPremic + 1;
    end; {while}
    { Premic, ki bi pokrivale pn točk, ni več; posvetimo se tistim, }
    pn := pn − 1;              { ki pokrivajo pn − 1 ali manj točk. }
  end; {while}

  WriteLn('Število uporabljenih premic: ', StPremic);
end. {Pokrivanje}
```

Nerodno pri tem postopku je, da hrani za vsako premico seznam vseh točk, ki jih ta premica pokriva. Ti seznami bi utegnili vsi skupaj požreti $O(n^3)$ prostora.[13] Pomnilniške zahteve bi lahko zmanjšali, če bi za vsako premico hranili samo podatek o tem, iz katerih dveh točk je bila pridobljena in koliko

[13] To se pravzaprav lahko zgodi le zaradi numeričnih nenatančnosti. Če so recimo točke A, B in C približno na isti premici, bi se lahko zgodilo, da pri opazovanju premice skozi A in C vidimo, da na njej leži tudi B, ko pa bi gledali premico skozi A in B, bi dobili občutek, da C ne leži na njej. Zato bi lahko eno in isto premico šteli po večkrat in pri tem opazili različne podmnožice točk, ki jih ta premica zares pokriva. Če takih numeričnih nenatančnosti ne bi bilo, pa je seveda jasno, da bi lahko vsaka točka pripadala največ $n - 1$ premicam, le pri naštevanju teh premic bi morali paziti, da ne bi iste premice šteli po večkrat. Lahko bi na primer naredili takole: če bi pri opazovanju točke T_i videli, da leži na premici skozi T_i in T_j tudi neka T_k, za katero je $k < i$, bi to premico ignorirali, saj bi vedeli, da smo jo morali že opaziti, ko smo gledali premice skozi T_k. Kakorkoli že, čim vemo, da pripada vsaka točka največ $n - 1$ premicam, vemo tudi, da bomo imeli največ $n(n - 1)$ zapisov TTockaNaPremici, tako da bo prostorska zahtevnost našega postopka le $O(n^2)$.

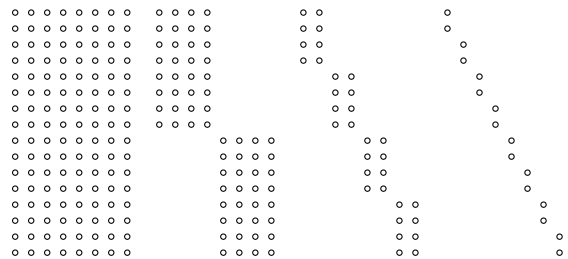
Imejmo m vrstic ($m > 3$) s po tremi točkami; ena naj bo vedno pri $x = 1$, drugi dve pa tako, da nobena poševna premica ne more pokriti več kot dveh točk naenkrat. Požrešni algoritem bi uporabil $m + 1$ premic namesto m premic.



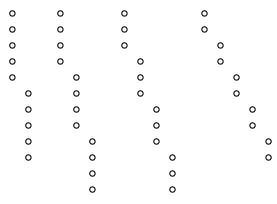


Imejmo pravokotnik točk (m vrstic, $m - 3$ stolpcev). Dodajmo v vsako vrstico še dve točki in to tako, da nobena poševna premica ne bo mogla pokriti več kot dveh takih točk. Vse skupaj lahko pokrijemo že z m vodoravnimi premicami, požrešni algoritem pa uporabi najprej $m - 3$ navpičnih premic za pravokotnik točk na levi in nato še m premic za ostale točke; skupaj torej $2m - 3$ namesto le m premic, kar je skoraj dvakrat preveč.

Skupino $n = 2^m \cdot (2^m - 1)$ točk je mogoče razporediti v 2^m vrstic s po $2^m - 1$ točkami tako, da bi požrešni algoritem našel rešitev z $m \cdot 2^{m-1}$ premicami, čeprav se da vse točke pokriti že s samo 2^m premicami. Na tej sliki je primer za $m = 4$, le stolpce bi bilo treba še tako razmakniti, da se ne bi dalo s poševnimi premicami pokriti po več kot dveh točk naenkrat. Razmerje med požrešno in optimalno rešitvijo je približno $\frac{1}{4} \lg n \approx 0{,}36 \ln n$.



Izberimo si števili h in m (na sliki je primer za $h = 10$ in $m = 5$). Točke postavljajmo najprej v stolpce po m, dokler ne dosežemo (ali presežemo) višine h. Potem jih podobno postavljajmo v stolpce po $m - 1$, $m - 2$, ..., 2. (Stolpce je treba še tako razmakniti, da poševne premice ne morejo pokriti več kot dveh točk naenkrat.) Požrešni algoritem uporablja navpične premice namesto vodoravnih in pri primernih h in m lahko porabi skoraj $(\frac{1}{2} \ln n)$-krat toliko premic kot optimalna rešitev (pri čemer je n število vseh točk); vendar pa se temu razmerju res približamo šele pri precej velikih n.



Ilustracija k rešitvi naloge 1989.3.3. Slike prikazujejo nekaj primerov, pri katerih požrešni algoritem ne najde najboljše rešitve.

točk trenutno pokriva. Ko si požrešni algoritem izbere neko premico in hoče pobrisati vse točke, ki jih ta pokriva, zdaj ne bi imel pri roki seznama teh točk, pač pa bi moral iti po vseh točkah in za vsako posebej preveriti, če jo izbrana premica pokriva ali ne. Podprogram Brisi pa tudi za dano točko ne bi imel na voljo seznama premic, ki pokrivajo to točko, in bi moral iti po vseh premicah ter pri vsaki pogledati, če to točko pokriva ali ne. Za brisanje točke lahko torej zdaj porabimo $O(n^2)$ časa, tako da je časovna zahtevnost celega algoritma $O(n^3)$, torej nič slabša kot pri gornjem programu. Prostorska zahtevnost pa je le $O(n^2)$, saj hranimo za vsako premico le konstantno mnogo podatkov.

Slike na str. 74 prikazujejo nekaj primerov, pri katerih požrešni algoritem ne vrne najboljše možne rešitve. Pri vsaki sliki je opisano, kako lahko take primere sestavimo in koliko je pri njih požrešna rešitev slabša od najboljše možne (kolikokrat več premic porabi, da pokrije vse točke).

R1989.3.4

N: 48

Preden se lotimo dela z izrazom, ga je koristno iz niza predelati v neko bolj standardno obliko, v kateri se nam ne bo treba več ukvarjati s presledki med operatorji in s tem, ali neka številka vsebuje decimalno piko ali ne. Predelali ga bomo v zaporedje „osnovnih simbolov“ ali „žetonov“ (*tokens*). To lahko storimo tako, da beremo vhodni niz znak za znakom; operatorji +, -, * in /, pa tudi oklepaji in zaklepaji, naj bodo že sami zase osnovni simboli, poleg tega pa imejmo še eno vrsto osnovnega simbola, ki bo predstavljal številske konstante (Num v spodnjem programu) in enega za konec izraza (EolT v spodnjem programu). Osnovni simbol bo predstavljen z zapisom TokenT, ki vsebuje tip simbola (t — zanj bi lahko uvedli poseben naštevni tip, še lažje pa bo, če uporabimo kar tip char), pri številskih konstantah pa tudi vrednost (polje x).

Pri branju izraza bo koristno, če bomo lahko naslednji znak le pogledali, ne pa ga tudi res prebrali (oz. če bomo lahko nek znak prebrali dvakrat). Lahko bi prebrali cel izraz v nek niz in se potem sprehajali po njem naprej in nazaj, čisto dobro pa je tudi, če v neki spremenljivki hranimo že prebrani znak, ki je bil vrnjen v zaporedje, da ga bomo prebrali še enkrat. Spodnji program to doseže s spremenljivko PutBack (ki ima vrednost Chr(0), če je „prazna“) in podprogramom GetCh, ki uporabi vrednost iz PutBack, če pa je ta prazna, prebere naslednji znak iz datoteke.

Za branje naslednjega osnovnega simbola imamo podprogram NextToken. Ta najprej preskoči presledke, če jih je kaj; nato, če naleti na enoznakovne simbole (operatorje, oklepaje) ali konec niza, vrne to kot en simbol; številko pa pretvori v tip real in jo vrne kot osnovni simbol tipa Num.

Če bi takole predelali izraz v zaporedje osnovnih simbolov, bi se lahko lotili računanja čisto po zdravi pameti. Pri vrstnem redu računanja moramo upoštevati oklepaje in prioriteto operatorjev. Lahko bi se torej za začetek sprehodili po izrazu in šteli, koliko oklepajev je trenutno odprtih ter kje se

začnejo. Ko bi naleteli na zaklepaj, bi del izraza od zadnjega oklepaja do tega zaklepaja izračunali z rekurzivnim klicem istega podprograma in nato v zaporedju osnovnih simbolov ves ta del izraza nadomestili z enim samim simbolom tipa Num, ki bi imel za vrednost kar vrednost tistega dela izraza. Ko bi se na ta način počasi znebili vseh oklepajev, bi lahko v mislih „razrezali“ izraz pri vsakem operatorju + ali - (razen pri - na začetku zaporedja ali tik za simboloma * ali /, kajti tak - je unaren). Vsak od nastalih podizrazov je sestavljen iz števil, ki so ločena z operatorjema * in /, tik pred kakšnim številom pa je lahko tudi unarni minus. Take z minusom pomnožimo z -1 in se minusa znebimo. Vrednost podizraza lahko zdaj izračunamo tako, da začnemo s prvim številom in ga nato z vsakim naslednjim številom pomnožimo ali pa delimo, odvisno od operatorja pred tem številom. Ko imamo vrednost vsakega podizraza, izračunamo vrednost celega izraza spet tako, da začnemo z vrednostjo prvega in ji nato prištevamo ali odštevamo vrednosti naslednjih, spet odvisno od tega, ali je pred nekim podizrazom + ali -.

Lahko pa smo še za odtenek bolj elegantni in vse skupaj združimo v en sam prehod skozi vhodne podatke. V ta namen imamo spodaj podprograme EvalExpr (za izračun celega izraza), EvalTerm (za izračun podizraza, dobljenega z množenjem in deljenjem) in EvalAtom (za izračun izraza v oklepajih, izraza z unarnim minusom ali pa izraza, ki je kar številska konstanta). Vsi pričakujejo kot parameter prvi osnovni simbol svojega dela izraza, ob koncu izvajanja pa vrnejo v njem prvi osnovni simbol za svojim delom izraza, kar bo potreboval klicatelj za nadaljevanje dela. Vrednosti izrazov računajo sproti. Na primer, EvalExpr pokliče EvalTerm, da bi dobil vrednost prvega podizraza. Če za tem podizrazom pride kaj drugega kot + ali -, lahko kar končamo (če je bil cel izraz pravilno zgrajen, lahko tam razen + ali - tako ali tako nastopa le še zaklepaj ali pa konec izraza), sicer pa preberemo naslednji izraz in njegovo vrednost prištejemo ali odštejemo ter tako nadaljujemo. Podobno dela EvalTerm, ki za vsak faktor v svojem podizrazu kliče EvalAtom.

```
function Eval(var f: text): real;

  const EOL = Chr(13);
  var PutBack: char;

  function GetCh: char;
  begin
    if PutBack = Chr(0) then begin
      if Eoln(f) then begin PutBack := EOL; ReadLn(f) end
      else Read(f, PutBack);
    end; {if}
    GetCh := PutBack; PutBack := Chr(0);
  end; {GetCh}

  const EolT = 'e'; Num = 'n';
```

```
type TokenT = record t: char; x: real end;

procedure NextToken(var t: TokenT);
var c: char; u: real;
begin
  { Preskočimo presledke. }
  repeat c := GetCh until not (c in [' ', Chr(9)]);
  if c = EOL then { Prišli smo do konca izraza. }
    begin t.t := EolT; PutBack := c end
  else if c in ['+', '-', '/', '*', '(', ')'] then
    t.t := c { Operator ali oklepaj, ki je že sam zase osnovni simbol. }
  else if c in ['0'..'9', '.'] then begin
    { Prebrati moramo število. }
    t.t := Num; t.x := 0;
    while c in ['0'..'9'] do { Števke pred decimalno piko. }
      begin t.x := t.x * 10 + (Ord(c) - Ord('0')); c := GetCh end;
    if c = '.' then begin
      c := GetCh; u := 1; { Decimalna pika in mogoče še števke za njo. }
      while c in ['0'..'9'] do begin
        u := u * 0.1; t.x := t.x + u * (Ord(c) - Ord('0')); c := GetCh;
      end; {while}
    end; {if}
    PutBack := c; { Ta znak že ne spada več k našemu številu. }
  end else
    begin WriteLn('Nedovoljen znak: #', Ord(c)); t.t := EolT end;
end; {NextToken}

function EvalExpr(var t: TokenT): real; forward;

function EvalAtom(var t: TokenT): real;
begin
  if t.t = '(' then begin
    NextToken(t); EvalAtom := EvalExpr(t);
    if t.t <> ')' then WriteLn('Pričakoval bi zaklepaj, ne pa ', t.t, '.');
    NextToken(t);
  end else if t.t = Num then begin
    EvalAtom := t.x; NextToken(t);
  end else if t.t = '-' then begin
    NextToken(t); EvalAtom := -EvalAtom(t);
  end else WriteLn('Pričakoval bi oklepaj, unarni minus ',
                   'ali številko, ne pa ', t.t, '.');
end; {EvalAtom}

function EvalTerm(var t: TokenT): real;
var x, y: real; op: char;
begin
  x := EvalAtom(t);
```

```
    while t.t in ['*', '/'] do begin
      op := t.t; NextToken(t); y := EvalAtom(t);
      if op = '*' then x := x * y else x := x / y;
    end; {while}
    if not (t.t in ['+', '-', ')', EolT]) then WriteLn('Pričakoval bi ',
      'zaklepaj, + , - ali konec izraza, ne pa ', t.t, '.');
    EvalTerm := x;
  end; {EvalTerm}

  function EvalExpr(var t: TokenT): real;
  var x, y: real; op: char;
  begin
    x := EvalTerm(t);
    while t.t in ['+', '-'] do begin
      op := t.t; NextToken(t); y := EvalTerm(t);
      if op = '+' then x := x + y else x := x - y;
    end; {while}
    if not (t.t in [')', EolT]) then
      WriteLn('Pričakoval bi zaklepaj ali konec izraza, ne pa ', t.t, '.');
    EvalExpr := x;
  end; {EvalTerm}

var t: TokenT; x: real;
begin {Eval}
  PutBack := Chr(0); NextToken(t);
  if t.t = EolT then begin WriteLn('Datoteka je prazna!'); Eval := 0 end
  else begin
    x := EvalExpr(t); Eval := x;
    if t.t <> EolT then WriteLn('Izraz ni pravilne oblike! ',
                                'Zatakne se pri ', t.t, '.');
    WriteLn('Vrednost izraza: ', x:10:5);
  end; {if}
end; {Eval}
```

Za tip osnovnega simbola, ki predstavlja konec izraza, bi lahko namesto EolT uporabili tudi kar EOL, a potem bi bila sporočila o napakah, če nepričakovano zgodaj naletimo na konec izraza, videti bolj čudno (za „`ne pa`“ ne bi bilo videti ničesar, pač pa bi se pika izpisala na začetku vrstice).

14. republiško tekmovanje v znanju računalništva (1990)

NALOGE ZA PRVO SKUPINO

1990.1.1

R: 84

Ko se računalnik vključi v komunikacijsko mrežo, ne pozna drugih računalnikov, ki so vključeni vanjo. Vsak računalnik v mreži ima svojo številko in ime. Računalnikov je največ MaxRac. **Napiši podprogram**, ki bo izpisal imena vseh računalnikov v mreži, vendar vsakega samo enkrat.

Na voljo imaš naslednja dva podprograma:

KdoSem — funkcija, ki vrne številko našega računalnika;

Vprašaj(Naslov, Ime, Sosedi, SosediL) — vpraša računalnik Naslov za njegovo ime in spisek njegovih sosedov. Ime računalnika dobimo v parametru Ime (8 znakov), število sosedov dobimo v SosediL, spisek številk sosedov pa v tabeli Sosedi. Največje mogoče število sosedov je MaxSosedi.

Opomba: številke računalnikov niso nujno zaporedna naravna števila; nekatera števila ne ustrezajo nobenemu računalniku. Podprogramu Vprasaj je dovoljeno podati le veljavno številko nekega (že znanega) računalnika. Za ime in spisek sosedov lahko seveda vprašaš tudi samega sebe.

1990.1.2

R: 86

V tabeli imamo podatke o imenih in letu rojstva množice oseb. Podatki so urejeni po imenih oseb v abecednem vrstnem redu. Vsa leta rojstva so med (vključno) 1880 in 1990. Podatke želimo urediti po letu rojstva, pri čemer želimo pri istem letu rojstva ohraniti urejenost po abecedi.

Opiši postopek, ki podatke iz podane tabele prepiše v drugo tabelo tako, da bodo urejeni po želenem kriteriju. Velikost dodatnih spremenljivk naj bo neodvisna od števila podatkov. Napiši rešitev, ki bo čim manjkrat pregledala posamezno osebo.

Namig: Ker je možnih let rojstva malo, lahko prešteješ, koliko oseb je rojenih v posameznem letu.

1990.1.3

R: 87

Podjetje „Zaphodove nore naprave“ načrtuje grafično kartico, ki naj bi se ponašala z zelo hitrim risanjem. Zato ima kartica na vsako točko rastrske slike (pixel) povezan svoj procesor. Vsak procesor izvaja svojo kopijo istega programa. Povezan je s svojimi štirimi sosedi. Vezi so oštevilčene od 1 do 4 v smeri urinega kazalca, začenši z zgornjo.

Barve so predstavljene s celimi števili. Neko ploskev na zaslonu, ki je omejena s točkami podane barve MejnaBarva (konstanta), želimo pobarvati s pravokotnim ponavljajočim se vzorcem velikosti XMax × YMax. Točka je znotraj ploskve, če ni mejne barve in je znotraj ploskve vsaj ena soseda.

Definiraj vsebino sporočila, ki si ga bodo procesorji pošiljali. Nato **napiši postopek**, ki bo tekel na vseh procesorjih in bo na zaslonu pobarval ploskev, omejeno z mejno barvo. Predpostaviš lahko, da bo sporočilo pravega formata na magičen način prišlo po zvezi v nek procesor znotraj ploskve.

Na voljo imaš naslednje podprograme:

JeSporocilo vrne številko vezi, kjer čaka sporočilo, ali 0, če ni nobenega sporočila;

Sprejmi(Zveza) vrne sporočilo iz zveze Zveza; dokler sporočila ni, čaka;

Poslji(Zveza, Sporocilo) pošlje sporočilo po zvezi Zveza;

MojaBarva vrne trenutno barvo točke, ki jo upravlja procesor;

PobarvajMe(Barva) pobarva točko, ki jo upravlja procesor, z barvo Barva;

Vzorec(x, y) vrne barvo, ki je na mestu (x, y) v vzorcu; za koordinate, ki so izven vzorca (če torej ne velja $0 \le x < XMax$ in $0 \le y < YMax$), vrednost ni definirana.

R: 88

1990.1.4

Kaj izpiše naslednji program? Namesto pisanja števil lahko napišeš tudi kratek program, ki da enake rezultate.

```
program Marvin(Output);
const
  Els = 128;
var
  a: array [1..Els] of integer;
  b: array [1..Els] of integer;
  c, d, e: integer;
begin {Marvin}
  c := 1; b[Els] := 0;
  for d := 1 to Els do begin a[d] := d; if d > 1 then b[d - 1] := d end;
  d := 0;
  while c <> 0 do begin e := b[c]; b[c] := d; d := c; c := e end;
  while d <> 0 do begin WriteLn(a[d]); e := b[d]; b[d] := c; c := d; d := e end;
end. {Marvin}
```

NALOGE ZA DRUGO SKUPINO

1990.2.1

Kljub temu, da je Marvin hiperinteligentni robot, mu tokrat ne preostane drugega, kot da godrnjaje naredi preprosto inventuro skladišča transgalaktičnega podjetja A & F. Skladišče je zgrajeno kot n-dimenzionalni kvader (n je med 1 in MaxN). Podatki, ki jih Marvin dobi pri vhodu v skladišče, so število dimenzij skladišča (n) in dolžina skladišča Dolz v vsaki od dimenzij (Dolz je tabela velikosti n). **Napiši program**, ki bo Marvinu izpisal seznam koordinat osnovnih celic, ki jih mora pregledati (vsako celico natanko enkrat). R: 89

1990.2.2

Naj bo (a) strogo naraščajoče zaporedje realnih števil $a_1 < a_2 < \ldots < a_n$. Tedaj element $a_{(n+1)\,\mathrm{div}\,2}$ imenujemo *srednji element zaporedja* (a). Tako je na primer v zaporedju -35, 2, 7, 14, 15 srednji element 7, v zaporedju 1, 3,14, 18, 19,5 pa je srednji element 3,14. R: 91

Naj bosta (a) in (b) dolgi, strogo naraščajoči zaporedji realnih števil, vsako s po n elementi, pri čemer je vsak element zaporedja (a) različen od vseh elementov zaporedja (b). **Opiši postopek**, ki dobi zaporedji (a) in (b) v tabelah in ki brez dodatne tabele kar se da hitro določi srednji element strogo naraščajočega zaporedja, ki ga skupaj tvorijo elementi zaporedij (a) in (b).

1990.2.3

Sto procesorjev (vsak izvaja svoj proces) je povezanih med seboj v obroč tako, da ima vsak stik le s svojim levim in desnim sosedom. Vsak procesor izvaja svojo kopijo naslednjega programa: R: 93

```
program Gorilnik;

var
  Gori: boolean;      { gorilnik gori }
  Gorivo: integer;    { količina goriva v gorilniku }
  Temper: integer;    { temperatura gorilnika }

  function Netilec: boolean; external;
  function LeviGori: boolean; external;
  function DesniGori: boolean; external;
  procedure Cakaj; external;

begin {Gorilnik}
  Gori := false; Gorivo := 100; Temper := 0;
  if Netilec then { natanko en procesor je „netilec“ }
    begin Gori := true; Temper := 100 end;
  repeat
    while (Temper < 80) or (Gorivo < 50) do begin      { ne gori }
      if LeviGori or DesniGori then                    { segrevanje od sosedov }
```

```
        begin Temper := Temper + 11; if Temper > 100 then Temper := 100 end;
      if Temper > 0 then Temper := Temper − 1;        { ohlajanje }
      if Gorivo < 100 then Gorivo := Gorivo + 1;      { dotok goriva }
      Cakaj;
    end; {while}
    Gori := true; Temper := 100;                      { vžig }
    while Gorivo >= 10 do                             { gorenje }
      begin Gorivo := Gorivo − 10; Cakaj end;
    Gori := false; Gorivo := 0;                       { ugasnitev }
  until false;
end. {Gorilnik}
```

Podprogram Cakaj zadrži izvajanje programa za 0,1 sekunde, sicer pa lahko predpostavimo, da je izvajanje preostalega programa zelo hitro. Vsi procesorji pričnejo izvajati svoj program hkrati. Funkciji LeviGori in DesniGori vrneta ob klicu stanje spremenljivke Gori v levem oziroma desnem procesu. Funkcija Netilec vrne true le enemu procesorju, vsem ostalim vrne false.

Če opazujemo stanje spremenljivke Gori v vsakem procesu v obroču, lahko opazimo določeno podmnožico procesov, v katerih velja Gori = true.

Opiši, kako se podmnožica „gorečih“ procesorjev spreminja s časom (kako se „plamen“ seli). Ali se ta podmnožica sčasoma ustali ali je spreminjanje stalno? Odgovore **utemelji**.

R: 94

1990.2.4

Hkrati poženemo dva programa, ki tečeta vsak na svojem procesorju. Oba imata dostop do iste globalne spremenljivke, ki je bila pred tem nastavljena na vrednost nič. Med izvajanjem vsak program natanko desettisočkrat prebere vrednost spremenljivke, prišteje ena in novo vrednost zapiše nazaj, ne da bi vedel, kaj počne drugi program. Upoštevaj, da je hitrost izvajanja programov lahko močno neenakomerna.

Kolikšni sta najmanjša in največja vrednost, ki ju spremenljivka lahko zavzame, ko oba programa končata z delom? Ali lahko zavzame poljubno vrednost med najmanjšo in največjo? Kaj pa, če imamo n programov, kjer je n poljubno število med vključno 1 in 42? Odgovor **utemelji**.

NALOGE ZA TRETJO SKUPINO

R: 95

1990.3.1

V vrsto želimo postaviti n domin. Vsaka domina ima dve polji. Vsako polje je označeno z 1 ali 2 ali ... m pikami. **Napiši program**, ki za poljubno dano množico domin ugotovi, ali lahko vse domine zložimo v vrsto. Pri zlaganju v vrsto morata imeti soležni polji sosednjih domin enako število pik. Podatki za program so število domin in število pik na vsakem polju vsake od njih.

1990.3.2

R: 97

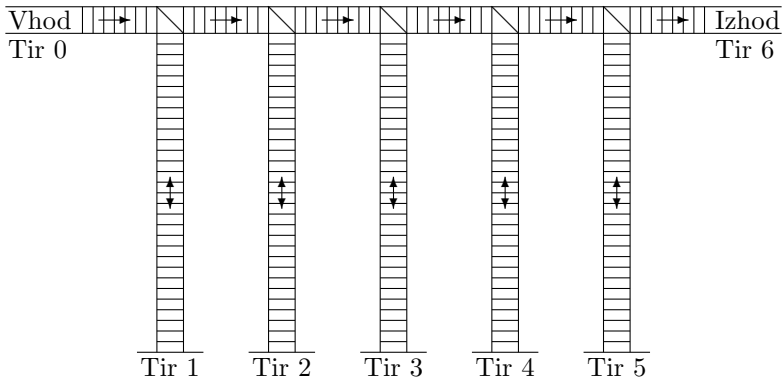
Železniška postaja je sestavljena iz 5 vzporednih slepih tirov. Na vhodu stoji 32 vagonov, oštevilčenih s številkami 1, 2, ..., 32, ki pa so med seboj premešani. Železničarji lahko premikajo po en vagon v smeri od vhoda proti izhodu. Vagon lahko torej premaknejo z vhoda postaje ali s poljubnega slepega tira na poljubni (kasnejši) slepi tir ali na izhod postaje. **Napiši program**, ki bo premikal vagone tako, da bodo na izhodni tir prihajali urejeni po naraščajočih številkah.

Na voljo imaš naslednje podprograme:

PremakniVagon(OdKod, Kam) premakne en vagon s tira OdKod na tir Kam. OdKod in Kam sta številki tirov, kjer je $0 \leq \mathsf{OdKod} < \mathsf{Kam} \leq 6$.

Prazen(Tir) vrne true, če je tir z oznako Tir prazen ($0 \leq \mathsf{Tir} \leq 6$).

Vagon(Tir) vrne številko vagona, ki je prvi na tiru Tir ($0 \leq \mathsf{Tir} \leq 6$); to je številka tistega vagona s tira Tir, ki ga je mogoče premakniti.



1990.3.3

R: 103

Med dvema računalnikoma, povezanima v komunikacijsko mrežo, potujejo datoteke. Vsaka datoteka je sestavljena iz več zapisov (vrstic), ki lahko potujejo kot paketi od pošiljatelja do prejemnika po različnih poteh, odvisno od obremenitve mreže. Paketi vsebujejo poleg zapisa z datoteke tudi identifikacijo datoteke, kateri pripadajo, zaporedno številko zapisa v datoteki in oznako, ali gre za zadnji zapis te datoteke. Istočasno lahko prihaja več datotek, katerih paketi se lahko med prenosom po različnih poteh različno zakasnijo in zato prihajajo v poljubnem vrstnem redu. Prav lahko se zgodi, da prispe zadnji zapis neke datoteke pred prvim, zato jih mora računalnik, ki jih sprejema, urediti po vrsti, preden jih dokončno shrani.

Napiši tisti del programa, ki prispele zapise izpiše v pravilnem vrstnem redu. V vsakem trenutku lahko hkrati prihaja največ 10 datotek, povprečna dolžina datoteke je 100 zapisov. Oblika prispelih paketov je takšna:

```
type PaketT = record
        VrstaPaketa: (Vmesni, Zadnji);   { vrsta prihajajočega paketa }
        Stevilka: integer;               { zaporedna številka paketa }
        Datoteka: 1..10;                 { številka datoteke, ki ji paket pripada }
        Zapis: ZapisT;                   { zapis v paketu }
      end; {PaketT}
```

Če je VrstaPaketa enaka Zadnji, potem nam njegova številka pove, koliko zapisov je v datoteki. Oblika tipa ZapisT ni pomembna. Na voljo imaš podprogram Pisi(Datoteka, Zapis), ki izpiše zapis iz enega paketa sporočila.

R: 105

1990.3.4

Marvin in Garvin sta nesrečna in malodušna robota. Raztovoriti morata vesoljsko ladjo, polno pangalaktičnega grloreza. Če se oba robota znajdeta hkrati v ladji, zavlada takó pesimistično vzdušje, da nadaljnje delo ni več mogoče, zato za medsebojno sinhronizacijo uporabljata spodaj podani algoritem. Na začetku sta oba robota izven vesoljske ladje. **Ali je možno, da se znajdeta oba robota hkrati v vesoljski ladji?** Odgovor utemelji!

Algoritem za robota Jaz (Jaz je konstanta z vrednostjo 1 ali 2):

```
const
  Jaz = ...;
var
  vLadji: 0..2;   { Ti dve spremenljivki si delita oba robota. Pomnilnik je organiziran }
  naVrsti: 1..2;  { tako, da lahko bere, piše ali testira vrednost ene spremenljivke    }
                  { le en robot naenkrat. Pred začetkom izvajanja programov je          }
                  { vrednost vLadji 0, naVrsti pa 1.                                    }

procedure Raztovarjanje;
begin
  repeat { zanka 0 }
    repeat { zanka 1 }
      repeat until (vLadji = 0) or (vLadji = Jaz); { zanka 2}
      naVrsti := Jaz;
      if vLadji = 0 then vLadji := Jaz;
    until (naVrsti = Jaz) and (vLadji = Jaz);
    { robot Jaz vstopi v ladjo, poišče zaboj grloreza, ga vzame in izstopi iz ladje }
    vLadji := 0;
  until LadjaPrazna;
end; {Raztovarjanje}
```

REŠITVE NALOG ZA PRVO SKUPINO

N: 79

R1990.1.1

Spodnji podprogram hrani v tabeli Spisek seznam vseh računalnikov, ki jih je doslej odkril. Sprehaja se po tem

spisku in vsak računalnik povpraša o sosedih; za dobljene sosede pogleda, če so že v spisku; če niso, jih doda. Tako bodo tudi na novo odkriti sosedje prej ali slej prišli na vrsto, da bomo tudi njih povprašali o njihovih sosedih in tako naprej.

```
const
  MaxRac = 1000;                    { največje število računalnikov v mreži }
  MaxSosedi = 20;                   { največje število povezav }
type ImeT = packed array [1..8] of char;
     SosediT = array [1..MaxSosedi] of integer;
var
  SpisekL: integer;                 { število računalnikov na spisku }
  Spisek: array [1..MaxRac] of record   { spisek računalnikov v mreži }
            Stevilka: integer;      { številka računalnika — naslov }
            Ime: ImeT;              { ime računalnika }
          end;

  function KdoSem: integer; external;
  procedure Vprasaj(Naslov: integer; var Ime: ImeT;
                    var Sosedi: SosediT; var SosediL: integer); external;

procedure NajdiVozlisca;
var Sosedi: SosediT;
    p, st, j, SosediL: integer;
begin
  { V spisek najprej vstavimo sebe. }
  SpisekL := 1; Spisek[SpisekL].Stevilka := KdoSem; St := 1;
  while st <= SpisekL do begin
    { Obdelamo vse sosede računalnika st. }
    Vprasaj(Spisek[st].Stevilka, Spisek[st].Ime, Sosedi, SosediL);
    for p := 1 to SosediL do begin
      { Za vsak računalnik iz Sosedi[st] pogledamo, ali je že na spisku. }
      j := 1; Spisek[SpisekL + 1].Stevilka := Sosedi[p];
      while Sosedi[p] <> Spisek[j].Stevilka do j := j + 1;
      if j > SpisekL then SpisekL := SpisekL + 1;
    end; {for}
    WriteLn(Spisek[st].Stevilka, ' ', Spisek[st].Ime);
    st := st + 1;
  end; {while}
end; {NajdiVozlisca}
```

Namesto seznama Spisek bi imeli lahko tudi razpršeno tabelo, tako da ne bi potrebovali notranje zanke **while**, pač pa bi le preverili, če je trenutni sosed že v razpršeni tabeli. To bi bilo hitreje, vendar bi pri našem problemu večino časa najbrž tako ali tako porabili za komuniciranje z drugimi računalniki v mreži.

N: 79

R1990.1.2

Ker je možnih let rojstva malo, bomo za vsako leto prešteli, koliko ljudi se je takrat rodilo. Spodnji podprogram hrani to število v so[Leto]. Zdaj vemo, da bodo v urejeni tabeli ljudje, rojeni leta 1880, pristali na indeksih od 1 do so[1880]; tisti, rojeni leta 1881, na indeksih od so[1880] + 1 do so[1880] + so[1881] in tako naprej. Tako lahko za vsako letnico izračunamo, na katerem indeksu se bodo v urejeni tabeli začeli podatki o ljudeh, rojenih tisto leto. Za te indekse lahko spet uporabimo tabelo so, ker podatkov o številu oseb, rojenih posamezno leto, kasneje ne bomo več potrebovali. Zdaj lahko opravimo drugi prehod po seznamu oseb; ko naletimo na človeka, rojenega leta Leto, ga vpišemo na indeks so[Leto] v izhodni tabeli b, vrednost so[Leto] pa povečamo za 1. Tako nam tabela so zdaj pravzaprav pove, kam vpisati naslednjo osebo, rojeno v posameznem letu. Pomembno pri tem načinu prerazporejanja oseb je, da se medsebojni vrstni red oseb, rojenih isto leto, ne bo spremenil — če je bil nekdo v tabeli a pred nekom drugim, rojenim isto leto, bo v tabeli b tudi.

Opisani postopek urejanja podatkov se imenuje „urejanje s štetjem (oz. preštevanjem)“ (*counting sort*). Uporabimo ga lahko v primerih, ko je možnih vrednosti ključa, po katerem urejamo, dovolj malo, da si lahko privoščimo za vsako možno vrednost ključa vzdrževati podatek o številu pojavitev te vrednosti v vhodnem zaporedju. (Pri naši nalogi je možnih le 111 ključev — letnice od 1880 do 1990.) Dobro je tudi, če je možnih vrednosti ključa razmeroma malo v primerjavi s številom elementov, ki bi jih radi uredili; če ni tako, bi znal biti kateri od splošnonamenskih postopkov urejanja vendarle učinkovitejši.

Lastnost, ki jo zahteva naša naloga, torej da postopek za urejanje ne spremeni medsebojnega vrstnega reda elementov z isto vrednostjo ključa (v našem primeru: ljudi, rojenih v istem letu), se imenuje *stabilnost*. Od znanih postopkov za urejanje so nekateri stabilni (npr. urejanje z mehurčki, z vstavljanjem, z izbiranjem in z zlivanjem), nekateri pa ne (npr. Shellovo urejanje, urejanje s kopico in običajna implementacija quicksorta).

```
const
  LetoMin = 1880;      { prvo možno leto rojstva }
  LetoMax = 1990;      { zadnje možno leto rojstva }
  MaxN = 13000;        { maksimalno število oseb }
type
  OsebaT = record
             Ime: packed array [1..32] of char; { podatki o osebi }
             Leto: LetoMin..LetoMax;
           end; {OsebaT}
  TabelaT = array [1..MaxN] of OsebaT;

procedure Uredi(var a, b: TabelaT; n: integer);
{ Dejansko število oseb, katerih podatke dobimo v tabeli a, je n.
  Urejene podatke o osebah vrnemo v tabeli b. }
```

```
var
  so: array [LetoMin..LetoMax] of integer; { št. oseb, rojenih v posameznem letu }
  i, j, k: integer;                        { števci }
begin
  { V tabeli so preštejemo, koliko oseb se je rodilo v posameznem letu. }
  for i := LetoMin to LetoMax do so[i] := 0; { začetne vrednosti }
  for i := 1 to n do so[a[i].Leto] := so[a[i].Leto] + 1; { štetje }
  { Podatke v tabeli preračunamo tako, da nam povedo, kje v pravilno
    urejeni tabeli b se začno osebe z danim letom rojstva. }
  j := so[LetoMin]; so[LetoMin] := 1;
  for i := LetoMin + 1 to LetoMax do
    begin k := so[i]; so[i] := so[i - 1] + j; j := k end;
  { Podatke o osebah prepišemo iz tabele a v tabelo b; upoštevamo, da nam
    so[r] pove, na katero mesto v tabeli b pride naslednja oseba z letom rojstva r. }
  for i := 1 to n do
    begin b[so[a[i].Leto]] := a[i]; so[a[i].Leto] := so[a[i].Leto] + 1 end;
end; {Uredi}
```

R1990.1.3

N: 79

Sporočila, ki si jih procesorji pošiljajo, naj bodo kar koordinate prejemnega procesorja. Ko torej nek procesor dobi sporočilo, lahko iz koordinat izračuna, kateri točki vzorca ustreza njegov položaj, tako da ve, s kakšno barvo se mora pobarvati. Nato še obvesti svoje sosede, razen seveda tistega, od katerega je sam dobil sporočilo. Poseben primer so procesorji, ki nadzirajo točke mejne barve — oni sporočil ne širijo, tako da se barvanje ustavi ob mejni barvi. Procesor lahko po tistem, ko se je že pobarval in obvestil sosede, prejme še več izvodov istega sporočila od raznih svojih sosedov, saj le-ti ne morejo vedeti, ali je bil že obveščen ali ne; taka odvečna sporočila lahko kar zavržemo.

```
procedure Pobarvaj;
var
  Zveza: integer;          { od kod smo dobili sporočilo }
  MojX, MojY: integer;     { kje v rastru smo }
  NovaBarva: integer;      { ustrezna barva iz vzorca }
begin
  if MojaBarva <> MejnaBarva then begin
      repeat Zveza := JeSporocilo until Zveza <> 0;
    repeat Zveza := JeSporocilo until Zveza <> 0; { čakamo na sporočilo }
    MojX := Sprejmi(Zveza);                       { sprejmemo svoje koordinate }
    MojY := Sprejmi(Zveza);
    NovaBarva := Vzorec(MojX mod XMax, MojY mod YMax);
    if NovaBarva <> MojaBarva then
      PobarvajMe(NovaBarva);                      { pobarvamo se }
    { Obvestimo sosede, razen tistega, ki je obvestil nas. }
    if Zveza <> 1 then begin Poslji(1, MojX); Poslji(1, MojY - 1) end;
    if Zveza <> 2 then begin Poslji(2, MojX + 1); Poslji(2, MojY) end;
```

```
    if Zveza <> 3 then begin Poslji(3, MojX); Poslji(3, MojY + 1) end;
    if Zveza <> 4 then begin Poslji(4, MojX − 1); Poslji(4, MojY) end;
  end; {if}
  repeat { počakamo, če nam bo še kdo ukazal, naj se pobarvamo }
    Zveza := JeSporocilo;
    if Zveza <> 0 then Zveza := Sprejmi(Zveza);  { zavržemo sporočilo }
  until false;
end; {Pobarvaj}
```

N: 80

R1990.1.4

Program izpiše tabelo a v obratnem vrstnem redu, torej vsa števila od 128 do 1.

Kratek program, ki da enak izpis, je tak:

```
program Marvin(Output);
const Els = 128;
var d: integer;
begin
  for d := Els downto 1 do WriteLn(d);
end. {Marvin}
```

Tabeli a in b uporablja kot seznam, v katerem je a[i] podatek in b[i] indeks naslednjega elementa seznama (kazalec na naslednji element). Del programa

```
  c := 1; b[Els] := 0;
  for d := 1 to Els do begin a[d] := d; if d > 1 then b[d − 1] := d end;
```

definira tabelo a in zgradi seznam. V danem primeru je naslednik i-tega elementa i + 1. Zadnji element ima indeks 0. Spremenljivka c vsebuje indeks prvega elementa seznama. Naslednja zanka v programu,

```
  d := 0;
  while c <> 0 do begin e := b[c]; b[c] := d; d := c; c := e end;
```

se pomika po tabeli indeksov do konca seznama. Pri tem obrne indekse v seznamu tako, da kažejo v nasprotno smer (v našem primeru i-ti element kaže na element i − 1). Ob začetku vsake ponovitve te zanke je c indeks trenutnega elementa, d pa indeks prejšnjega. (V e si začasno zapomnimo indeks naslednjega.) Po izhodu iz zanke spremenljivka d vsebuje indeks zadnjega elementa v seznamu. Zadnja zanka

```
  while d <> 0 do begin WriteLn(a[d]); e := b[d]; b[d] := c; c := d; d := e end;
```

izpisuje podatkovne elemente seznama, popravlja indekse na začetno stanje in se premika proti začetku seznama. Ob začetku vsake ponovitve te zanke kaže d na trenutni element, c na naslednjega, v e pa si začasno zapomnimo indeks prejšnjega. Po izhodu iz zanke je tabela indeksov b enaka kot po zgraditvi seznama. Spremenljivka c zopet kaže na prvi element seznama.

REŠITVE NALOG ZA DRUGO SKUPINO

R1990.2.1

N: 81

Če imamo n dimenzij in dolžine stranic od d_1 do d_n, lahko celice oštevilčimo od 0 do $m-1$ za $m = d_1 \cdot d_2 \cdot \ldots d_n$. Potem lahko preprosto naštejemo vsa ta števila in iz vsakega izračunamo koordinate celice, ki ji to število pripada. Dvorazsežni kvader bi lahko oštevilčili tako: celica s koordinatama (a_1, a_2) dobi indeks $i(a_1, a_2) = a_1 + d_1 a_2$. (Dogovorimo se, da bomo koordinate šteli od 0 do $d_i - 1$, ne od 1 do d_i.) Iz številke i lahko izračunamo koordinati po formuli $a_1 = i \bmod d_1$, $a_2 = i \operatorname{div} d_2$. V treh dimenzijah bi vzeli preslikavo $i(a_1, a_2, a_3) = a_1 + d_1 a_2 + d_1 d_2 a_3$ in obratno preslikavo $a_1 = i \bmod d_1$, $a_2 = (i \operatorname{div} d_1) \bmod d_2$, $a_3 = (i \operatorname{div} d_1) \operatorname{div} d_2$. Tako lahko nadaljujemo s poljubno mnogo dimenzijami:

$$i(a_1, \ldots, a_n) = \textstyle\sum_{j=1}^{n} a_j \prod_{k=1}^{j-1} d_j$$
$$\text{in obratno } a_j = (i \operatorname{div} \textstyle\prod_{k=1}^{j-1} d_j) \bmod d_j.$$

```
program MarvinovVodic(Input, Output);
const MaxN = 10;      { največja dimenzija kvadra }
type DimenzijeT = array [1..MaxN] of integer;
var
  n: integer;         { dimenzija kvadra }
  Dolz: DimenzijeT;   { dimenzije stranic }
  j: integer;         { števec po dimenzijah }

procedure Obhod(n: integer; var Dolz: DimenzijeT);
{ Sprehod skozi celice n-dimenzionalnega kvadra s stranicami Dolz. }
var i, j, k, m: integer;
begin
  m := 1; for j := 1 to n do m := m * Dolz[j];
  for i := 0 to m - 1 do begin
    k := i;
    for j := 1 to n do
      begin Write((k mod Dolz[j] + 1):3); k := k div Dolz[j] end;
    WriteLn;
  end; {for}
end; {Obhod}

begin {MarvinovVodic}
  Write('Dimenzija kvadra: '); ReadLn(n);
  WriteLn('Velikosti stranic');
  for j := 1 to n do begin Write(j, ': '); ReadLn(Dolz[j]) end;
  Obhod(n, Dolz);
end. {MarvinovVodic}
```

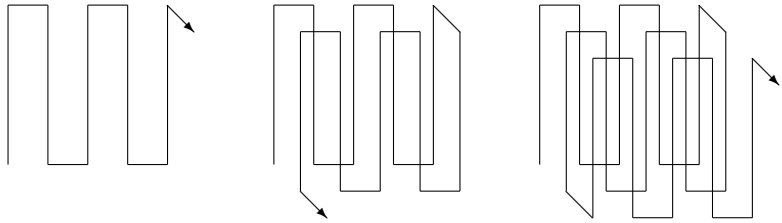
(*Opomba:* Write(j, ': ') lahko včasih pripelje do nerodnosti pri izpisu. Standard jezika pascal namreč ne predpisuje privzete širine polja pri izpisu

številskih vrednosti; torej je od prevajalnika odvisno, ali si bo Write(j) razlagal kot Write(j:1) ali pa mogoče kot Write(j:10) ali kaj podobnega. Da se izognemo morebitnim presenečenjem, bi bilo varneje uporabiti Write(j:1, ': ').)

Če bi hoteli robotu prihraniti pot (česar pa naloga ni zahtevala), bi lahko uporabili naslednji program. Z njim robot vedno stopi v *sosednjo* celico skladišča. Tu zapisani program je v tesni zvezi z nalogo o Grayevem kodiranju iz knjige „Enajsta šola računalništva“ (nalogi 1032B in 1133B), ki kaže na nekaj presenetljivih povezav med Hamiltonovo potjo skozi kvader (naš problem), Grayevim kodiranjem, hanojskimi stolpi in še čim.

```
program ZaLenobe(Input, Output);
const
  MaxN = 10;                          { največja dimenzija kvadra }
type
  DimenzijeT = array [1..MaxN] of integer;
var
  n: integer;                         { dimenzija kvadra }
  Dolz: DimenzijeT;                   { dimenzije stranic }
  Koord: DimenzijeT;                  { koordinate pregledovane kockice }
  Raste: array [1..MaxN] of boolean;  { števec v tej koordinati raste }
  Konec: boolean;                     { konec korakanja }
  Prenos: boolean;                    { stopiti moramo še po naslednji koordinati }
  i: integer;                         { števec }
begin
  Write('Dimenzija kvadra: '); ReadLn(n);
  WriteLn('Velikosti stranic');
  for i := 1 to n do begin Write(i:1, ': '); ReadLn(Dolz[i]) end;
  for i := 1 to n do begin Koord[i] := 1; Raste[i] := true end; { začetek }
  repeat
    for i := 1 to n do Write(Koord[i]:3);  WriteLn; { izpišemo, kje smo }
    i := 0; Konec := false;
    repeat
      Prenos := false; i := i + 1;      { stopimo po i-ti koordinati }
      if i > n then Konec := true       { prenos z zadnjega mesta = konec poti }
      else if Raste[i] then begin
        if Koord[i] < Dolz[i] then Koord[i] := Koord[i] + 1 { naprej }
        else begin Raste[i] := not Raste[i]; Prenos := true end;
      end
      else begin
        if Koord[i] > 1 then Koord[i] := Koord[i] - 1       { nazaj }
        else begin Raste[i] := not Raste[i]; Prenos := true end;
      end; {if}
      { če je Prenos = true, moramo stopiti še po koordinati i + 1 }
    until not Prenos;
  until Konec;
end. {ZaLenobe}
```

Pri tem načinu premikanja imamo za vsako dimenzijo predvideno neko trenutno smer gibanja (v smeri naraščajočih ali pa v smeri padajočih koordinat). Poskusimo se premakniti v smeri prve dimenzije, če pa to ne gre v želeno smer (ker bi padli ven iz kvadra), se bomo poskusili premakniti v naslednji dimenziji, smer prve dimenzije pa v mislih obrnemo. Tako bi na primer hodili najprej cik-cak v prvih dveh dimenzijah, ko pa bi te celice izčrpali, bi naredili en korak v tretji dimenziji in nato spet cik-cak v prvih dveh dimenzijah, le da v nasprotni smeri kot doslej.



Podobna tej nalogi je tudi 2002.1.1 (str. 483, rešitev na str. 500), kjer je treba preračunavati med koordinatami celice in njenim položajem na robotovi poti.

R1990.2.2

N: 81

Prva preprosta rešitev je naslednja. Če bi zlili obe zaporedji v eno samo dolgo urejeno zaporedje, bi bilo v njem $2n$ elementov in srednji element bi bil torej tisti na indeksu n. Zdaj lahko simuliramo zlivanje, dokler ne pridemo do n-tega elementa. Rešitev je še posebej preprosta, saj nam zaradi enakih dolžin zaporedij ni treba paziti, ali nam bo zmanjkalo kakega zaporedja. Spodnji program se pomika po prvem zaporedju s števcem ia, po drugem z ib, vsakič pa se premakne naprej po tistem zaporedju, ki ima na trenutnem položaju manjši element. Po $n - 1$ takšnih korakih kaže eden od teh dveh indeksov ravno na n-ti element zlitega zaporedja.

```
type TabelaT = array [1..100] of real;

function Srednji(n: integer; var a, b: TabelaT): real;
var i, ia, ib: integer;
begin
  ia := 1; ib := 1;
  for i := 1 to n − 1 do
    { Invarianta: manjši izmed elementov a[ia] in b[ib] je obenem tudi
      i-ti najmanjši v uniji vseh elementov zaporedij a in b. }
    if a[ia] < b[ib] then ia := ia + 1 else ib := ib + 1;
  if a[ia] < b[ib] then Srednji := a[ia] else Srednji := b[ib];
end; {Srednji}
```

Vendar pa je časovna zahtevnost tega postopka $O(n)$. Obstaja tudi bistveno hitrejša rešitev, ki temelji na hkratni bisekciji obeh zaporedij. Zaradi enostavnejšega razmišljanja predpostavimo, da so v zaporedjih a in b sama različna števila (če to v praksi ne drži, se lahko pri primerjanju dveh elementov istega zaporedja, če se izkaže, da sta enaka, delamo, da je manjši tisti z manjšim indeksom; če sta iz različnih zaporedij, pa se delajmo, da je manjši tisti iz zaporedja a).

Iz definicije srednjega elementa sledi, da bo srednji element, ki ga iščemo (recimo mu x), n-ti najmanjši izmed vseh $2n$ elementov zaporedij a in b. Torej je $n-1$ elementov manjših od njega, n pa večjih od njega.

Recimo za začetek, da je n lih: $n = 2k-1$ za nek $k \geq 1$. Srednji element zaporedja a je torej a_k, srednji element zaporedja b pa je b_k. Primerjajmo a_k in b_k. Če se izkaže, da je $a_k < b_k$, to pomeni, da so od b_k manjši vsi elementi $a_1, \ldots, a_k$ in $b_1, \ldots, b_{k-1}$; to je skupaj $k+(k-1) = 2k-1 = n$ elementov. Od b_k je torej manjših vsaj n elementov (mogoče pa še kakšen več iz zaporedja a); mi pa smo v prejšnjem odstavku videli, da je od iskanega srednjega elementa x manjših natanko $n-1$ elementov. Torej je $b_k > x$, vsi nadaljnji elementi zaporedja b pa so seveda še večji in torej noben od njih ne more biti x. Zato lahko elemente $b_{k+1}, b_{k+2}, \ldots, b_n$ zavržemo.[14] Iz $a_k < b_k$ pa sledi tudi, da so od a_k večji vsi elementi $a_{k+1}, \ldots, a_n$ in $b_k, b_{k+1}, \ldots, b_n$, torej je od a_k večjih vsaj $(k-1)+k = n$ elementov. Od prej vemo, da je od x večjih natanko n elementov, torej mora biti $a_k = x$ (če je od a_k večjih natanko n elementov) ali pa $a_k < x$ (če je od a_k večjih več kot n elementov). V vsakem primeru to pomeni, da so prejšnji elementi zaporedja a, torej $a_1, \ldots, a_{k-1}$, vsi manjši od x in jih lahko tudi zavržemo.

Če bi se izkazalo, da je $a_k > b_k$, bi lahko opravili enak razmislek kot v gornjem odstavku, le namesto a bi si povsod mislili b in obratno. Učinek je v obeh primerih enak: iz enega zaporedja zavržemo $k-1$ elementov, ki so bili vsi manjši od x, iz drugega pa tudi prav toliko elementov, ki pa so bili vsi večji od x. Zato imata na ta način skrajšani zaporedji še vedno isti srednji element; obenem pa sta tudi še vedno obe enako dolgi. Torej imamo pred seboj problem enake oblike kot na začetku, le s krajšima zaporedjema, in njegova rešitev je prav isti x, ki je obenem tudi rešitev prvotnega problema.

Doslej smo govorili o možnosti, da je n lih. Recimo zdaj, da je n sod: $n = 2k$ za nek $k \geq 1$. Srednja elementa sta spet a_k in b_k. Če je $a_k < b_k$, je od b_k manjših vsaj $k+(k-1) = n-1$ elementov, torej je $b_k \geq x$ in lahko zavržemo člene $b_{k+1}, \ldots, b_n$; od a_k pa je večjih vsaj $k+(k+1) = n+1$ elementov, torej je $a_k < x$ in lahko zavržemo člene $a_1, \ldots, a_k$. — Če je $a_k > b_k$, spet razmišljamo podobno, le a in b imata zamenjani vlogi.

[14] S to utemeljitvijo bi lahko zavrgli tudi b_k, a tega ne bomo storili. Tako bomo zagotovili, da bomo iz a zavrgli enako mnogo elementov kot iz b in bosta zaporedji tudi po tem ohranili enako dolžino. Zato bomo lahko v nadaljevanju iskali srednji element skrajšanih zaporedij z enakim postopkom kot tu na prvotnih zaporedjih.

V vseh primerih smo dobili dve krajši zaporedji (namesto po n elementov imata po $(n+1)\operatorname{div}2$ elementov) in zanju vemo, da je srednji element njune unije isti kot srednji element x unije obeh prvotnih zaporedij. Torej lahko iskano število x poiščemo tako, da se z enakim postopkom lotimo skrajšanih zaporedij. Postopek se lahko ustavi, ko dobimo dve zaporedji dolžine 1; obe skupaj imata torej le dva elementa in takrat je „srednji element“ kar manjši izmed teh dveh elementov. V vsakem koraku imamo le konstantno mnogo dela, dolžina zaporedij pa se nam približno razpolovi, zato je časovna zahtevnost dobljenega postopka samo $O(\lg n)$.

V praksi imamo zaporedji predstavljeni z dvema tabelama in ko je treba zavreči nek del zaporedja, teh elementov ni treba zares brisati, saj vedno brišemo z začetka ali s konca zaporedij. Zato je dovolj, če si zapomnimo indeks prvega in zadnjega še nezbrisanega elementa, tadva indeksa pa potem upoštevamo tudi pri računanju srednjih elementov: če ima zaporedje namesto $a_1, \ldots, a_n$ člene $a_l, \ldots, a_r$, njegov srednji element ni $a_{(n+1)\operatorname{div}2}$, pač pa $a_{(l+r)\operatorname{div}2}$, njegova dolžina pa ni n, pač pa $r-l+1$. To, ali je ta dolžina soda ali ne, si spodnji podprogram zapomni v spremenljivki Zamik.

```
type TabelaT = array [1..100] of real;

function Srednji(n: integer; var a, b: TabelaT): real;
var
  aLevi, bLevi, aDesni, bDesni, aSrednji, bSrednji: integer;
  Zamik: boolean;
begin
  aLevi := 1; aDesni := n; bLevi := 1; bDesni := n;
  while aLevi < aDesni do begin
    aSrednji := (aLevi + aDesni) div 2; bSrednji := (bLevi + bDesni) div 2;
    Zamik := Odd(aDesni - aLevi);
    if a[aSrednji] <= b[bSrednji] then begin
      aLevi := aSrednji; bDesni := bSrednji;
      if Zamik then aLevi := aLevi + 1;
    end else begin
      aDesni := aSrednji; bLevi := bSrednji;
      if Zamik then bLevi := bLevi + 1;
    end; {if}
  end; {while}
  if a[aLevi] < b[bLevi] then Srednji := a[aLevi] else Srednji := b[bLevi];
end; {Srednji}
```

R1990.2.3

Hladen gorilnik se od gorečega soseda segreje do vžiga v 0,8 sekunde. Po vžigu gori največ eno sekundo, dokler ne porabi vsega goriva. Plamen se more prenesti le na sosednji, z gorivom dovolj napolnjeni gorilnik. Po ugasnitvi traja pet sekund, da se gorilnik dovolj napolni z gorivom in je ponovno pripravljen na vžig. V tem času se ohladi

N: 81

dovolj, da ne more priti do samovžiga. Ker gori plamen največ eno sekundo, se lahko širi le v eno smer (proti polnim gorilnikom), ne more pa se vrniti po poti pravkar ugaslih gorilnikov. Plamen se od netilca širi na obe strani tako, da hkrati gorita eden do dva sosednja gorilnika na vsaki strani. Ker so gorilniki povezani v krog, se oba potujoča plamena srečata (v gorilniku diametralno nasproti netilca) po štiridesetih sekundah. Tam plamen ugasne zato, ker se pravkar izgoreli gorilniki v eni sekundi (dokler traja plamen) ne napolnijo z gorivom dovolj za ponoven vžig. Od tedaj se noben gorilnik ne prižge več.

N: 82

R1990.2.4

Po izteku obeh programov lahko spremenljivka zavzame poljubno vrednost med vključno 2 in 20000. Izberimo si poljuben n med 2 in 20000 in pokažimo enega od načinov, kako dobimo na koncu izvajanja programa vrednost n.

Označimo $p = n \operatorname{div} 2$ in $q = n - p$. Ker je $2 \le n \le 20000$, velja $1 \le p, q \le 10000$.

Prvi program prebere 0 in čaka. Drugi program $(10000 - p)$-krat prebere, poveča in zapiše vrednost spremenljivke. Zapisana vrednost je zdaj $10000 - p$, drugemu programu ostane še natanko p ponovitev.

Prvi program poveča svojo prebrano vrednost in zapiše 1. Drugi program prebere 1 in čaka. Prvi program $(10000 - q)$-krat prebere, poveča in zapiše vrednost spremenljivke. Zapisana vrednost je zdaj $10000 - q + 1$, prvemu programu ostane še natanko $q - 1$ ponovitev.

Drugi program poveča svojo prebrano vrednost in zapiše 2; zdaj mu ostane še natanko $p - 1$ ponovitev. Vse svoje preostale ponovitve izvede, preden prvi program nadaljuje z delom. Ob njegovem zaključku je vrednost spremenljivke enaka $p + 1$. Po zaključku dela drugega programa nadaljuje z delom prvi program: prebere vrednost $p + 1$ in jo v $q - 1$ ponovitvah poveča do $p + q = n$, ko konča s svojim delom.

Naj bo m število programov. Največja vrednost je enaka $10000 \cdot m$. Za $m = 1$ je najmanjša vrednost spremenljivke enaka največji, medtem ko je za $m \ge 2$ enaka 2. O tem se lahko prepričamo takole: če hočemo dobiti vrednost, manjšo ali enako 20000, lahko uporabimo enak postopek kot zgoraj, ostale programe pa pustimo od začetka do konca teči v času med tistim, ko prvi program prebere 0, in časom, ko ta program zapiše 1. Če pa hočemo vrednost, večjo od 20000, lahko uporabimo gornji postopek, da pridemo do 20000, na konec dodamo še toliko branj in pisanj iz ostalih programov (lepo sinhroniziranih, brez prepletanja), da bo končna vrednost enaka želeni, odvečna branja in pisanja ostalih programov pa spet lahko stlačimo v čas med tistim, ko prvi program prebere 0, in tistim, ko zapiše 1.

Vrednosti nad $10000 \cdot m$ očitno ne moremo dobiti, saj je vseh povečevanj skupaj le $10000 \cdot m$. Pri $m = 1$ tudi ne moremo dobiti drugačne vrednosti kot 10000, saj je možen potek računanja le ta, da edini program lepo po vrsti izvede vsa svoja branja in pisanja.

Prepričajmo se še, da ne moremo dobiti vrednosti, manjših od 2. Ker je spremenljivka na začetku enaka 0 in je nikoli ne zmanjšujemo, je njena vrednost vedno nenegativna; zato pa, ko jo nek procesor prebere in poveča za 1 ter zapiše, bo gotovo zapisal pozitivno vrednost. Torej končna vrednost spremenljivke ne more biti 0 (ker bi bilo to mogoče le, če ne bi noben procesor nikoli ničesar zapisal). Končna vrednost bi bila lahko 1 le, če bi procesor, ki izvede zadnje pisanje, pri svojem zadnjem branju prebral ničlo; toda to bi se dalo le, če ne bi pred njim nihče pisal, v resnici pa je (če nihče drug) pred tistim svojim zadnjim branjem 9999-krat pisal že on sam.

REŠITVE NALOG ZA TRETJO SKUPINO

R1990.3.1

N: 82

Razpored domin bomo iskali z rekurzijo. V spodnjem programu počne to podprogram Postavi, ki poskuša na konec trenutnega razporeda dodati še eno domino. Preizkusiti mora vse možne domine, vsako v obeh možnih položajih; če se nova domina ujema s tisto, ki je bila dotlej zadnja, bomo poskusili z rekurzivnim klicem poskrbeti za razporeditev še preostalih domin. Koristno je voditi množico domin, ki smo jih že postavili v trenutni razpored (Zasedene), tako da pri razmišljanju o tem, kaj bi postavili na naslednje mesto, ne bomo izgubljali časa z njimi. Ko dodamo domino v razpored, jo dodamo tudi v to množico, ko pa se nato rekurzivni klic vrne in bomo namesto nje poskušali dodati kakšno drugo, jo moramo iz množice spet zbrisati.

```
program Domine(Output);
const
  n = 10;    { število domin, ki jih želimo postaviti v vrsto }
  m = 5;     { največje število pik na polju domine }
type
  DominaT = array [0..1] of 1..m;    { domina ima dve polji }
  PostavitevT = record               { domina v vrsti }
                  Dom: integer;      { številka domine }
                  Obrat: integer;    { liho: obrnjena; sodo: neobrnjena }
                end; {PostavitevT}
var
  Vrsta: array [0..n] of PostavitevT;    { vrsta, ki jo sestavljamo }
  Zaloga: array [1..n] of DominaT;       { domine, ki jih imamo na razpolago }
  Zasedene: set of 1..n;                 { domine, ki so že v vrsti }

procedure NapraviDomine;
{ Prebere, izračuna, napravi ali načara zalogo domin; na primer takole: }
var i: integer;

  function Random(Min, Max: integer): integer; external;

begin
```

```
  for i := 1 to n do begin
    Zaloga[i, 0] := Random(1, m); Zaloga[i, 1] := Random(1, m);
    WriteLn(i:3, ' (', Zaloga[i, 0]:1, ', ', Zaloga[i, 1]:1, ')');
  end; {for}
end; {NapraviDomine}

procedure IzpisiDomine;
var i: integer;
begin
  for i := 1 to n do with Vrsta[i] do
    WriteLn(Dom:3, ' (', Zaloga[Dom, Obrat mod 2]:1, ', ',
            Zaloga[Dom, (Obrat + 1) mod 2]:1, ')');
end; {IzpisiDomine}

function StPik(v: PostavitevT; Poz: integer): integer;
{ Vrne število pik domine na polju Poz, upoštevajoč rotacijo domine. }
begin
  StPik := Zaloga[v.Dom, (v.Obrat + Poz) mod 2];
end; {StPik}

function Primerjaj(v1, v2: PostavitevT): boolean;
{ Primerja dve domini. }
begin
  if Zasedene = [] then Primerjaj := true { prve domine ne primerjamo s prejšnjo }
  else Primerjaj := StPik(v1, 1) = StPik(v2, 0);
end; {Primerjaj}

function Postavi(Mesto: integer): boolean;
{ Postavi domine od Mesto naprej. }
var
  Nova: PostavitevT;          { domina, ki jo bomo postavili na Mesto }
  Uspeh, Neuspeh: boolean;    { uspeh/neuspeh postavitve vrste od Mesto naprej }
begin
  with Nova do begin Dom := 0; Obrat := 0 end; { začnemo s prvo domino }
  Uspeh := false; Neuspeh := false;
  repeat
    Nova.Dom := Nova.Dom + 1;
    if Nova.Dom > n then { ni šlo — poskusimo z obrnjenimi dominami }
      with Nova do begin Dom := 1; Obrat := Obrat + 1 end;
    if Nova.Obrat > 1 then
      Neuspeh := true { izčrpali smo vse možnosti — ne gre }
    else if not (Nova.Dom in Zasedene)
          and Primerjaj(Vrsta[Mesto - 1], Nova) then begin
      Vrsta[Mesto] := Nova; Zasedene := Zasedene + [Nova.Dom];
      if Mesto = n then Uspeh := true else Uspeh := Postavi(Mesto + 1);
      if not Uspeh then Zasedene := Zasedene - [Nova.Dom];
    end; {if}
```

```
  until Uspeh or Neuspeh;
  Postavi := Uspeh;
end; {Postavi}

begin {Domine}
  NapraviDomine; Zasedene := [];
  if not Postavi(1) then WriteLn('Domin se ne da postaviti v vrsto.')
  else begin WriteLn('Domine se da postaviti v vrsto:'); IzpisiDomine end;
end. {Domine}
```

R1990.3.2

N: 83

Lahko si pomagamo z zlivanjem. To je postopek, s katerim dobimo iz dveh ali več urejenih zaporedij novo, daljše urejeno zaporedje, v katerem so vsi elementi vhodnih zaporedij. Postopek je preprost: pogledamo prvi element vsakega vhodnega zaporedja in najmanjšega med njimi premaknemo iz tega zaporedja na konec izhodnega zaporedja. Ta korak zdaj ponavljamo in tako v izhodno zaporedje vsakič dodamo naslednji element po velikosti; ko se vsa vhodna zaporedja izpraznijo, je postopek končan.

Pri nas bodo zaporedja tiri, elementi zaporedij pa vagoni. Postopek se nam bo rahlo zapletel, ker vagon, ki ga kot zadnjega dodamo na nek tir, kasneje prvi pride z njega; ko torej pri zlivanju dodajamo na tir vagone po naraščajočih številkah, bo kasneje, ko bomo ta tir uporabili kot enega od vhodov pri nekem kasnejšem zlivanju, videti, kot da so vagoni na njem urejeni v nasprotnem vrstnem redu, torej po padajočih številkah, saj se bo s tega tira prvi pripeljal vagon z največjo številko in tako naprej.

Kakorkoli že, na začetku imamo le eno neurejeno zaporedje 32 vagonov (na tiru 0). Preden bomo lahko kaj zlivali, ga moramo razbiti na več krajših zaporedij. Pri tem moramo paziti na pravilo, da se vagon ne more nikoli premakniti na tir z manjšo številko od tistega, na katerem se nahaja trenutno; pri zlivanju nam torej vagoni prihajajo na tire z vse višjimi številkami in paziti moramo, da nam ne bo tirov zmanjkalo, še preden bodo opravljena vsa zlivanja.

Za začetek premaknimo en vagon s tira 0 na tir 1. Zdaj si lahko mislimo, da imata tako tir 0 kot tir 1 urejeno zaporedje dolžine 1 (na tiru 0 je sicer za tem enim vagonom še trideset drugih v nekem neznanem vrstnem redu, ampak zanje se zdajle pri zlivanju ne bomo zmenili). Ti dve zaporedji zdaj zlijmo in na tiru 2 dobimo (narobe obrnjeno) urejeno zaporedje dveh vagonov.

Premaknimo spet en vagon s tira 0 na tir 1. Lahko se delamo, da imamo na tirih 0 in 1 narobe obrnjeni urejeni zaporedji dolžine 1; na tiru 2 pa je še od prej narobe obrnjeno urejeno zaporedje dolžine 2. Vse troje zlijmo in na tiru 3 dobimo (prav obrnjeno) urejeno zaporedje dolžine 4.

Zdaj lahko s podobnimi operacijami kot prej pridelamo na tiru 2 še eno (prav obrnjeno) urejeno zaporedje dolžine 2, na tirih 0 in 1 pa po en vagon; ko vse to (vključno s štirimi vagoni na tiru 3) zlijemo, dobimo na tiru 4 narobe obrnjeno urejeno zaporedje dolžine 8.

Potem spet s podobnimi operacijami pripravimo na tirih 1, 2 in 3 narobe obrnjena urejena zaporedja dolžine 1, 2 in 4; nato zlivamo s tirov 0–4 na tir 5 in dobimo tam prav obrnjeno urejeno zaporedje dolžine 16.

Če vse skupaj ponovimo še enkrat, da dobimo na tirih 1–4 spet urejena zaporedja dolžine 1, 2, 4 in 8, lahko zdaj zlijemo vse to skupaj s tirom 5 in še zadnjim preostalim vagonom s tira 0 ter rezultat (urejeno zaporedje dolžine 32) odpošiljamo naravnost na izhodni tir 6.

Dobro je paziti še na naslednje: pri našem postopku vedno, ko prvič pošiljamo vagone na nek tir, nastane na njem zaporedje, ki je obrnjeno ravno narobe kot ob prvem zlivanju na prejšnji tir. Ker smo začeli s tem, da smo si na tiru 1 mislili prav obrnjeno zaporedje dolžine 1, smo na tiru 2 dobili narobe obrnjeno, na tiru 3 spet prav obrnjeno in tako naprej, na koncu pa na tiru 5 tudi prav obrnjeno. Zato nam pri tistem zadnjem zlivanju vagoni odhajajo na izhodni tir po naraščajočih številkah, tako kot smo želeli. Če pa bi imeli na primer 64 vagonov in šest pomožnih tirov, bi nam pri opisanem postopku na koncu nastala narobe obrnjena zaporedja in tudi vagoni bi na izhodni tir prihajali po padajočih številkah. Očitno je, da bi morali v tem primeru že od vsega začetka ravno obrniti vrstni red na vseh tirih (torej začeti s predpostavko, da je osamljeni vagon na tiru 1 narobe obrnjeno urejeno zaporedje dolžine 1). To je odvisno od tega, ali je število tirov sodo ali liho. Spodnji program prenaša podatke o zahtevani urejenosti (naraščajoči ali padajoči) kar s parametrom ob rekurzivnih klicih. Glavno je to, da imamo pred zadnjim zlivanjem (tistim, ki bo pošiljalo vagone na izhodni tir), na vseh tirih naraščajoča zaporedja.

```
program ZelezniskaPostaja;
const
  n = 5;                { število slepih tirov }
  VhodniTir = 0;        { indeks vhodnega tira }
  IzhodniTir = n + 1; { indeks izhodnega tira }

  procedure PremakniVagon(OdKod, Kam: integer); external;
  function Prazen(Tir: integer): boolean; external;
  function Vagon(Tir: integer): integer; external;

{ Zlije vsebino tirov 1..StTirov na tir StTirov + 1. V zlivanje vključi tudi prvi vagon
  s tira 0. Če je Obrnjeno = true, predpostavi, da so vagoni na vhodnih tirih urejeni
  padajoče. Na tiru StTirov + 1 bo urejenost v vsakem primeru ravno nasprotna. }
procedure Zlivanje(StTirov: integer; Obrnjeno: boolean);
var Ze0: boolean; Tir, Min, KjeMin: integer;
begin
  Ze0 := false; { Vagona s tira 0 še nismo premaknili. }
  repeat
    { Poiščimo med prvimi vagoni s tirov 1..StTirov najmanjšega
      (ali največjega, če je Obrnjeno = true). Če je Ze0 = false,
      gledamo tudi tir 0. }
```

```
      KjeMin := −1; if Ze0 then Tir := 1 else Tir := 0;
      while Tir <= StTirov do begin
        if not Prazen(Tir) then
          if (KjeMin < 0) or ((Vagon(Tir) < Min) <> Obrnjeno) then
            begin Min := Vagon(Tir); KjeMin := Tir end;
        Tir := Tir + 1;
      end; {while}
      { Premaknimo najdeni vagon na tir StTirov + 1. }
      if KjeMin >= 0 then PremakniVagon(KjeMin, StTirov + 1);
      if KjeMin = 0 then Ze0 := true; { s tira 0 ne smemo več brati }
    until KjeMin < 0;
end; {Zlivanje}

{ Predpostavi, da so tiri 1..StTirov trenutno prazni. Poskrbi, da se za vse i od 1
  do StTirov na tiru i nahaja 2^i vagonov, urejenih naraščajoče, če je Obrnjeno = true,
  in padajoče, če je Obrnjeno = false. Vse te vagone vzame z vhodnega tira 0. }
procedure NapolniTire(StTirov: integer; Obrnjeno: boolean);
begin
  if StTirov = 1 then PremakniVagon(0, 1)
  else begin
    { Napolnimo prvih StTirov − 1 tirov v nasprotnem vrstnem redu. }
    NapolniTire(StTirov − 1, not Obrnjeno);
    { Z zlivanjem dobimo na tiru StTirov ravno prav vagonov
      v ravno pravem vrstnem redu. }
    Zlivanje(StTirov − 1, not Obrnjeno);
    { Prvih StTirov − 1 je spet praznih, napolnimo jih zdaj v
      želenem vrstnem redu. }
    NapolniTire(StTirov − 1, Obrnjeno);
  end;
end; {NapolniTire}

begin {ZelezniskaPostaja}
  { Pripravimo na i-tem tiru 2^i vagonov v pravem vrstnem redu,
    za vse i od 1 do n. En vagon ostane še na vhodnem tiru. }
  NapolniTire(n, false);
  { Zdaj jih zlijemo na izhodni tir. Kot si želimo, bodo
    prišli vagoni z nižjimi številkami prej na izhodni tir. }
  Zlivanje(n, false);
end. {ZelezniskaPostaja}
```

Malo drugačen, čeprav po svoje zelo podoben, pa je tudi naslednji razmislek. Vagone v mislih razdelimo v dve skupini: 1–16 in 17–32. Iz naloge je videti, da bi moralo biti pet pomožnih tirov dovolj za urejanje $32 = 2^5$ vagonov, torej si lahko mislimo, da bi utegnili biti za 16 vagonov dovolj že štirje tiri. Uredimo torej najprej vagone od 1 do 16 z uporabo pomožnih tirov 2–5, pomožni tir 1 pa uporabimo zato, da nanj odlagamo vagone s številkami 17–32, ko jih dobivamo

z vhodnega tira. Ko je to opravljeno in smo vagone 1–16 srečno in v pravem vrstnem redu odposlali na izhodni tir, lahko zdaj uredimo še preostalih 16 vagonov, torej tiste s številkami 17–32, ki nas čakajo na tiru 1, kamor smo jih prej odložili.

Za urejanje 16 vagonov bi seveda uporabili enak razmislek: razdelimo jih v dve skupini po osem, tiste iz druge skupine odlagamo na prvi prosti pomožni tir, ostale pa urejamo sproti. To rekurzivno razmišljanje se konča pri enem vagonu, ki ga „uredimo“ brez kakršnih koli pomožnih tirov preprosto tako, da ga pošljemo z vhodnega tira na izhodnega. Program nam malce zaplete le dejstvo, da je hkrati v teku več urejanj: tiste vagone, ki jih ne odlagamo na pomožni tir, takoj pošiljamo v obdelavo naslednjemu urejanju, ki jih bo mogoče odložilo na svoj pomožni tir, mogoče pa spet poslalo naprej in podobno.

```
procedure PrenosNaprej(STira, PomozniTir, StOd, StDo, StVagona: integer);
{ Ta podprogram opravi en korak urejanja za vagone s številkami StOd..StDo.
  Trenutno je treba nekaj narediti z vagonom StVagona, ki prihaja s tira STira. Če je
  ta vagon s prve polovice intervala StOd..StDo, ga predamo podprogramu za urejanje
  te polovice (z rekurzivnim klicem), sicer pa ga odložimo na stranski tir. }
var Meja: integer;
begin
  Meja := (StOd + StDo) div 2;
  { Vagone (Meja + 1)..StDo bomo odložili na pomožni tir,
    vagone StOd..Meja pa bomo prenesli naprej. Pri StOd = StDo
    je možen itak en sam vagon in pomožni tir je tedaj isti kot izhodni. }
  if (StVagona > Meja) or (StOd = StDo) then PremakniVagon(STira, PomozniTir)
  else PrenosNaprej(STira, PomozniTir + 1, StOd, Meja, StVagona);
end; {PrenosNaprej}

procedure SprazniTir(StTira, StOd, StDo: integer);
{ Predpostavi, da se na tiru StTira nahajajo vagoni StOd..StDo. Poskrbi,
  da se ta tir sprazni in vsi vagoni pridejo na izhodni tir v pravem vrstnem redu. }
var Meja: integer;
begin
  Meja := (StOd + StDo) div 2;
  if StTira < n then if not Prazen(StTira + 1) then
    { Naslednji tir bomo potrebovali kot pomožni tir, a so na njem
      še stari vagoni, torej najprej spraznimo tega. }
    SprazniTir(StTira + 1, StOd − (StDo − Meja), StOd − 1);
  { Spraznimo zdaj zahtevani tir. }
  while not Prazen(StTira) do begin
    PrenosNaprej(StTira, StTira + 1, StOd, StDo, Vagon(StTira));
  end; {while}
  { Z rekurzivnim klicem spraznimo še naslednje pomožne tire. }
  if StTira < n then SprazniTir(StTira + 1, Meja + 1, StDo);
end; {SprazniTir}
```

```
begin {ZelezniskaPostaja}
  SprazniTir(VhodniTir, 1, 1 shl n);
end. {ZelezniskaPostaja}
```

Isti postopek lahko zapišemo tudi bolj eksplicitno. Vagone si mislimo oštevilčene od 0 do 31 namesto od 1 do 32. Če skušamo sprazniti tir $n-b$, si mislimo, da so tiri $n-b+1, \ldots, n$ že prazni in predpostavimo, da se številke vagonov na tiru $n-b$ razlikujejo le v spodnjih b bitih, v preostalih $n-b$ bitih pa imajo vsi ti vagoni enake vrednosti. Zdaj pri vsakem vagonu poglejmo, kateri je v njegovi številki najvišji prižgani bit izmed spodnjih b bitov; če je to bit $b-1$, gre ta vagon na tir $n-b+1$, če je to bit $b-2$, gre na $n-b+2$ in tako naprej. Če ima nek vagon na spodnjih b bitih same ničle, gre naravnost na izhodni tir. Na koncu z rekurzivnimi klici obdelajmo vagone, ki so se nam nabrali na tirih $n-b+1, \ldots, n$.

```
const n = 5; { Število slepih tirov; dvojiški logaritem števila vagonov. }

  procedure PremakniVagon(OdKod, Kam: integer); external;
  function Prazen(Tir: integer): boolean; external;
  function Vagon(Tir: integer): integer; external;

{ Vrne indeks najvišjega prižganega bita v x med biti 0..(m - 1).
  Če so same ničle, vrne -1. }
function NajvisjiBit(x, m: integer): integer;
var j: integer;
begin
  j := m - 1;
  while j >= 0 do
    if (x and (1 shl j)) <> 0 then break
    else j := j - 1;
  NajvisjiBit := j;
end; {NajvisjiBit}

procedure Uredi(Tir: integer);
var i, StBitov: integer;
begin
  { Predpostavka: na tiru Tir so vagoni, katerih številke (če bi jih šteli od
    0 naprej namesto od 1 naprej) bi se razlikovale le v spodnjih StBitov bitih,
    tiri od Tir + 1 do n pa so prazni. }
  StBitov := n - Tir;
  { Razporedimo vagone med tire Tir + 1 do n + 1 glede na najvišji prižgani bit
    med spodnjimi StBitov biti. To nam zagotovi, da pridejo vsi s tira i + 1
    v vrstnem redu pred vsemi s tira i. }
  while not Prazen(Tir) do
    PremakniVagon(Tir, n - NajvisjiBit(Vagon(Tir) - 1, StBitov));
  { Počistimo tire od n do Tir + 1. }
  for i := n downto Tir + 1 do Uredi(i);
```

```
  { Rezultat: prazni so tiri od Tir do n, vagone, ki so bili prej na tiru Tir,
    pa smo zdaj v pravem vrstnem redu premaknili na izhodni tir. }
end; {Uredi}

begin {ZelezniskaPostaja}
  Uredi(0);
end. {ZelezniskaPostaja}
```

Še opomba glede funkcije NajvisjiBit. V njej smo uporabili operator **and** nad celimi števili, čeprav je v standardnem pascalu definiran le nad logičnimi vrednostmi; tudi operatorja **shl** in ukaza **break** standardni pascal nima. Če bi se torej hoteli bolj držati standarda, bi lahko naredili nekaj takega:

```
function NajvisjiBit(x, m: integer): integer;
var j: integer;
begin
  NajvisjiBit := −1;
  for j := 0 to m − 1 do begin
    if Odd(x) then NajvisjiBit := j;
    x := x div 2;
  end; {for}
end; {NajvisjiBit}
```

Majhna slabost te različice je, da gre vedno po vseh m bitih, medtem ko gre prejšnja različica od zgornjih bitov proti spodnjim in ustavi že pri prvem prižganem bitu (in od števil $0, \ldots, 2^m - 1$ je kar polovica takih, pri katerih je prižgan že kar najvišji bit).[15]

Razmislimo še o tem, kolikokrat naši algoritmi premikajo vagone. Naj bo $f(n)$ število klicev podprograma PremakniVagon, če delamo z 2^n vagoni in n pomožnimi tiri. Prvi algoritem ima $f(0) = 1$ (če imamo le en vagon, ga samo premaknemo z vhodnega tira na izhodnega) in $f(n) = 2f(n-1) + 2^{n-1}$ ($2f(n-1)$ zaradi dveh rekurzivnih klicev v podprogramu NapolniTire, 2^{n-1} pa zaradi zlivanja). Drugi in tretji algoritem imata tudi $f(0) = 1$, pri večjih n pa upoštevamo, da se pol vagonov (torej 2^{n-1} vagonov) odloži na prvi pomožni tir, z ostalimi pa ravnamo takoj tako, kot da bi imeli le $n-1$ tirov in 2^{n-1} vagonov; na koncu še tiste, ki smo jih odložili na prvi pomožni tir, uredimo po enakem postopku, torej spet kot da bi imeli le $n-1$ pomožnih tirov in 2^{n-1} vagonov. Tako smo spet dobili zvezo $f(n) = 2f(n-1) + 2^{n-1}$. Vsi trije algoritmi torej izvedejo enako število premikov. Če rekurzivno zvezo za $f(n)$ vstavljamo samo vase, lahko dobimo tudi eksplicitno obliko: $f(n) = 2f(n-1) + 2^{n-1} = 2(2f(n-2) + 2^{n-2}) + 2^{n-1} = 2^2 f(n-2) + 2 \cdot 2^{n-1} = 2^3 f(n-3) + 3 \cdot 2^{n-1} = \ldots = 2^n f(0) + n \cdot 2^{n-1} = (n+2)2^{n-1}$. Pri naši nalogi je $n = 5$ in potrebujemo 112 premikov.[16]

[15] Iskanja najvišjega prižganega bita bi se lahko lotili še na razne druge načine; glej rešitev naloge 2000.1.2 na str. 404.

[16] Gl. tudi *The On-Line Encyclopedia of Integer Sequences*, A001792.

Zanimivo vprašanje pri našem problemu urejanja vagonov s pomočjo pomožnih slepih tirov je tudi to, koliko pomožnih tirov potrebujemo, da lahko uredimo vsako zaporedje N vagonov. Izkaže se, da za šest ali manj vagonov zadoščata že dva pomožna tira (za sedem vagonov pa včasih potrebujemo že tri pomožne tire); potem pa, če znamo z n pomožnimi tiri urediti N vagonov, lahko z $n+1$ pomožnimi tiri uredimo $2N$ vagonov (z enakim rekurzivnim razmislekom, kakršnega smo videli že zgoraj v naši rešitvi za pet tirov in 32 vagonov). Z n pomožnimi tiri lahko torej vsekakor uredimo vsako zaporedje $3 \cdot 2^{n-1}$ ali manj vagonov; če to obrnemo, vidimo, da za urejanje poljubnega zaporedja N vagonov gotovo zadostuje $\lceil \log_2(2N/3) \rceil$ pomožnih tirov.[17]

R1990.3.3

Podprogram ShraniPaket spremlja število prispelih paketov vsake datoteke in podatke o njih hrani v tabeli Paketi. Ko prispejo vsi zapisi neke datoteke, jih z zaporednimi klici podprograma Pisi po vrsti izpiše in sprosti pomnilnik, ki so ga zasedali, da je na voljo novim datotekam. N: 83

```
const
  MaxPaketov = 1500; { več kot število datotek × povprečna dolžina }
  MaxDatotek = 10;
type
  ShranjenZapisT = record                    { oblika podatkov o prispelih zapisih }
                     Stevilka: integer;      { številka zapisa v datoteki }
                     Naslednji: integer;     { naslednji element v verigi }
                     Zapis: ZapisT;          { vsebina zapisa }
                   end; {ShranjenZapisT}
  OpisDatotekeT = record                     { oblika opisa prihajajoče datoteke }
                    ZadnjiZapis: integer;    { številka zadnjega zapisa }
                    Sprejetih: integer;      { število sprejetih zapisov }
                    PrviZapis: integer;      { prvi element v verigi zapisov }
                  end; {OpisDatotekeT}
var PrazniZapisi: integer; { prvi element v verigi prostih zapisov v tabeli Zapisi }
    Opisi: array [1..MaxDatotek] of OpisDatotekeT;      { opisi datotek }
    Zapisi: array [1..MaxPaketov] of ShranjenZapisT;    { seznam zapisov }

  procedure Pisi(Datoteka: integer; Zapis: ZapisT); external;
```

[17] To je torej zgornja meja za minimalno potrebno število pomožnih tirov, s katerimi se da urediti vsa zaporedja N vagonov; znana pa je tudi spodnja meja, namreč $\frac{1}{2}\log_2 N$ — z manj kot toliko tiri se gotovo ne bo dalo urediti vseh zaporedij N vagonov. Naš problem urejanja vagonov lahko tudi posplošimo, če skladov (= pomožnih slepih tirov) ne zvežemo v zaporedje, ampak v poljuben drug graf, in če namesto skladov v vozliščih grafa dovolimo tudi vrste. Literatura: Knuth, *The Art of Computer Programming*, 3. knjiga, nalogi 5.2.4.19–20; R. Tarjan, *Sorting using networks of queues and stacks*, Journal of the ACM, 19(2):341–346, April 1972; T. Jiang, M. Li, P. Vitányi, *Average-case analysis of algorithms using Kolmogorov complexity*, J. of Comp. Science and Technology, 15(5):402–408, September 2000.

```
procedure PripraviPakete;
{ Pripravi tabeli Opisi in Zapisi. }
var i: integer;
begin
  { Cela tabela Zapisi je ena sama dolga veriga praznih zapisov. }
  PrazniZapisi := 1;
  for i := 1 to MaxPaketov − 1 do Zapisi[i].Naslednji := i + 1;
  Zapisi[MaxPaketov].Naslednji := −1;
  { Na začetku še ne sprejemamo nobene datoteke. }
  for i := 1 to MaxDatotek do with Opisi[i] do
    begin ZadnjiZapis := 0; Sprejetih := 0; PrviZapis := −1 end;
end; {PripraviPakete}

procedure SprostiDatoteko(Dat: integer);
{ Sprosti prostor v tabeli Zapisi, ki ga je zasedala datoteka Dat.
  Njene zapise postavimo na začetek verige praznih zapisov. }
var i, j: integer;
begin
  j := PrazniZapisi;
  with Opisi[Dat] do begin
    i := PrviZapis; PrviZapis := −1;
    ZadnjiZapis := 0; Sprejetih := 0; PrazniZapisi := i;
  end; {with}
  while Zapisi[i].Naslednji <> −1 do i := Zapisi[i].Naslednji;
  Zapisi[i].Naslednji := j;
end; {SprostiDatoteko}

procedure ShraniPaket(p: PaketT);
{ Shrani prispeli paket p v tabelo Zapisi in izpiše datoteko, če so prišli že vsi zapisi. }
var i, j: integer;
begin
  if p.VrstaPaketa = Zadnji then
    Opisi[p.Datoteka].ZadnjiZapis := p.Stevilka;
  Opisi[p.Datoteka].Sprejetih := Opisi[p.Datoteka].Sprejetih + 1;
  i := PrazniZapisi; PrazniZapisi := Zapisi[i].Naslednji;
  with Zapisi[i] do begin
    Naslednji := Opisi[p.Datoteka].PrviZapis;
    Stevilka := p.Stevilka; Zapis := p.Zapis;
  end; {with}
  Opisi[p.Datoteka].PrviZapis := i;
  with Opisi[p.Datoteka] do
    if Sprejetih = ZadnjiZapis then begin
      for i := 1 to ZadnjiZapis do begin
        j := PrviZapis;
        while Zapisi[j].Stevilka <> i do j := Zapisi[j].Naslednji;
        Pisi(p.Datoteka, Zapisi[j].Zapis);
      end; {for}
```

```
        SprostiDatoteko(p.Datoteka);
    end; {if}
end; {ShraniPaket}
```

Slabost tega programa je, da bi odpovedal pri sprejemanju kakšne zelo dolge datoteke (več kot MaxPaketov paketov). Lahko bi ga izboljšali, da bi pomnilnik za shranjene pakete zasegal in sproščal dinamično. Če hranimo pakete v seznamih, povezanih s kazalci, bi jih lahko uredili s kakšno različico zlivanja (*merge sort*). Lahko bi tudi imeli za vsako datoteko kazalce na prvih nekaj sto paketov kar v tabeli (vsak element tabele ustreza eni od prvih toliko zaporednih številk paketa, tako da teh paketov na koncu sploh ne bo treba urejati), preostale pakete (če je datoteka tako dolga, da je to potrebno) pa bi hranili v seznamu: s tem bi se pri večini datotek izognili potrebi po urejanju paketov (na primer: če imamo v tabeli prostora za tristo paketov, povprečna datoteka pa ima sto paketov, je gotovo kvečjemu tretjina datotek daljših od tristo paketov).

R1990.3.4

Algoritem ne zagotavlja medsebojnega izključevanja — oba robota se lahko hkrati znajdeta v vesoljski ladji. Oglejmo si primer, kako lahko pride do tega:

N: 84

vLadji ima vrednost 0
robot 1: preskoči zanko 2 in nastavi naVrsti := 1
preveri pogoj vLadji = 0 in vstopi v telo stavka **if**
robot 2: preskoči zanko 2 in nastavi naVrsti := 2
preveri pogoj vLadji = 0 in vstopi v telo stavka **if**
nastavi vLadji := 2
preveri (vLadji = 2) **and** (naVrsti = 2) in vstopi v ladjo
robot 1: nastavi vLadji := 1 in se vrne na začetek zanke 1
preskoči zanko 2 in nastavi naVrsti := 1
preveri (vLadji = 1) **and** (naVrsti = 1) in vstopi v ladjo

Oglejmo si še primer malo drugačnega sinhronizacijskega algoritma, ki (za razliko od tistega iz besedila naloge) uspešno preprečuje, da bi se lahko v ladji hkrati znašla oba robota. Podobno kot pri algoritmu iz besedila naloge bomo tudi tu predpostavili, da si globalne spremenljivke delita oba robota, dostop do njih pa je izveden tako, da robota ne moreta oba hkrati dostopati do ene in iste spremenljivke.

```
const Jaz = ...;
var Hocem: array [1..2] of boolean value [1..2: false];
    naVrsti: 1..2 value 1;

procedure Raztovarjanje;
begin
```

```
  repeat
    Hocem[Jaz] := true;
    naVrsti := 3 − Jaz;
    while Hocem[3 − Jaz] and (naVrsti = 3 − Jaz) do begin end;
    stopi v ladjo, poberi zaboj, izstopi iz ladje;
    Hocem[Jaz] := false;
  until LadjaPrazna;
end; {Raztovarjanje}
```

Robot n lahko prek vrednosti Hocem[n] pove, da bi bil rad zdaj v ladji (oz. bi rad vstopil vanjo, če je zaenkrat še zunaj). Če hočeta v ladjo oba robota, bo vanjo vstopil tisti, čigar oznako vsebuje spremenljivka naVrsti — kateri je to, je odvisno od tega, kateremu je uspelo prej izvesti vrstico naVrsti := 3 − Jaz.

Ko hoče na primer eden od robotov stopiti v ladjo, to najprej najavi v tabeli Hocem, nato pa postavi naVrsti na oznako drugega robota in dá s tem tudi njemu možnost stopiti v ladjo. (1) Če je drugi robot takrat že v ladji, bo prvi robot čakal v svoji zanki **while**, dokler ne bo drugi ob izstopu iz ladje postavil svoje Hocem na false. (2) Če drugi robot takrat čaka na vstop (v svoji zanki **while**, bo zdaj lahko vstopil, naš prvi robot pa bo začel v svoji zanki **while** čakati na njegov izstop, tako kot pri primeru (1). (3) Če je drugi robot takrat zunaj ladje in si še ne prizadeva vstopiti vanjo, je njegov Hocem že zdaj false in bo lahko naš prvi robot takoj vstopil. (4) Če pa je drugi robot sicer že postavil svojo Hocem na true, ni pa še izvedel druge vrstice, s katero bi postavil naVrsti na oznako prvega robota, bo naš prvi robot počakal, dokler drugi ne izvede še tiste druge vrstice, nato pa bo drugi začel čakati v svoji zanki **while**, naš prvi robot pa bo v *svoji* zanki **while** opazil, da pogoj ni več izpolnjen, in bo vstopil v ladjo.[18]

[18] O tem, da ta algoritem res prepreči, da bi se oba robota hkrati znašla v ladji, se lahko prepričamo tudi tako, da sistematično pregledamo vsa možna stanja celotnega sistema. Stanje v tem primeru obsega vrednosti vseh spremenljivk in za vsakega od robotov še trenutno mesto izvajanja v podprogramu Raztovarjanje. Opisani algoritem je predlagal Gary Peterson leta 1981; gl. Wikipedijo *s. v.* Peterson's algorithm in njegov članek *Myths about the mutual exclusion problem*, Information Processing Letters, 12(3):115–116, June 1981. Glej tudi Y. Bar-David, G. Taubenfeld, *Automatic discovery of mutual exclusion algorithms*, Proc. DISC 2003; v tem članku avtorja opisujeta, kako sta s sistematičnim generiranjem in preverjanjem vseh možnih algoritmov (do določene dolžine) našla še veliko podobnih sinhronizacijskih postopkov, ki delujejo prav tako dobro kot Petersenov.

15. republiško tekmovanje v znanju računalništva (1991)

NALOGE ZA PRVO SKUPINO

1991.1.1

R: 115

Mojster Turing je izumil prav čuden stroj. Stroj namreč nima običajnega pomnilnika. Dela z magnetnim trakom, s katerega lahko bere, pa tudi piše lahko po njem. To počne s posebno bralno-pisalno glavo, ki jo premika naprej in nazaj po posameznih zapisih. Zapisi imajo lahko vrednost „0“, „1“ ali „P“, kar pomeni prazno. Na začetku dela je glava na začetku traku (trak ima začetek, konca pa ne — v tisto smer je neskončno dolg).

Stroj upravljaš z naslednjimi ukazi:

PremakniGlavo(Smer: SmerT) premakne glavo v smer Smer (naprej ali nazaj);

PreberiZnak: ZnakT prebere znak s traku, pri čemer se glava ne premakne;

IzpisiZnak(Znak: ZnakT) napiše znak na trak, pri čemer se glava ne premakne.

Turing je pospravljal svojo sobo in našel naslednji program. Žal se ne more spomniti, **kaj program naredi**. Ali mu lahko pomagaš?

```
program TuringovStroj(Input, Output);

  type SmerT = (eNazaj, eNaprej);
       ZnakT = (e0, e1, eP);
  procedure PremakniGlavo(Smer: SmerT); external;
  function PreberiZnak: ZnakT; external;
  procedure IzpisiZnak(Znak: ZnakT); external;

begin
  while PreberiZnak <> eP do
    if PreberiZnak = e0 then begin
      repeat PremakniGlavo(eNaprej)
      until (PreberiZnak = e1) or (PreberiZnak = eP);
      if PreberiZnak = eP then begin
        PremakniGlavo(eNazaj);
        IzpisiZnak(eP);
      end else begin
        PremakniGlavo(eNazaj);
        IzpisiZnak(e1);
        PremakniGlavo(eNaprej);
      end; {if}
```

```
    end else begin
      repeat PremakniGlavo(eNaprej)
      until (PreberiZnak = e0) or (PreberiZnak = eP);
      if PreberiZnak = eP then begin
        PremakniGlavo(eNazaj);
        IzpisiZnak(eP);
      end else begin
        PremakniGlavo(eNazaj);
        IzpisiZnak(e0);
        PremakniGlavo(eNaprej);
      end; {if}
    end; {if}
end. {TuringovStroj}
```

R: 115

1991.1.2

Program za igranje šaha (med človekom in računalnikom) bi radi dopolnili v študijski šahovski program, ki bi omogočal igralcu vračanje svojih potez in vlečenje drugačnih. Program naj bi hranil zadnjih deset šahistovih potez. Šahist naj bi imel tudi možnost, da odigrane in nato vrnjene poteze ponovno odigra, ne da bi moral iste poteze ponovno vpisovati. Ponoviti je možno le poteze, ki so bile tik pred tem vrnjene — po vpisani novi (drugačni) potezi ponovitev ni več možna.

```
type UkazT = (Vrni, Ponovi, Vleci, Konec);
     PotezaT = ...;
```

Šahistovo naslednjo željo ugotovi podprogram:

```
procedure BeriUkaz(var Ukaz: UkazT; var Poteza: PotezaT); external;
```

pri tem je Ukaz lahko:

Vrni	zahtevano je vračanje poteze, če je še kakšna shranjena;
Ponovi	zahtevana je ponovitev poteze, če je bila pred tem vrnjena;
Vleci	igranje nove poteze; pri tem je v spremenljivki Poteza zapisana nova šahistova poteza;
Konec	konec partije.

Pri ukazih Vrni in Ponovi vrednost spremenljivke Poteza ni definirana.

Na voljo imaš še dva podprograma:

```
procedure IgrajPotezo(Poteza: PotezaT); external;
```

odigra podano šahistovo potezo (in računalnikovo protipotezo),

```
procedure VrniPotezo(Poteza: PotezaT); external;
```

pa opravi obratno potezo od podane poteze — vrne šahovnico v stanje pred odigrano navedeno šahistovo potezo. Podprogram deluje pravilno le v primeru, da vračamo poteze v obratnem vrstnem redu, kot so bile pred tem odigrane.

Napiši program!

1991.1.3

Napiši program, ki prepiše izvorno kodo pascalskega programa z vhoda na izhod in pri tem uporabi za izpis rezerviranih besed in komentarjev pisave, različne od pisave preostalega besedila. Rezervirane besede so tista zaporedja črk, za katera funkcija RezervBeseda vrne true. Komentarji se prično z znakom „{“ in končajo z „}“, vmes pa lahko nastopajo poljubni znaki razen znaka „}“. V besedilu lahko nastopajo tudi nizi, ki se prično in končajo z znakom „’“, vmes pa se lahko pojavi poljuben znak razen „’“.

R: 117

Na voljo imaš naslednje podprograme:

PreberiZnak(c) vrne naslednji znak z vhoda v spremenljivko c.

IzpisiZnak(c) izpiše znak c na izhod.

SpremeniPisavo(Pisava) spremeni obliko izpisa na izhodu. Spremenljivka Pisava lahko zavzame vrednosti:

- Krepko (za izpis rezerviranih besed),
- Nagnjeno (za izpis komentarjev),
- Normalno (za izpis vsega ostalega besedila).

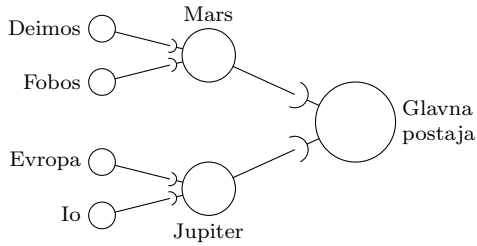
RezervBeseda(Niz, NizL) vrne vrednost true, če je niz dolžine NizL rezervirana beseda, in false, če ni.[19]

Kot približen primer izpisa lahko služi program iz prve naloge.

1991.1.4

V osončju so medplanetarno pomembne odločitve sprejete, če zanje glasujejo predstavniki vseh štirih naseljenih lunic. Med glasovanjem sprejme vesoljska postaja *Mars* glasova z lun *Deimos* in *Fobos* ter pošlje v glavno postajo skupen glas obeh lun (DA, če sta obe ZA, sicer NE). Enako stori postaja *Jupiter* z glasovoma z lun *Evropa* in *Io*.

R: 118



[19] Mišljeno je, da je parameter Niz tipa **packed array** [1..MaxDolz] **of** char (ne pa npr. string) in zato potrebuje eksplicitен podatek o dolžini besede.

V glavni postaji iz obeh prejetih glasov na enak način ugotovijo, ali je bila odločitev sprejeta (če *Mars* in *Jupiter* oba pošljeta glas DA, je odločitev sprejeta, sicer pa ne). Nekoč se je zgodilo, da se je na eni od treh postaj pokvaril eden od sprejemnikov, tako da je bilo videti, da sprejema glas ZA, ne glede na to, kaj je v resnici sprejemal. Da bi takšno napako v bodoče pravočasno odkrili, so sklenili, da bodo odslej pred pravimi glasovanji opravili nekaj poskusnih glasovanj s predpisanimi izjavami (glasovi DA in NE). Napiši **zaporedje poskusnih glasovanj** (za vsako glasovanje 4 predpisane izjave z lun), s katerimi je vedno mogoče odkriti, kateri od šestih sprejemnikov se je pokvaril, ali pa dejstvo, da so vsi brezhibni. (Zanima nas le okvara, pri kateri natanko en sprejemnik trdi, da sprejema DA ne glede na to, kar v resnici sprejema; z drugačnimi okvarami se tu ne ubadamo.)

NALOGE ZA DRUGO SKUPINO

R: 120

1991.2.1

Na nekaj primerih **izračunaj**, kaj program izpiše, in **razloži** način delovanja (postopek)!

```
program KajIzpisem(Input, Output);
type
  Stevilo = 0..255;

var
  a, b: Stevilo;
  i, k, Ham: integer;

  procedure Podprogram(p1: Stevilo; var p2: Stevilo; i: integer);
  var
    b1: integer;
    b2: integer;
  begin
    b1 := (p1 div i) mod 2;
    b2 := (p2 div i) mod 2;
    p2 := (p2 mod i) + Abs(b2 - b1) * i + (p2 div (2 * i)) * (2 * i);
  end; {Podprogram}

begin
  ReadLn(a, b);
  i := 1;
  for k := 1 to 8 do begin
    Podprogram(a, b, i);
    i := 2 * i;
  end; {for}
  Ham := 0;
  for k := 8 downto 1 do begin
    i := i div 2;
```

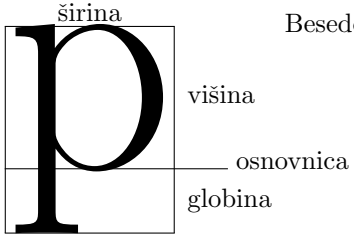
```
    if Odd(b div i) then Ham := Ham + 1;    { Odd(n) pove, če je število n liho. }
  end; {for}
  WriteLn(Ham);
end.
```

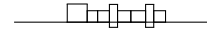
1991.2.2

Program za stavljenje besedil TEX obravnava vsako črko, kot da je zaprta v pravokotnik. Črke ene vrstice so nanizane na črto, ki jo imenujemo *osnovnica*. R: 121

Vsak pravokotnik opišemo z

- globino, to je razdalja med osnovnico in spodnjim robom pravokotnika;
- višino, to je razdalja med osnovnico in zgornjim robom pravokotnika;
- širino.



Besedo „Urejanje“ TEX vidi kot: 

Vrstice nanizanih pravokotnikov TEX razmakne na primerno razdaljo in jih pripravi za izpis. Včasih pa želimo vrstice stisniti čimbolj skupaj, vendar tako, da se pravokotniki ne prekrivajo. **Napiši podprogram**, ki izračuna najmanjšo razdaljo med dvema neprazna mima vrstama (razdaljo med njunima osnovnicama), pri kateri se pravokotniki ne prekrivajo.[20] Podatki naj bodo predstavljeni v obliki:

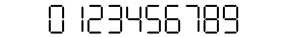
```
const VrstaM = ...;    { Največje dovoljeno število pravokotnikov v vrsti. }
type SkatlaT = record
                 Visina, Globina, Sirina: integer;
               end;
     VrstaT = array [1..VrstaM] of SkatlaT;
```

Argumenti, ki jih dobi podprogram, so:

- Vrsta1, Vrsta2 tipa VrstaT in
- Dolzina1, Dolzina2 (število pravokotnikov v vrsti) tipa integer.

[20] Zanimivo (le malo težjo) različico te naloge dobimo, če ne zahtevamo, da se obe vrstici začneta pri isti x-koordinati, ampak dovolimo, da se ena vrstica zamika malo v levo ali desno glede na drugo.

R: 126 **1991.2.3** Želimo narediti program, ki bo bral podatke z izpolnjenih formularjev. Številke na formularjih so natisnjene s posebnim tiskalnikom, ki jih izpisuje tako, kot to počne večina kalkulatorjev.



Program za razpoznavanje podatkov najprej vsako prebrano številko razbije na sedem delov (ki ustrezajo sedmim segmentom, iz katerih so sestavljene številke). Nato se za vsakega od teh delov odloči, ali je na formularju tam črnilo ali praznina (to stori z nekim nam neznanim postopkom). Pri tem se včasih tudi zmoti. Zato kliče naš program, ki ta delno pravilni opis primerja z opisi vseh številk in pove, kateri je najbolj podoben. Če se ne more odločiti, naj vrne presledek. **Napiši podprogram** za primerjanje znakov.

Oblika znakov (številk), ki se lahko pojavijo v formularju, je shranjena v tabeli OblikeZnakov.

```
type Znak = array [1..7] of boolean;
var OblikeZnakov: array ['0'..'9'] of Znak;
```

Naš podprogram pa dobi en argument tipa Znak.

R: 127 **1991.2.4** Med dvema računalnikoma želimo prenašati poljubna zaporedja bitov, grupiranih v pakete. Na voljo imamo serijsko linijo (prenaša se en bit naenkrat), dolžine posameznih paketov ne poznamo vnaprej in tudi nimamo dovolj pomnilnika, da bi lahko shranili ves paket.

Med paketi dodamo določemo (fiksno dogovorjeno) zaporedje bitov, ki ga bo sprejemni računalnik vedno razpoznal kot mejo med paketi.

Predlagaj postopek, ki bo omogočal prenos poljubnega zaporedja bitov in ki bo omogočal, da na sprejemni strani zanesljivo določimo začetek in konec vsakega paketa. Postopek lahko seveda dodaja svoje bite tudi med bite paketa, vendar naj bo skupno število dodanih bitov majhno v primerjavi s številom koristnih bitov v paketu.

NALOGE ZA TRETJO SKUPINO

R: 128 **1991.3.1** Ugotovi, **kaj počne** naslednji podprogram. **Preizkusi** ga najprej na primeru.

```
type byte = 0..255; { osembitna nepredznačena števila }
function KajStorim(a, b: byte): byte;
var
  Stevec: byte;
  Rezultat: byte;
  Premik: byte;
```

```
begin
  Rezultat := 0;
  Premik := 0;
  for Stevec := 1 to 8 do begin
    Premik := 2 * Premik + (a div 128);
    a := (a mod 128) * 2;
    Rezultat := 2 * Rezultat;
    if b <= Premik then begin
      Premik := Premik − b;
      Rezultat := Rezultat + 1;
    end; {if}
  end; {for}
  KajStorim := Rezultat;
end; {KajStorim}
```

1991.3.2

Napiši podprogram Parse, ki prebere izraz in ga izpiše z oklepaji glede na prioriteto operatorjev. Izraz sestavljajo simboli ⟨*operator*⟩, ⟨*člen*⟩ in ⟨*konec*⟩ (simbol za konec izraza). Izraz je zaporedje simbolov:

R: 129

$$\langle člen\rangle\ \langle operator\rangle\ \langle člen\rangle\ \langle operator\rangle \cdots \langle člen\rangle\ \langle konec\rangle$$

Na razpolago imamo naštevni tip SymDef, ki definira simbole:

```
type SymDef = (sClen, sOp, sKonec);
```

Podprogram GetSym vpiše v spremenljivko Sym tip naslednjega simbola v zaporedju. Če ima spremenljivka Sym vrednost sClen, se v spremenljivki Ch po vrnitvi iz GetSym nahaja znak, ki označuje člen; če ima spremenljivka Sym vrednost sOp, se v spremenljivki OpPri nahaja število, ki označuje prioriteto operatorja (nižja številka označuje višjo prioriteto), v spremenljivki Ch pa znak, ki označuje operator. Če bi bilo možno postaviti oklepaje na več enakovrednih načinov (operatorji z enako prioriteto), potem so vse inačice enako pravilne — izberemo poljubno.

Primer: če v izrazih uporabljamo operatorje s prioritetami:[21]

$+$	$-$	$*$	$\uparrow$
3	3	2	1

[21] Iz prikazanega primera je videti, kot da so prioritete vedno majhna naravna števila. Vendar pa, če smo pri sestavljanju rešitve previdni, gre tudi brez te dodatne omejitve — za rešitev je dovolj že, če znamo prioritete samo primerjati med sabo in povedati, katera je višja in katera nižja.

Mimogrede, zanimiv problem dobimo tudi, če zahtevo te naloge obrnemo in poskušamo odstraniti iz izraza vse odvečne oklepaje, ne da bi mu pri tem spremenili pomen. Gl. npr. nalogo E z ACMovega srednjeevropskega študentskega tekmovanja v programiranju CERC 2000 (Praga, 11. nov. 2000).

mora podprogram Parse za vsakega od naslednjih vhodnih izrazov izpisati takšen izhodni izraz:

vhodni izraz	izhodni izraz
$a + b$	$(a + b)$
$a + b * c$	$(a + (b * c))$
$a * b + c$	$((a * b) + c)$
$a + b \uparrow c * d$	$(a + ((b \uparrow c) * d))$
$a + b + c$	$((a + b) + c)$ ali pa $(a + (b + c))$

Uporabljaj naslednje zgoraj opisane spremenljivke in podprograme:

```
var Sym: SymDef;
    Ch: char;
    OpPri: integer;

procedure GetSym; external;
```

R: 131

1991.3.3

Napiši podprogram, ki dobi na vhodu izvorni niz znakov in vzorec ("wild-card") ter njuni dolžini. Podprogram naj vrne true, če izvorni niz ustreza vzorcu. Vzorec sestavljajo znaki, ki morajo biti enaki znakom v izvornem nizu; znaka `*` in `?` v vzorcu imata poseben pomen: `*` pomeni nič, enega ali poljubno znakov v izvornem nizu; `?` pomeni poljuben znak. Predpostaviš lahko, da se znaka `*` in `?` ne pojavita v izvornem nizu znakov.

Primeri:

`'2124123124'`	ustreza	`'*123*124*'`
`'1234567'`	ustreza	`'12*34*7'`
`'111222333'`	ustreza	`'1*2*3'`
`'12321'`	ne ustreza	`'123?*2'`

R: 133

1991.3.4

Pri prenašanju podatkov med dvema računalnikoma lahko prihaja do motenj na komunikacijski zvezi. Da kljub temu zagotovimo zanesljiv prenos, običajno podatke (npr. zaporedje znakov) grupiramo v bloke (npr. po 256 znakov), vsak blok pa opremimo z dodatno informacijo, ki omogoča prejemnemu računalniku ugotoviti morebitne napake v bloku. Prejemni računalnik lahko oddajnemu potrdi pravilen sprejem vsakega bloka ali pa zahteva njegovo ponovitev.

Pravilom (algoritmu), po katerih se ravnata oba računalnika pri medsebojni komunikaciji in opisu blokov (dolžina bloka, dodatna informacija) pravimo *protokol*.

Če je oddaljenost med računalnikoma relativno majhna in hitrost prenašanja podatkov med njima ne posebno velika, potem se sprotno potrjevanje (ali zahtevanje ponovitve) vsakega bloka obnese.

Radi bi prenašali podatke od računalnika v Evropi do računalnika v Ameriki. Pošta nam je zagotovila hitro komunikacijsko zvezo (približno milijon bitov na sekundo) prek geostacionarnega telekomunikacijskega satelita (oddaljenega od površine Zemlje slabih 36 000 km čas potovanja elektromagnetnega valovanja do satelita in nazaj je približno 1/4 sekunde).

Ali bo naš preprosti protokol še vedno učinkovit? Zakaj? **Predlagaj učinkovitejši protokol!** Premisli tudi, ali zanesljivost prenosa (verjetnost napake / znak) vpliva na optimalno dolžino blokov.

REŠITVE NALOG ZA PRVO SKUPINO

R1991.1.1

N: 107

Program za opisani stroj prestavi elemente traku (ničle in enice) za eno mesto proti začetku traku, prvi element pa zavrže. To opravilo izvede tako, da poišče meje med zaporedji ničel in zaporedji enic ter prestavi le ustrezen element na meji. Ko naleti na „P“, napiše presledek tudi prek zadnjega nepraznega elementa, s čimer skrajša pomaknjeno zaporedje za ena.

Če bi bilo na traku ob začetku izvajanja programa več zaporedij ničel in/ali enic, vmes pa presledki, bi program pobrisal prvi znak le iz prve skupine ne-presledkov (ter to skupino premaknil za eno mesto proti začetku), ostale skupine pa bi ostale nedotaknjene. Med prvo in drugo skupino ne-presledkov bi bil po novem en presledek več kot ob začetku izvajanja programa.

Turingov stroj se imenuje po matematiku Alanu Turingu, ki ga je predlagal leta 1936 kot formalizacijo, s katero bi si lahko pomagali pri razmišljanju o tem, kaj je mogoče izračunati „efektivno“ (strojno, mehansko, algoritemsko). Obstaja še več različic Turingovega stroja, ki se od tu opisanega ločijo npr. po tem, da dovolijo več vrst zapisov (ne le „0“, „1“ in „P“ kot v naši nalogi), da imajo trak neskončen v obe smeri, da imajo več trakov ipd. Izkaže pa se, da takšne razširitve ne vplivajo bistveno na zmogljivosti stroja. Dobro je, da je stroj čim preprostejši, ker je potem o njegovem delovanju lažje razmišljati z matematičnimi orodji. Zato Turingov stroj tudi nima pomnilnika z neposrednim dostopom, ampak lahko podatke hrani le na traku (in zato tudi program, podan v naši nalogi, ne uporablja spremenljivk); edina izjema je ta, da ima stroj vedno pri roki podatek o tem, kateri stavek programa se trenutno izvaja.[22]

R1991.1.2

N: 108

Pomagali si bomo s tabelo potez (Poteza), ki jo bomo uporabljali kot neke vrste sklad (na njem je PotezaL potez). V spremenljivki Zadnja si zapomnimo, do kod smo se že vrnili z vračanjem

[22] Več o Turingovih strojih najdemo v učbenikih teoretičnega računalništva, npr. M. Sipser: *Introduction to the Theory of Computation*, 1996; J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, 1979.

potez. Če je treba potegniti novo potezo, vse za zadnjo pozabimo (PotezaL := Zadnja) in nato dodamo novo potezo na vrh sklada. Vračanje in ponavljanje izvedemo preprosto tako, da zmanjšujemo in povečujemo spremenljivko Zadnja (in pazimo, da ostane na intervalu 1..PotezaL).

```
program Vracanje(Input, Output);
type
  UkazT = (Vrni, Ponovi, Vleci, Konec);
  PotezaT = record ... end;

  procedure BeriUkaz(var Ukaz: UkazT; var Poteza: PotezaT); external;
  procedure IgrajPotezo(Poteza: PotezaT); external;
  procedure VrniPotezo(Poteza: PotezaT); external;

const PotezaM = 10; { število shranjenih potez }
var
  Poteza: array [1..PotezaM] of PotezaT; { shranjene poteze }
  PotezaL: integer;      { število shranjenih potez }
  Zadnja: integer;       { številka zadnje veljavne poteze v shrambi }
  { Pri vlečenju novih potez velja Zadnja = PotezaL,
    sicer pa se lahko spremenljivka Zadnja vrača po shranjenih potezah. }
  Ukaz: UkazT;           { šahistov ukaz }
  p: PotezaT;            { šahistova poteza, če je Ukaz = Vleci }
  i: integer;
begin
  PotezaL := 0; Zadnja := 0;
  repeat
    BeriUkaz(Ukaz, p);
    case Ukaz of
      Vrni:
        if Zadnja > 0 then
          begin VrniPotezo(Poteza[Zadnja]); Zadnja := Zadnja - 1 end;
      Ponovi:
        if Zadnja < PotezaL then
          begin Zadnja := Zadnja + 1; IgrajPotezo(Poteza[Zadnja]) end;
      Vleci:
        begin
          PotezaL := Zadnja; { Zavržemo morebitne še shranjene novejše poteze. }
          if PotezaL < PotezaM then { V shrambi je prostor. }
            PotezaL := PotezaL + 1
          else { Ni prostora, zavržemo najstarejšo shranjeno potezo. }
            for i := 2 to PotezaL do Poteza[i - 1] := Poteza[i];
          Poteza[PotezaL] := p; Zadnja := PotezaL; IgrajPotezo(p);
        end;
    end; {case}
  until Ukaz = Konec;
end. {Vracanje}
```

R1991.1.3

N: 109

Vhodne podatke beremo znak za znakom. Z zastavicama Niz in Komentar si zapomnimo, če smo trenutno v nizu ali v komentarju; ti dve zastavici spreminjamo, ko pridemo do znaka za začetek ali konec niza ali komentarja. Na začetku in koncu komentarja tudi spremenimo pisavo. Ko smo v nizu ali komentarju, prebrane znake nespremenjene tudi izpisujemo. Če nismo niti v nizu niti v komentarju, pa naletimo na zaporedje črk, jih dodajamo v niz Beseda in jih še ne izpisujemo; ko se črke končajo, preverimo, ali je nastala beseda rezervirana in jo v tem primeru izpišemo krepko, sicer pa jo izpišemo v normalni pisavi. V vsakem primeru niz Beseda potem spraznimo, da bo pripravljen na naslednje zaporedje črk, ko bomo spet prišli do kakšnega. Za praznjenje niza Beseda poskrbimo tudi, ko pridemo do konca vhoda, saj je lahko takrat v nizu Beseda še nekaj črk, ki jih še nismo izpisali (res pa je, da pri sintaktično pravilnih pascalskih programih do tega najbrž ne bo prišlo, saj se končajo s piko (za besedo **end**) in mogoče še kakšnim komentarjem). Paziti moramo še na možnost, da je neko zaporedje črk predolgo za tabelo Beseda; takšno gotovo ne bo rezervirana beseda (saj se da vnaprej ugotoviti, kako dolga je najdaljša rezervirana beseda), zato jo lahko izpisujemo kar sproti.

```
program Izpis(Input, Output);
const MaxDolz = 30; { Največja možna dolžina rezervirane besede. }
type
  PisavaT = (Normalno, Nagnjeno, Krepko);
  BesedaT = packed array [1..MaxDolz] of char;
var
  c: char;
  Beseda: BesedaT;
  BesedaL: integer;
  i: integer;
  Niz, Komentar, Rez: boolean;

  procedure PreberiZnak(var c: char); external;
  procedure IzpisiZnak(c: char); external;
  procedure SpremeniPisavo(Pisava: PisavaT); external;
  function RezervBeseda(b: BesedaT; bL: integer): boolean; external;

  { Ta podprogram kličemo, ko pridemo do konca besede. Če je to zelo dolga beseda,
    smo jo izpisovali že sproti in je zdaj ni treba, sicer pa moramo preveriti,
    če je to mogoče ena od rezerviranih besed. }
  procedure KonecBesede;
  begin
    if BesedaL <= MaxDolz then begin
      Rez := RezervBeseda(Beseda, BesedaL);
      if Rez then SpremeniPisavo(Krepko);
      for i := 1 to BesedaL do IzpisiZnak(Beseda[i]);
      if Rez then SpremeniPisavo(Normalno);
```

```
      end; {if}
      BesedaL := 0;
    end; {KonecBesede}

begin {Izpis}
  BesedaL := 0; Niz := false; Komentar := false; SpremeniPisavo(Normalno);
  while not Eof do begin
    PreberiZnak(c);
    if (BesedaL > 0) and not (c in ['A'..'Z', 'a'..'z']) then
      KonecBesede;
    if Niz then begin
      if c = '''' then Niz := false;
      IzpisiZnak(c);
    end else if Komentar then begin
      IzpisiZnak(c);
      if c = '}' then
        begin Komentar := false; SpremeniPisavo(Normalno) end;
    end else if c = '{' then begin
      Komentar := true; SpremeniPisavo(Nagnjeno); IzpisiZnak(c);
    end else if c = '''' then begin
      IzpisiZnak(c); Niz := true;
    end else if c in ['A'..'Z', 'a'..'z', '0'..'9'] then begin
      if BesedaL = MaxDolz then begin
        { Trenutna beseda bo dolga vsaj MaxDolz + 1, torej to ni rezervirana
          beseda. Izpišimo, kar smo že shranili, ostalo pa bomo izpisovali sproti.
          Spremenljivka BesedaL bo ostala pri vrednosti MaxDolz + 1. }
        for i := 1 to BesedaL do IzpisiZnak(Beseda[i]);
        BesedaL := BesedaL + 1;
      end; {if}
      if BesedaL > MaxDolz then IzpisiZnak(c)
      else begin BesedaL := BesedaL + 1; Beseda[BesedaL] := c end;
    end else IzpisiZnak(c);
  end; {while}
  if BesedaL > 0 then KonecBesede;
end. {Izpis}
```

N: 109

R1991.1.4

Minimalno število poskusnih glasovanj je štiri. V nalogi se skriva testiranje logičnega (digitalnega) vezja, sestavljenega iz treh vrat IN (kot kaže leva slika na str. 119), kjer se je na enih od vrat en vhodni signal „zataknil“ na 1. Sprejemnike na Marsu, Jupitru in glavni postaji označimo z $x_1, \ldots, x_6$. Pri vsakem poskusnem glasovanju določimo, kakšne vrednosti pridejo (s posameznih naseljenih lun) do vhodov $x_1, \ldots, x_4$, nato pa pogledamo, kakšen je izhod vezja.

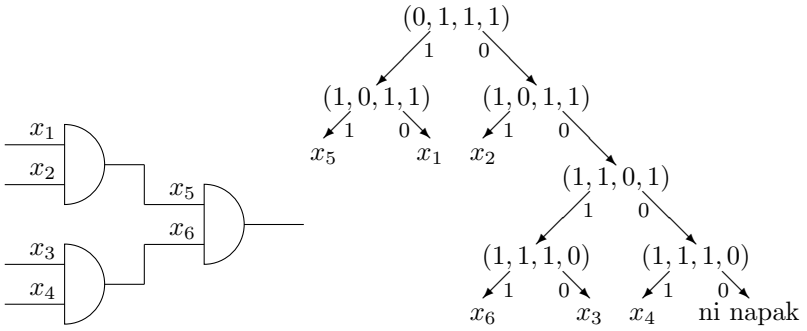
Do primernega zaporedja poskusnih glasovanj lahko pridemo iz pravilnostnih tabel za vseh šest logičnih funkcij, ki opisujejo možna vezja, ki jih dobimo iz prikazanega, če se eden od vhodov „zatakne“ na 1. Lahko

pa si pomagamo tudi s preprostim razmislekom. Če na primer postavimo $x_1 = 0, x_2 = x_3 = x_4 = 1$, izhod iz vezja pa je vendarle 1, pomeni, da je $x_5 = 1$, torej je bodisi x_5 zataknjen na 1 ali pa x_5 deluje normalno, vendar je že do njega prišla vrednost 1, kar pa pomeni, da je x_1 zataknjen na 1. Ti dve možnosti lahko ločimo tako, da postavimo $x_1 = 1, x_2 = 0$: ker x_2 gotovo ni zataknjen, pride zdaj do x_5 zagotovo vrednost 0; če je izhod iz vezja še vedno 1, pomeni, da je zataknjen x_5, sicer pa mora biti zataknjen x_1.

Če pa je prvi test dal rezultat 0, vemo, da je z x_1 in x_5 vse v redu. Zdaj lahko postavimo $x_1 = 1, x_2 = 0$ in če je rezultat vezja zdaj 1, pomeni, da je zataknjen x_2 (druga možnost bi bila, da bi bil zataknjen x_5, kar pa že vemo, da ni res).

Zdaj lahko enako razmišljanje ponovimo za drugo polovico vezja, torej počnemo enake stvari kot v prejšnjih dveh odstavkih, le namesto x_5 si mislimo x_6, namesto x_1 in x_2 pa x_3 in x_4. Če tudi zdaj ne odkrijemo zataknjenega vhoda, pomeni, da so vsi vhodi dobri.

Primerno zaporedje poskusnih glasovanj, s katerim lahko testiramo celoten sistem, je torej: 0111, 1011, 1101, 1110. Pri tem smo vsako glasovanje predstavili z zaporedjem (vektorjem) štirih bitov, ki povedo, kaj pride do vhodov $x_1, \ldots, x_4$. Celoten postopek testiranja lahko zapišemo tudi v obliki odločitvenega drevesa. Vsako notranje vozlišče predstavlja eno poskusno glasovanje; v odvisnosti od rezultata, ki ga ugotovi glavna postaja, se moramo po tem glasovanju premakniti v enega od obeh otrok trenutnega vozlišča. Ko pridemo do lista, pa že lahko točno ugotovimo, kateri vhod je pokvarjen.



Prepričajmo se še o tem, da ni mogoče sestaviti odločitvenega drevesa, ki bi vedno odkrilo zataknjeni vhod (ali ugotovilo, da vsi delujejo pravilno) z največ tremi glasovanji. Očitno je, da vektorjev 1111, 0101, 0110, 1001, 1010 in tistih s tremi ali štirimi ničlami nima smisla dajati na vhode, ker je rezultat neodvisen od tega, ali je kak vhod zataknjen (in kateri). Za prvo glasovanje nam ostaneta v bistvu le dve različni možnosti: vhod z eno ničlo (eden od 0111, 1011, 1101 in 1110) in tak z dvema (0011 ali pa 1100). Če se odločimo za drugo možnost, npr. za $x_1 = x_2 = 0$, $x_3 = x_4 = 1$, nam rezultat vezja pove, ali je x_5 zataknjen

(če je izhod vezja 1) ali ne (če je izhod vezja 0). Če izvemo, da ni zataknjen, nam ostane še šest možnih ugotovitev (pet možnih pokvarjenih vhodov in še možnost, da je vse v redu), med katerimi bi morali zdaj ločiti v naslednjih dveh glasovanjih. Toda z dvema glasovanjema lahko ločimo le štiri možnosti. Pri vhodu $x_1 = x_2 = 1$, $x_3 = x_4 = 0$ velja simetričen razmislek. Torej, če se hočemo izogniti potrebi po štirih glasovanjih, moramo pri prvem glasovanju postaviti na 0 le enega od x_1, x_2, x_3, x_4. Toda zdaj poteka glasovanje na način, kot smo ga opisali zgoraj: če dobimo na izhodu 1, bomo z enim dodatnim glasovanjem ugotovili, kateri vhod je zataknjen, če pa dobimo 0, bomo zdaj za dva vedeli, da sta dobra (na primer x_1 in x_5 pri odločitvenem drevesu na sliki), še vedno pa nam ostane pet možnosti (štirje možni pokvarjeni vhodi in možnost, da je vse v redu), torej spet ne bomo mogli ločiti vseh s samo dvema glasovanjema. Tako torej vidimo, da glasovanj ne moremo razporediti tako, da bi vedno preizkusili vezje s tremi ali manj glasovanji.

Prikazano odločitveno drevo bi odkrilo tudi primere, ko je zataknjenih več vhodov, le da se tedaj ne da vedno samo z opazovanjem izhoda celega vezja ugotoviti, kateri so zataknjeni. Na primer, če sta zataknjena x_1 in x_2 obenem, je izhod vezja vedno enak, kot če bi bil zataknjen vhod x_5.

Če bi obstajala tudi možnost, da se eden od vhodov zatakne na 0, medtem ko ostali delujejo pravilno, bi jo lahko odkrili tako, da bi postavili $x_1 = x_2 = x_3 = x_4 = 1$; če je kateri od vhodov zataknjen na 0, bo izhod iz vezja 0 namesto 1.

REŠITVE NALOG ZA DRUGO SKUPINO

N: 110

R1991.2.1

Program izpiše, v koliko istoležnih bitih se razlikujeta dvojiška zapisa prebranih števil.

Ob klicih podprograma Podprogram med izvajanjem prve zanke po k je i vedno enak 2^{k-1}, torej je potenca števila 2. Zato izraz p1 **div** i zamakne število p1 za $k-1$ mest v desno, (p1 **div** i) **mod** 2 pa izlušči bit $k-1$ števila p1 (b1 dobi vrednost 1, če je bil bit $k-1$ v p1 prižgan, sicer pa 0). Podobno v b2 pripravimo istoležni bit števila p2. Vrednost Abs(b2 − b1) je zdaj enaka 0, če sta bila oba bita prižgana ali oba ugasnjena, sicer pa ima vrednost 1: to je torej ravno b1 **xor** b2. Zdaj bi radi ta rezultat shranili v bit $k-1$ spremenljivke b2 (prejšnja vrednost tega bita se bo ob tem izgubila). Vrednost p2 **mod** i vsebuje spodnjih $k-1$ bitov števila p2, torej od 0 do $k-2$. Vrednost p2 **div** (2 * i) vsebuje bite od vključno k naprej, vendar so zamaknjeni na mesta od 0 naprej; če jo pomnožimo z 2 * i, pridejo ti biti spet na prvotna mesta, na nižjih mestih pa so ničle. Če zdaj to dvoje seštejemo, se rezultat od prvotne vrednosti p2 razlikuje le po tem, da ima bit $k-1$ zanesljivo ugasnjen. Ko prištejemo še Abs(b2 − b1) * i, se bo bit $k-1$ prižgal, če je bilo b1 različno od b2, sicer pa bo ostal ugasnjen. Tako smo dosegli prav to, kar smo želeli: bit $k-1$ števila p2 smo **xor** ali z istoležnim bitom števila p1.

Ko se v zanki pokliče Podprogram po enkrat za vsak bit, je skupni rezultat vsega tega prav tak, kot če bi izvedli prireditev b := a **xor** b. Druga zanka **for** prešteje, koliko je v tej vrednosti prižganih bitov. i ima na začetku vrednost 2^8, ker pa ga takoj na začetku vsake iteracije delimo z 2, ima potem vrednost 2^{k-1}. Število b **div** i zdaj vsebuje b-jeve bite od vključno bita $k-1$ naprej, le da so zamaknjeni na najbolj spodnje položaje; bit $k-1$ sam je tako zamaknjen v bit 0 in lahko preverimo, če je prižgan, preprosto tako, da preverimo, če je število b **div** i liho. Tako lahko preštejemo enice v številu b (oz. v a **xor** b za prvotni vrednosti a in b) ter s tem izvemo, pri koliko bitih se a in b razlikujeta. Ta vrednost (ki se nam med izvajanjem te zanke počasi nabira v spremenljivki Ham) se imenuje tudi Hammingova razdalja med bitnima nizoma a in b.

R1991.2.2

Program reši problem tako, da se premika po obeh vrstah hkrati in za vsak rob pravokotnika izračuna razdaljo do pravokotnika v drugi vrsti. Iskana najmanjša razdalja med osnovnicama je enaka največji od teh medsebojnih razdalj. N: 111

V mislih postavimo obe vrsti pravokotnikov eno nad drugo tako, da se obe vrsti začenjata na x-koordinati 0. Naj bo $g(x)$ globina zgornje vrste pravokotnikov pri koordinati x, $h(x)$ pa višina spodnje vrste pri koordinati x. Če nočemo, da se bosta vrsti pri tej x-koordinati prekrivali, morata biti njuni osnovnici razmaknjeni vsaj za $g(x) + h(x)$. To mora zdaj veljati pri vseh x, zato je najmanjši primerni razmik kar $\max_x(g(x) + h(x))$. Pri tem je dovolj, če gre x od 0 do širine krajše od obeh vrst — od tam naprej ostane le še ena vrsta in se torej ne more prekrivati z drugo.

Vrednost funkcije $g(x)$ se spremeni le, ko pridemo v zgornji vrsti do naslednjega pravokotnika, vrednost $h(x)$ pa, ko pridemo do naslednje v spodnji vrsti. Zato je dovolj, če izračunamo $g(x) + h(x)$ le pri vsakem takem x, ko se je katera od njiju spremenila, in poiščemo maksimum tako dobljenih vsot. Spodnji program se v zanki premika desno po obeh vrstah; na začetku vsake ponovitve velja, da smo v zgornji vrsti pregledali že prvih ia pravokotnikov (njihova skupna širina pa je SirinaA), v spodnji pa prvih ib pravokotnikov (s skupno širino SirinaB). Naslednja sprememba funkcije $g(x)$ torej nastopi pri $x =$ SirinaA, naslednja sprememba $h(x)$ pa pri $x =$ SirinaB. Zato pogledamo manjšo od teh dveh vrednosti; če je to prva, se premaknemo naprej po zgornji vrsti pravokotnikov (povečamo ia), sicer pa po spodnji (povečamo ib). V vsakem primeru je vrednost $g(x)$ enaka a[ia].Globina, vrednost $h(x)$ pa b[ib].Visina in njuna vsota (Razdalja) je nova kandidatka za najmanjši sprejemljivi razmik med osnovnicama (ki ga bomo pripravili v DosedanjaRazdalja).

```
const VrstaM = 999;
type
  SkatlaT = record Visina, Globina, Sirina: integer end;
  VrstaT = array [1..VrstaM] of SkatlaT;
```

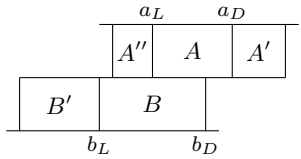
```
function NajmanjsaRazdalja(var a: VrstaT; aL: integer;
                           var b: VrstaT; bL: integer): integer;
{ Funkcija vrne najmanjšo razdaljo med sosednjima vrstama pravokotnikov,
  pri kateri se pravokotniki obeh vrstic še ne prekrivajo med seboj. }
var
  Konec: boolean;
  ia, ib: 1..VrstaM; { indeksa trenutno pregledovanega pravokotnika v obeh vrsticah }
  SirinaA, SirinaB: integer; { doslej pregledani dolžini obeh vrstic }
  StaraSirinaA: integer;
  Razdalja: integer; { globina gornjega pravokotnika + višina spodnjega
                       pravokotnika v trenutni točki }
  DosedanjaRazdalja: integer; { največja razdalja doslej }
begin
  if (aL = 0) or (bL = 0) then begin NajmanjsaRazdalja := 0; exit end;
  ia := 1; ib := 1; Konec := false;
  SirinaA := a[ia].Sirina; SirinaB := b[ib].Sirina;
  DosedanjaRazdalja := a[ia].Globina + b[ib].Visina;
  repeat
    StaraSirinaA := SirinaA;
    if SirinaA <= SirinaB then begin
      if ia >= aL then Konec := true
      else begin ia := ia + 1; SirinaA := SirinaA + a[ia].Sirina end;
    end; {if}
    if SirinaB <= StaraSirinaA then begin
      if iB >= bL then Konec := true
      else begin ib := ib + 1; SirinaB := SirinaB + b[ib].Sirina end;
    end; {if}
    if not Konec then begin
      Razdalja := a[ia].Globina + b[ib].Visina;
      if Razdalja > DosedanjaRazdalja then
        DosedanjaRazdalja := Razdalja;
    end; {if}
  until Konec;
  NajmanjsaRazdalja := DosedanjaRazdalja;
end; {NajmanjsaRazdalja}
```

Če je spodnja vrstica krajša od zgornje, bi se lahko zgodilo, da bi kak zelo globok pravokotnik na koncu zgornje vrstice štrlel v globino še pod osnovnico druge vrstice. Podobno se seveda lahko zgodi, če je zgornja krajša od spodnje in je pri koncu spodnje nek zelo visok pravokotnik. Če nas to moti (npr. ker nam zapleta nadaljnje delo, če bi hoteli zdaj pod drugo vrstico položiti še tretjo in tako naprej), bi morali krajšo vrstico podaljšati s še enim „navideznim“ pravokotnikom, ki bi imel višino in globino 0, širino pa tolikšno, da bi se širini obeh vrstic potem ujemali.

Zanimiv problem nastane, če dopustimo še premikanje vrst levo in desno. V nekaterih primerih je potem mogoče vrstici še bolj približati.

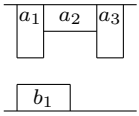
Recimo, da bo zgornja vrsta pravokotnikov pri miru in se bo začela vedno pri $x = 0$; spodnja vrsta pa naj se recimo začne pri $x = t$ za različne vrednosti t. Recimo, da se dva pravokotnika (eden iz gornje in eden iz spodnje vrstice) „stikata“, če pokrivata kakšen skupen interval x-koordinat. Množico vseh stikov, do katerih pride pri nekem konkretnem položaju spodnje vrstice (torej pri nekem konkretnem t), označimo s $S(t)$. Naj bo „razmik“ posameznega stika s, recimo mu $r(s)$, vsota globine zgornjega in višine spodnjega pravokotnika, ki se tu stikata. Če fiksiramo t (torej levi rob spodnje vrstice), je najmanjši potrebni razmik med osnovnicama obeh vrstic kar $R(t) := \max_{s \in S(t)} r(s)$. Nas zanima najmanjša vrednost $R(t)$ po vseh t z nekega intervala (verjetno je omejitev vsaj ta, da se mora interval x-koordinat, ki jih pokriva spodnja vrstica, vsaj deloma prekrivati s tistim, ki ga pokriva zgornja vrstica, saj drugače vrstici druga drugi sploh ne bi bili v napoto in bi bil problem trivialen). Kot vidimo iz formule za $R(t)$, je dovolj, če pogledamo le tiste t, pri katerih pride v množici $S(t)$ do kakšne spremembe — torej takrat, ko nastane kakšen nov stik ali pa se kakšen stari izgubi. Koristno je pregledovati t-je sistematično od manjših proti večjim, saj so tako spremembe v množici $S(t)$ vedno majhne in postopne in se ni potrebno vsakič znova ukvarjati z vsemi stiki.

Naj bo $n(s)$ najmanjša naslednja vrednost zamika t, pri kateri stik s doživi kakšno spremembo. Za to je nekaj možnosti, vzamemo pa najmanjšo med njimi. Označimo trenutno vrednost t-ja s t_0. Recimo, da zgornji pravokotnik (recimo mu A) stika s pokriva na x-osi interval $[a_L, a_D]$, spodnji (recimo mu B) pa $[b_L, b_D]$ (glej sliko). (1) Vsekakor bo prišlo do spremembe pri $t = t_0 + (a_D - b_L)$, ko bo stik s sploh izginil. (2) No, če je $b_D < a_D$, bo nastopila sprememba tudi pri $t = t_0 + (a_D - b_D)$: pojavil se bo nov stik med B in med A-jevim naslednikom v zgornji vrstici (razen če je A zadnji v zgornji vrstici). (3) Še ena možnost je, če je $b_L < a_L$ in B ni prvi pravokotnik spodnje vrstice; tedaj bo pri $t = t_0 + (a_L - b_L)$ nastal nov stik, in sicer med A-jem ter B-jevim predhodnikom v spodnji vrstici. (Pri tej možnosti je dovolj, če jo upoštevamo le takrat, ko je A prvi v svoji vrstici; pri ostalih bomo ustrezne stike dobili že zaradi točke (2).) — Vrednost $n(s)$ bo torej najmanjša izmed teh treh možnosti (oz. kolikor jih je pri tem konkretnem s v resnici mogočih).



(1) Ko se bo spodnja vrstica premaknila desno za $a_D - b_L$ enot, bo stik (A, B) izginil.
(2) Ko se bo premaknila za $a_D - b_D$ enot, bo nastal nov stik (A', B).
(3) Ko se bo premaknila za $a_L - b_L$ enot, bo nastal nov stik (A, B'). Toda če obstaja pravokotnik A'', bomo novi stik (A, B') opazili tudi pri delu s stikom (A'', B').

Ko smo torej končali z delom pri trenutnem t, je naslednji, s katerim se bo treba ubadati, $t' := \min_{s \in S(t)} n(s)$. Da bomo lahko to vrednost učinkoviteje določili, je koristno vrednosti $n(s)$ za vse trenutne stike (torej vse $s \in S(t)$) hraniti v kopici, tako da je najmanjša med njimi vedno pri roki.[23] Tako torej ni težko ugotoviti, za koliko naj povečamo t; nato pa si moramo ogledati stik, ki je ta premik povzročil, in ga po potrebi zbrisati in/ali ustvariti kakšen nov stik, pač v skladu z razmislekom iz prejšnjega odstavka. Paziti moramo še na možnost, da ima več dotedanjih stikov isto vrednost $n(s)$ in moramo torej ob premiku t-ja poskrbeti za spremembe pri vseh teh stikih. S tem bomo po premiku t-ja primerno popravili oz. ažurirali tudi množico $S(t)$. Nato nas bo seveda zanimal minimalni potrebni razmik med osnovnicama vrstic pri tem t-ju, torej vrednost $\max_{s \in S(t)} r(s)$; da nam ne bo treba iti po vseh stikih, je koristno imeti tudi te vrednosti v neki kopici (in to tako, da bo največja vedno pri vrhu). Pomembno je, da od vseh sprememb, ki nastopijo pri prehodu na novi t, najprej izvedemo brisanja stikov, nato izračunamo $\max_{s \in S(t)} r(s)$ in šele nato dodamo nove stike. Brez tega namreč ne bi mogli izkoristiti nekaterih bolj tesnih sestavitev pravokotnikov (primer je na spodnji sliki: brisanje stika (a_1, b_1) nastopi pri istem t kot dodajanje stika (a_3, b_1); če torej ne pogledamo $\max_{s \in S(t)} r(s)$ med tem brisanjem in tem dodajanjem, ne bomo opazili, da lahko pravokotnik b_1 postavimo v vdolbino med a_1 in a_3 (in pod a_2)). Zato si v spodnji psevdokodi pomagamo s pomožnim seznamom D.



Kakšna je časovna zahtevnost tega postopka? Če pregledamo vse možne vrednosti t, pride vsak pravokotnik spodnje vrstice z vsakim pravokotnikom zgornje prej ali slej vsaj za nekaj časa v stik. Če je v vsaki vrstici N pravokotnikov, imamo torej vsega skupaj $O(N^2)$ stikov. Vsak stik lahko, kot smo videli ob razmišljanju o vrednosti $n(s)$, največ trikrat povzroči spremembe, torej bo od nas zahteval le konstantno število operacij v kopici; vsaka od teh operacij pa ima časovno zahtevnost $O(\log N)$, saj v vsakem trenutku obstaja le $O(N)$ stikov (če gremo po x-osi od leve proti desni, pride do novega stika natanko tam, kjer se začne ali konča kak pravokotnik v zgornji in/ali v spodnji

[23] Bralec, ki mu kopice niso domače, si lahko namesto kopice misli tudi navaden seznam (ali tabelo), v katerem je ob vsakem stiku napisana še njegova vrednost $n(s)$. Pri takem seznamu ni težko dodajati in brisati elementov, pa tudi poiskati takega z najmanjšo vrednostjo $n(s)$; težava je le v tem, da pri seznamu vsaj kakšna od teh operacij zahteva v najslabšem primeru linearen sprehod po celem seznamu. Kopica pa je podatkovna struktura, ki podpira prav takšne operacije, vendar bolj učinkovito — v času $O(\log N)$, če je N število elementov v kopici.

vrstici; in ker imamo $O(N)$ pravokotnikov, je tudi le toliko stikov). Za celoten algoritem tako porabimo $O(N^2 \log N)$ časa.

Vhod: seznama pravokotnikov $a = \langle A_1, \ldots, A_{|a|} \rangle$ in $b = \langle B_1, \ldots, B_{|b|} \rangle$.
Pomožni funkciji: $r(i,j)$ = globina pravokotnika A_i + višina B_j
$n(i,j,t)$ (opisana spodaj).

$t :=$ −skupna širina vseh pravokotnikov seznama b;
$K_n :=$ prazna kopica (manjše vrednosti bodo pri vrhu);
$K_r :=$ prazna kopica (večje vrednosti bodo pri vrhu);
dodaj stik $(1, |b|)$ v obe kopici;
$M := r(1, |b|)$;
$D :=$ prazen seznam;
while K_n ni prazna **do begin**
    $t' :=$ najmanjša vrednost iz K_n;
    if $t' > t$ **then begin**
        $M := \min\{M,$ največja vrednost iz $K_r\}$;
        dodaj vse stike iz D v obe kopici;
        $D :=$ prazen seznam; $t := t'$;
    end;
    $(i,j) :=$ stik z najmanjšo vrednostjo v K_n;
    $a_L := t +$ (skupna širina $A_1, \ldots, A_{i-1}$); $a_D := a_L +$ širina A_i;
    $b_L := t +$ (skupna širina $B_1, \ldots, B_{j-1}$); $b_D := b_L +$ širina B_j;
    if $b_D = a_D$ **and** $i < |a|$ **then** dodaj $(i+1, j)$ v D;
    if $b_L = a_L$ **and** $i = 1$ **and** $j > 1$ **then** dodaj $(i, j-1)$ v D;
    if $b_L = a_D$ **then** zbriši (i,j) iz obeh kopic
    else spremeni vrednost stika (i,j) v K_n na $n(i,j,t)$;
end;
vrni M;

Ob dodajanju novega stika v kopici uporabimo kot vrednost tega stika v kopici K_n vrednost $n(i,j,t)$, v kopici K_r pa vrednost $r(i,j)$. Vsote širin pravokotnikov od A_1 do A_{i-1} si naračunamo enkrat samkrat, na začetku, za vse i; podobno tudi za $B_1, \ldots, B_{j-1}$ za vse j.

Funkcija $n(i,j,t)$:
    izračunaj a_L, a_D, b_L, b_D enako kot zgoraj;
    $n := t + a_D - b_L$; (pri takem t bo treba ta stik zbrisati)
    if $(b_D < a_D)$ **and** $(i < |a|)$
    then $n := \min\{n, t + a_D - b_D\}$; (takrat bo treba dodati stik $(i+1, j)$)
    if $(b_L < a_L)$ **and** $(i = 1)$ **and** $(j > 1)$
    then $n := \min\{n, t + a_L - b_L\}$; (takrat t bo treba dodati stik $(i, j-1)$)
    vrni n;

Doslej opisani postopek dovoli za t vse take vrednosti, pri katerih imata obe vrstici sploh kaj skupnega (torej take, pri katerih obstaja vsaj en stik). Če pa je dovoljen le nek manjši interval t-jev, recimo od t_1 do t_2, moramo postopek malo dopolniti. Množico stikov, ki so v veljavi pri $t = t_1$, lahko določimo s postopkom, ki je čisto podoben podprogramu NajmanjsaRazdalja z začetka te rešitve. Po vsakem povečanju t-ja (v stavku $t := t'$) pa moramo pogledati, če ni zdaj večji od t_2; če je, moramo prekiniti izvajanje glavne zanke.

Oglejmo si še malo preprostejšo različico doslej opisanega postopka. Stik med pravokotnikom $A = [a_L, a_D]$ iz zgornje vrstice in $B = [b_L, b_D]$ iz spodnje vrstice je prisoten natanko tedaj, ko se spodnja vrstica začne pri nekem $x = t$, za katerega velja $t + b_L < a_D$ (drugače bi bil B v celoti desno od A) in $t + b_D > a_L$ (drugače bi bil B v celoti levo od A). Stik $s = (A, B)$ je torej prisoten od $t = a_L - b_D$ do $t = a_D - b_L$; recimo tema dvema vrednostma „leva“ in „desna“ meja stika s. Za vse možne pare (A, B) lahko izračunamo obe meji, vse te meje zložimo v en sam dolg seznam in jih uredimo naraščajoče (če je več enakih mej, naj pridejo desne pred levimi); pri vsaki meji si tudi zapomnimo, kateremu stiku pripada in ali je leva ali desna. Potem se moramo le sprehoditi po tako urejenem seznamu od začetka do konca; ko se premaknemo mimo neke leve meje, moramo njen stik dodati v kopico K_r, ko se premaknemo mimo neke desne meje, pa moramo njen stik pobrisati iz K_r. Po vsaki taki spremembi pogledamo, kakšen razmik zahteva trenutna množica stikov, med vsemi tako dobljenimi razmiki pa si zapomnimo najmanjšega. Lepo pri tej različici rešitve je, da se nam ni več treba ubadati z vzdrževanjem kopice K_n in z razmišljanjem o tem, kdaj je treba vanjo kaj dodati; časovna zahtevnost je še vedno $O(N^2 \log N)$, enako kot doslej; slabost te različice s seznamom pa je, da imamo v seznamu podatke za vse stike, torej zasede $O(N^2)$ prostora, kopica K_n pri prejšnji rešitvi pa je vsebovala le $O(N)$ stikov in je torej zasedla le $O(N)$ prostora. V obeh različicah pa lahko uporabljeno tehniko gledamo kot primer „pometanja“ ali preleta ravnine (*plane sweep*), ki pride prav pri mnogih geometrijskih problemih (gl. npr. nalogo 1998.2.3, str. 345, rešitev na str. 355).

N: 112

R1991.2.3

Za vsak možen znak (od 0 do 9) pogledamo, na koliko mestih se razlikuje od prebranega (to prešteje podprogram Razlika). V spremenljivki MinRazl si zapomnimo razliko do doslej najpodobnejšega najdenega znaka, NajblizjiZnak pa pove, kateri znak je to bil. Če je novi znak še bližji, si ga zapomnimo kot novega najbližjega; če pa se razlikuje od prebranega za prav toliko kot doslej najbližji, postavimo NajblizjiZnak na presledek, kajti če ne bomo našli kakšnega še bližjega znaka, se ne bomo mogli enolično odločiti za najbližji znak in bomo morali zato vrniti presledek.

```
type ZnakT = array [1..7] of boolean;
var OblikeZnakov: array ['0'..'9'] of ZnakT;
```

```
function DolociZnak(zn: ZnakT): char;    { Vrne najbolj podoben znak podani obliki }
var                                      { ali pa presledek, če najde več enakovrednih kandidatov. }
  c, NajblizjiZnak: char;
  Razl, MinRazl: integer;

  function Razlika(var zn1, zn2: ZnakT): integer;
  { Funkcija vrne število segmentov, v katerih se razlikujeta dva znaka. }
  var i, Razl: integer;
  begin {Razlika}
    Razl := 0;
    for i := 1 to 7 do
      if zn1[i] <> zn2[i] then Razl := Razl + 1;
    Razlika := Razl;
  end; {Razlika}

begin {DolociZnak}
  NajblizjiZnak := ' '; MinRazl := 8;
  for c := '0' to '9' do begin
    Razl := Razlika(zn, OblikeZnakov[c]);
    if Razl < MinRazl then
      begin MinRazl := Razl; NajblizjiZnak := c end
    else if Razl = MinRazl then
      NajblizjiZnak := ' '; { Ne moremo enolično določiti znaka. }
  end; {for}
  DolociZnak := NajblizjiZnak;
end; {DolociZnak}
```

R1991.2.4

Naloga ima seveda precej bistveno različnih rešitev. Ena od možnih rešitev, ki je podobna v praksi znanemu protokolu SDLC ali HDLC, je na kratko naslednja: N: 112

Za začetni znak si izberemo npr. zaporedje bitov 01111110. Pred vsakim paketom oddamo najprej ta znak, bit 1, ki označuje, da gre za veljaven paket, nato oddajamo zaporedne bite paketa in na koncu spet ta znak. Med paketi oddajamo same ničle. Če se v oddanem zaporedju bitov pojavi zaporedje bitov 0111111, pred naslednjim bitom oddamo vrinjeni bit 1, nato pa nadaljujemo z oddajanjem podatkovnih bitov.

Ko na sprejemni strani sprejmemo zaporedje 0111111, počakamo še na naslednji bit. Če je ta bit 0, nam bit za njim pove, ali gre za začetek novega paketa (1) ali pa je to konec prejšnjega paketa (0). Če pa je bil naslednji (osmi) bit 1, ga zavržemo in sprejemamo nadaljevanje paketa.

Za ta postopek potrebujemo na sprejemni strani samo nekaj bitov pomnilnika (če začnemo sprejemati začetni znak, ga moramo hraniti, dokler ne ugotovimo, ali gre za pravi začetni znak ali pa bo imel vrinjeno enico). Šele ko sprejmemo vrinjeno enico, smemo zadržane bite zares razglasiti za sprejete podatke.

Učinkovitost tega postopka (razmerje med količino informacijskih bitov in skupno količino prenesenih bitov) pri paketih dolžine n bitov in povsem slučajno porazdeljenih podatkovnih bitih je približno $n/(17 + n + n/256) = 256n/(257n + 17)$. Pri $n = 1000$ je to 97,95 % in pri $n = 10000$ že 99,44 %.

Če bi si izbrali začetni znak drugačne dolžine, bi lahko učinkovitost pri daljših blokih še poljubno povečali, seveda pa bi zato potrebovali nekaj več bitov pomnilnika na sprejemni strani.

Zahteva o majhni porabi pomnilnika v tej nalogi ni umetna, saj tak prenos delajo običajni sinhroni vmesniki, kjer pomnilnika navadno res nimamo veliko. Zaporedje bitov v našem primeru je navadno kar zaporedje bytov (čeprav ne nujno), zato je izbira osembitnega začetnega znaka zelo naravna. Hkrati nam pakiranje v osembitne byte na sprejemni strani predstavlja natanko tisto zakasnitev pri sprejemu, ki je potrebna, da se lahko odločimo, ali smo sprejeli začetni znak ali pa smo sprejeli znak z vrinjeno enico. Po vrivanju enic ima tak postopek ime "bit stuffing".

REŠITVE NALOG ZA TRETJO SKUPINO

N: 112

R1991.3.1

Podprogram deli prvi parameter z drugim. V resnici oponaša pisno deljenje. Poglejmo ga še enkrat, tokrat z ustreznimi komentarji.

```
function KajStorim(a, b: byte): byte;
{ Funkcija vrne kvocient a/b dveh nepredznačenih 8-bitnih števil. }
var
  Stevec: byte;
  Rezultat: byte; { Tu bomo sestavljali rezultat. }
  Premik: byte;   { Tu zbiramo številke levo od označenega mesta. }
begin
  Rezultat := 0; Premik := 0;
  for Stevec := 1 to 8 do begin
    Premik := 2 * Premik + (a div 128); { Zgornji bit a dodamo v Premik. }
    a := (a mod 128) * 2;          { Premaknemo bite v levo. }
    Rezultat := 2 * Rezultat;      { V rezultat dodamo v spodnji bit ničlo. }
    if Premik >= b then begin      { Če je dobljeno število že večje od delitelja, }
      Premik := Premik - b;        { ga zmanjšamo za delitelj. }
      Rezultat := Rezultat + 1;    { Ničlo v rezultatu spremenimo v ena. }
    end; {if}
  end; {for}
  KajStorim := Rezultat;
end; {KajStorim}
```

Razlika v primerjavi s pisnim deljenjem, kot smo ga sicer navajeni, je le v tem, da delamo tu v dvojiškem zapisu, ne pa v desetiškem. Zato se nam ni

treba vprašati, *kolikokrat* gre delitelj b v Premik, pač pa le, ali sploh gre ali ne. Če gre, dodamo na konec količnika števko 1, sicer pa 0. Dodajanje števke na konec pravzaprav pomeni, da dotedanjo vrednost količnika pomnožimo z 2 in prištejemo novo števko. Na vsakem koraku moramo na koncu Premika pripisati še naslednjo števko (torej: naslednji bit) deljenca. Pri tem sproti skrbimo, da se ta vedno nahaja v najvišjem bitu spremenljivke a; tako lahko do nje pridemo z a **div** 128, nato pa jo v a postavimo na 0 (z a **mod** 128) in nato a pomnožimo z 2, tako da se vsi preostali biti zamaknejo za eno mesto navzgor in s tem na najvišje mesto pride naslednji bit deljenca.

Pri deljenju z 0 se gornji podprogram obnaša malo nenavadno: delitelj b gre zdaj vedno v Premik, zato količniku vedno pripišemo enico in tako ne glede na a dobimo rezultat 255.

Na tak način lahko realiziramo deljenje na procesorjih, ki ne znajo deliti. Za izvajanje operacij a **div** 128 in a **mod** 128, ki se pojavljata v gornjem podprogramu, namreč ne potrebujemo pravega deljenja, ampak le preproste operacije na bitih. Za a **div** 128 je dovolj že premik vrednosti a za sedem bitov v desno (ali pa preverjanje, če je bit 7 prižgan), za a **mod** 128 pa je dovolj, če v a ugasnemo bit 7.

R1991.3.2

N: 113

Pravilna rešitev naloge predstavlja centralni del sintaktičnega analizatorja za prolog. Razlika je le v tem, da prolog dopušča poleg infiksnih operatorjev (med argumentoma) tudi prefiksne (pred argumentom) in postfiksne (za argumentom). En operator je lahko hkrati vseh teh tipov in različne prioritete za vsak tip.

Izraz in njegove podizraze lahko v pomnilniku predstavimo z drevesi. Notranja vozlišča drevesa ustrezajo operatorjem, listi drevesa pa predstavljajo člene izraza.

Podprogram Term prebira izraz, dokler ne naleti na operator, ki veže dovolj šibko (prioriteta > MxOpPri). Prebrani del izraza predstavi z drevesom in v Root vrne kazalec nanj. (Za začetek zgradi drevo z enim samim vozliščem, ki predstavlja trenutno prebrani člen.) Koren tega drevesa mora biti najšibkejši operator v prebranem delu izraza; njegovo prioriteto si zapomnimo v spremenljivki MnOpPri. Če naletimo na nek še šibkejši operator, pomeni, da bo v resnici koren moral postati ta, doslej zgrajeno drevo pa bo le levo poddrevo tega novega korena. Njegovo desno poddrevo pa bo moralo pokrivati nadaljnji del izraza do naslednjega šibkejšega operatorja, tako da lahko to poddrevo zgradimo z rekurzivnim klicem.

WrtTerm le pravilno izpiše izraz in si pri tem spet pomaga z rekurzijo. Podprogramu Parse tako ni treba storiti drugega, kot da kliče Term z dovolj visoko vrednostjo parametra, da bo Term zanesljivo prebral ves izraz; potem je treba izraz le še izpisati.

Spodnji program tudi predpostavi, da ima na voljo nek podprogram Error, ki ga lahko kliče v primeru napak. Error naj bi izpisal kakšno sporočilo o napaki

in prekinil delovanje programa. V praksi bi bilo verjetno dobro poleg številke napake izpisati tudi, kje v vhodnem izrazu je prišlo do nje.

```
const
  MxPri = 1200;
type
  SymDef = (sClen, sOp, sKonec);
  PtExpDef = ↑ExpDef;
  ExpDef = record
             case Sym: SymDef of
               sClen: (Val: char);
               sOp: (Op: char; LExp, RExp: PtExpDef);
           end;
var
  Sym: SymDef;
  Ch: char;
  OpPri: integer;

  procedure Error(n: integer); external;
  procedure GetSym; external;

procedure Term(MxOpPri: integer; var Root: PtExpDef);
var
  MnOpPri: integer;
  OldRoot: PtExpDef;
begin {Term}
  if Sym <> sClen then Error(2); { Predpostavimo, da je trenutni simbol člen. }
  New(Root); Root↑.Sym := Sym; Root↑.Val := Ch;
  MnOpPri := 0; GetSym;
  while (Sym = sOp) and (OpPri <= MxOpPri) do begin
    OldRoot := Root; New(Root); Root↑.Sym := Sym;
    Root↑.Op := Ch; Root↑.LExp := OldRoot; MnOpPri := OpPri;
    GetSym; Term(MnOpPri, Root↑.RExp);
  end; {while}
end; {Term}

procedure WrtTerm(Root: PtExpDef);
begin
  if Root↑.Sym = sClen then Write(Root↑.Val)
  else begin
    Write('('); WrtTerm(Root↑.LExp); Write(Root↑.Op);
    WrtTerm(Root↑.RExp); Write(')');
  end; {if}
end; {WrtTerm}

procedure Parse;
var Root: PtExpDef;
begin
```

```
  GetSym; Term(MxPri, Root);
  if Sym <> sKonec then Error(3);
  WrtTerm(Root);
end; {Parse}
```

R1991.3.3

N: 114

Problem lahko rešimo rekurzivno. Če se vzorec začenja na nekaj znakov, različnih od `?` in `*`, se mora tudi niz začenjati na enake znake, sicer se ne ujemata. Ko v vzorcu naletimo na `*`, lahko nekaj (nič ali več) znakov niza preskočimo, nato pa se mora preostanek niza ujemati s preostankom vzorca, kar lahko preverimo z rekurzivnim klicem. Seveda vnaprej ne moremo vedeti, koliko znakov naj bi ustrezalo tisti zvezdici, tako da moramo preizkusiti vse možnosti (notranja zanka **repeat** v spodnjem programu). Če pa v vzorcu naletimo na `?`, preskočimo en znak niza in nadaljujemo normalno. Podprogram Ujemanje, ki se bo klical rekurzivno, mora reševati probleme oblike „ali se del niza od položaja Ni naprej ujema z delom vzorca od položaja Vi naprej?“.

```
type Niz = packed array [1..10] of char;

function Ujemanje(N, V: Niz; Nd, Vd: integer): boolean;

  function Poskus(Ni, Vi: integer): boolean;
  var Uspesno: boolean;
  begin
    if Vi > Vd then { Prišli smo do konca vzorca — ali tudi do konca niza? }
      Poskus := Ni > Nd
    else if V[Vi] = '*' then begin
      Ni := Ni - 1;
      repeat { Poskusimo z zvezdico pokriti 0, 1, 2, ... znakov niza. }
        Ni := Ni + 1; Uspesno := Poskus(Ni, Vi + 1);
      until Uspesno or (Ni > Nd);
      Poskus := Uspesno;
    end else if (Ni <= Nd) and ((V[Vi] = '?') or (V[Vi] = N[Ni])) then
      { Trenutni znak niza in trenutni znak vzorca se ujemata. }
      Poskus := Poskus(Ni + 1, Vi + 1)  { Pregledujmo naprej. }
    else
      { Trenutni znak vzorca ni niti zvezdica niti vprašaj, trenutni znak niza se
        z njim ne ujema; torej se vzorec V[Vi..Vd] ne ujema z nizom N[Ni..Nd]. }
      Poskus := false;
  end; {Poskus}

begin
  Ujemanje := Poskus(1, 1);
end; {Ujemanje}
```

Slabost takšne rešitve je, da se lahko pri hudobno sestavljenem vzorcu podprogram Poskus kliče zelo velikokrat. Vsaka zvezdica se lahko ujema z 0, 1, 2,

3 itd. znaki. Naš podprogram preizkuša te možnosti po vrsti in šele ko se mu pri eni zatakne, poskusi z naslednjo. Če je v vzorcu veliko zvezdic in je niz takšne oblike, da se ne da dovolj zgodaj ugotoviti, če smo z zvezdico pokrili premalo ali preveč znakov niza, se lahko zgodi, da bo skupno število vseh preizkušenih možnosti naraščalo eksponentno hitro v odvisnosti od števila zvezdic v vzorcu. V takih primerih bi lahko porabil naš podprogram nesprejemljivo veliko časa.[24]

To rešitev lahko izboljšamo, če opazimo, da Poskus niti ne spreminja niza ali vzorca niti ne dela s kakšnimi globalnimi spremenljivkami (z drugimi besedami: je brez „stranskih učinkov“), tako da je njegov rezultat odvisen le od tega, kakšna parametra Ni in Vi je dobil. Če ga torej večkrat kličemo z istima vrednostma teh dveh parametrov, bo dajal vsakič tudi enak rezultat. Torej je pametno, da si po prvem klicu z nekim parom (Ni, Vi) rezultat zapomnimo v neki tabeli in kasneje, če bi ga spet potrebovali, ne izvedemo novega rekurzivnega klica, pač pa vzamemo rezultat iz tabele. Zdaj se bo torej izvedel največ en klic za vsak par (Ni, Vi), teh pa je največ $\mathsf{Nd} \cdot \mathsf{Vd}$).

Namesto takega sprotnega shranjevanja rezultatov v tabeli (čemur pravijo tudi „memoizacija“) bi jih lahko računali tudi sistematično, saj vemo, v kakšnem vrstnem redu jih bomo potrebovali: lahko bi jih računali po padajočih Ni in pri vsakem od njih še po padajočem Vi. Potem tudi vidimo, da si v resnici ni treba zapomniti več kot ene vrstice tabele rezultatov (ko računamo za nek Ni, potrebujemo le rezultate za $\mathsf{Ni} - 1$ in razne možne Vi). Taki tehniki reševanja pravimo tudi „dinamično programiranje“.

Še ena majhna izboljšava bi bila, da pri zvezdici ne bi poskušali na vse

[24] Oglejmo si konkreten primer. Recimo, da bi imeli niz `aa...a` (n `a`-jev) in vzorec `*a*a...*ab` (n zvezdic in `a`-jev ter nato `b`). Niz se seveda ne ujema z vzorcem, vendar mora naš program, da se o tem res prepriča, izvesti med drugim vse tiste rekurzivne klice podprograma Poskus, ki ne izvedejo sami nobenega rekurzivnega klica. To pa so tisti, ki jim od vzorca ostane le še `b`, in tisti, ki jim je niza že zmanjkalo (vidijo le še prazen niz), pri vzorcu pa trenutno vidijo črko `a`. No, do prve možnosti lahko pridemo le tako, da pri nobeni zvezdici ne preskočimo nobenega znaka, ker bo drugače v nizu zmanjkalo `a`-jev, še preden bomo v vzorcu prišli do `b`-ja; tak rekurziven klic je torej samo en. Do druge možnosti pa lahko pridemo na veliko načinov: če vidimo v vzorcu i-ti `a`, pomeni, da smo s prejšnjimi i zvezdicami preskočili vsega skupaj $n - i + 1$ črk `a` v nizu (ostalih $i - 1$ `a`-jev niza pa se je ujelo z dosedanjimi $i - 1$ `a`-ji iz vzorca). Na koliko načinov lahko razbijemo $n - i + 1$ `a`-jev na i kosov (lahko praznih)? To je tako, kot če bi med `a`-je vrivali $i - 1$ „ograjic“ `|`, ki bi ponazarjale meje med kosi; dobimo ravno vse nize dolžine n, ki jih sestavlja $n-i+1$ `a`-jev in $i-1$ ograjic `|`. Takih pa je $\binom{n}{i-1}$ (izmed n mest moramo izbrati $i-1$ mest, na katerih bodo ograjice, drugod pa bodo `a`-ji). Skupno se torej na ta način izvede $\sum_{i=1}^{n} \binom{n}{i-1} = \sum_{i=0}^{n-1} \binom{n}{i}$ robnih rekurzivnih klicev. Če prištejemo k temu še tisti en klic, ki pride v vzorcu do `b`-ja, in upoštevamo, da je $1 = \binom{n}{n}$, dobimo vsoto $\sum_{i=0}^{n} \binom{n}{n}$; to je vsota števil v $(n+1)$-vi vrstici Pascalovega trikotnika in znaša, kot vemo, ravno 2^n. Torej bi se pri takem nizu in takem vzorcu izvedlo v našem programu 2^n takih rekurzivnih klicev podprograma Poskus, ki ne izvedejo sami naprej nobenega novega rekurzivnega klica; že samo zaradi njih (če sploh ne razmišljamo še o rekurzivnih klicih nad njimi) bi bila časovna zahtevnost našega postopka v tem primeru eksponentna v odvisnosti od dolžine niza.

možne načine preskočiti 0 ali več znakov, tako da se iz klica (Ni, Vi) zaredijo klici (i, Vi + 1) za i od Ni do Nd; dovolj bi bilo že, če bi izvedli le klica (Ni, Vi + 1) (če zvezdica ne pokrije nobenega znaka niza) in (Ni + 1, Vi) (zvezdica za začetek pokrije trenutni znak niza, nato pa bo že rekurzivni klic razmislil o tem, ali mora pokriti še kaj več kot to).

R1991.3.4

N: 114

Pri hitrosti 1 MB/s in 256 znakov dolgih blokih potrebujemo približno 0,002 s za sam prenos podatkov enega bloka, poleg tega pa še dvakrat po 1/4 s čakalnega časa, da oddajni računalnik prejme potrditev sprejema bloka od sprejemnika. To pomeni, da je izkoristek komunikacijske linije manj kot pol odstotka, zato opisani preprosti protokol ni primerjen za komunikacijo na zelo velike razdalje.

Izkoristek bi lahko povečali z uporabo daljših blokov, vendar dolžine bloka zaradi verjetnosti napak ne smemo poljubno podaljševati (če se pojavi napaka kjerkoli v bloku, je treba ponoviti prenos celotnega bloka).[25]

Bolje je, da ne čakamo potrditve vsakega bloka posebej, ampak nadaljujemo s pošiljanjem zaporednih oštevilčenih blokov, obenem pa si poslane, a še ne potrjene bloke shranimo za primer, da bi jih bilo treba zaradi komunikacijske napake ponovno poslati.

Ko sprejemni računalnik uspešno prejme nek blok, pošlje oddajnemu računalniku zaporedno število tega uspešno prejetega bloka. Tako lahko oddajni računalnik sprosti pomnilnik za že potrjene bloke.

Če sprejemni računalnik naleti na napačen blok ali ugotovi, da številke blokov niso zaporedne (kakšen blok manjka), pošlje zahtevo za ponovitev vseh blokov od prvega napačnega dalje. Nato čaka na ponovitev zahtevanega bloka in njegovih naslednikov — morebitne bloke, ki še prihajajo za napačnim blokom, zavrže. Po ponovnem sprejemu zahtevanega bloka lahko s komunikacijo normalno nadaljuje.

S skrbnejšim protokolom bi lahko zahtevali ponovitev le napačnih blokov in izkoristili morebitne vmesne pravilne bloke, vendar zaporednost številk blokov olajša delo in omogoča odkrivanje manjkajočih blokov brez dodatnega časovnega nadzora.

V praksi moramo dopustiti možnost, da oddajni računalnik pošilja bloke hitreje, kot jih je sprejemni računalnik sposoben obdelovati. Zato je treba skupaj s potrditvami sprejema pošiljati tudi število paketov, ki jih je sprejemnik še sposoben sprejeti (ima zanje dovolj pomnilnika). Temu številu pravimo

[25] Poleg tega včasih ne bi radi čakali, da se nam bo nabralo dovolj podatkov za cel dolg blok, pa tudi ne bi radi pošiljali dolgega bloka z malo podatki (in veliko ničelnimi biti ali kakšno podobno nekoristno vsebino), ker potem traja dlje, da dobimo od sprejemnika kakršen koli odgovor (ker mora sprejemnik prevzeti in pregledati dolg blok namesto kratkega). Res pa je, da to ni tako pomembno, če imamo opravka s tako dolgimi zakasnitvami kot pri tej nalogi (četrt sekunde v vsako smer).

„kredit“. Oddajnik mora tedaj skrbeti, da nikoli ne pošlje več paketov, kot ima kredita.

Običajno moramo dopustiti tudi možnost, da se kakšna potrditev izgubi ali da je zveza začasno prekinjena. Da prebrodimo to vrsto težav, je treba protokol dopolniti z največjimi čakalnimi časi na posamezen dogodek in z akcijami, ki so potrebne ob prekoračitvi teh časovnih omejitev.

Optimalna dolžina blokov je odvisna od zanesljivosti prenosa in dolžine dodatne informacije, ki je dodana vsakemu bloku. Če je verjetnost napake velika, moramo uporabljati kratke bloke, da vsaj nekateri ostanejo nepoškodovani. Pri majhni verjetnosti napake so primernejši daljši bloki, da pošiljanje dodatne informacije ni tako pogosto. (Če naš protokol v primeru napake zahteva ponovno pošiljanje vseh paketov od vključno tistega, pri katerem je prišlo do napake, pa je škoda tudi pri kratkih blokih velika, saj je treba ponoviti za vsaj pol sekunde prometa.)

16. državno tekmovanje v znanju računalništva (1992)

NALOGE ZA PRVO SKUPINO

1992.1.1

Program za avtomatsko programiranje je iz primerov izluščil relacijo med argumentoma x in y ter vrednostjo funkcije in jo zapisal v obliki naslednjega programa. **Katero relacijo** se je naučil? Odgovor **utemelji**!

R: 140

```
program KajStorim(Input, Output);
var
  x, y, a: 0..MaxInt;
  Kaj: boolean;
begin
  ReadLn(x, y);
  Kaj := true;
  while x <> 0 do begin
    a := x; x := y; y := a;
    y := y - 1;
    Kaj := not Kaj;
  end; {while}
  if Kaj then WriteLn('DA')
  else WriteLn('NE');
end. {KajStorim}
```

```
#include <stdio.h>

void main()
{
  unsigned int x, y, a, kaj;
  scanf("%u%u", &x, &y);
  kaj = 1;
  while (x != 0) {
    a = x; x = y; y = a;
    y--;
    kaj = !kaj;
  }
  if (kaj) printf("DA\n");
  else printf("NE\n");
}
```

1992.1.2

Uslužbenci službe *Pomoč-informacije* sedijo vsak ob svojem telefonu in odgovarjajo na telefonske klice. Njihovi telefoni so povezani na lokalno telefonsko centralo, ki sprejema klice in jih dodeljuje posameznim (prostim) telefonistom.

R: 140

Napiši postopek za upravljanje lokalne telefonske centrale, ki bo sprejemala klice iz telefonskega omrežja in jih dodeljevala telefonistom. Pri tem naj centrala pazi, da bodo vsi telefonisti enakomerno obremenjeni. Naslednji telefonski klic naj dodeli telefonistu, ki trenutno že najdlje počiva.

Lokalna telefonska centrala ima na razpolago naslednje podprograme:

Zvoni — vrne vrednost true, če telefonska centrala zaznava iz omrežja klic, ki še ni prevezan, sicer false.

Zveži(i) — centrala zveže klic iz omrežja z i-tim telefonistom.

Počiva(i) — vrne vrednost true, če je i-ti telefonist prost, sicer false.

R: 141 **1992.1.3** Mačka je igrivo razpoložena in lovi miško. Vsakič ko jo ujame, ji miška lahko uide takoj ali pa kasneje, ko se je mačka z njo v gobčku že nekaj časa potikala naokrog. **Napiši program**, ki bere koordinate, kjer sta se gibali mačka in miška, in izpiše, kolikokrat je miška pobegnila mački. Koordinate so pari celih števil.

R: 141 **1992.1.4** Na slavnostno večerjo kluba *Veseli bitki* je prišel tudi častni gost — računalničar K. Nuth. Častni gost ni nikogar poznal, njega pa so poznali vsi. Tik pred sladico je na veliki dogodek naravnost z letališča prihitel tuj novinar, ki se je hotel srečati s slavnim računalničarjem, ni pa znal niti besedice po naše. Med vožnjo z letališča se je naučil samo eno vprašanje: „Ali poznaš to osebo?“ Njegov načrt je bil preprost — stopil bo k vsakemu gostu in ga za vsakega ostalega gosta po vrsti vprašal, če ga morebiti pozna. Na koncu bo zagotovo vedel, kdo ne pozna nikogar.

Ko je prišel, je ves zgrožen ugotovil, da je povabljencev veliko preveč. Še preden bi uspel zbrati vse odgovore, bi bilo konec večerje. K sreči pa se da najti častnega gosta tudi z veliko manjkrat postavljenim vprašanjem. **V kakšnem vrstnem redu** bi ti spraševal, da bi čim hitreje našel častnega gosta?

NALOGE ZA DRUGO SKUPINO

R: 142 **1992.2.1** **Kaj izpiše naslednji program** za poljubno prebrano (nenegativno) število? Odgovor **utemelji**.

Opombi:

- Funkcija Xor izračuna „ekskluzivni ali“ po bitih dveh števil.
  $\mathsf{Xor}(0,0) = \mathsf{Xor}(1,1) = 1$; $\mathsf{Xor}(0,1) = \mathsf{Xor}(1,0) = 0$;
  $\mathsf{Xor}(10,9) = \mathsf{Xor}(001010_2, 001001_2) = 000011_2 = 3$.
- Funkcija Ord deluje tako:
  Ord(false) = 0, Ord(true) = 1.

```
program TheAnswer(Input, Output);
var
  s, j, m1, m2: 0..MaxInt;
begin
  ReadLn(s);
  m1 := 0; m2 := 0;
  for j := 7 downto 1 do begin
    m1 := 2 * m1 + Ord(Odd(s));
    m2 := 2 * m2
          + Ord(Odd(s) = Odd(j));
    s := s div 2;
  end;
```

```
#include <stdio.h>
void main()
{
  unsigned int s, j, m1, m2;
  scanf("%u", &s);
  m1 = m2 = 0;
  for (j = 7; j > 0; j--) {
    m1 = (m1 << 1) + (s & 1);
    m2 = (m2 << 1)
         + ((s & 1) == (j & 1));
    s >>= 1;
  }
```

```
  WriteLn(Xor(m1, m2));
end. {TheAnswer}
```

```
  printf("%u\n", m1 ↑ m2);
}
```

1992.2.2

V Ministrstvu za raziskovanje rude in zapravljanje časa žigosata papirje dva uradnika, War in Bert. Vsak papir ožigosa najprej War, nato pa še Bert. Ker sta različno hitra, se lahko zgodi, da se naberejo papirji, ki jih je War že ožigosal, Bert pa še ne. Zato imajo v ministrstvu robota, ki skrbi za papirje na poti od Wara k Bertu. Robot je opremljen z roko, ki lahko drži en papir, in s skladiščem. V skladišču je prostora za natanko dva kupa papirjev, imenujmo ju A in B. R: 142

Robota upravljamo z naslednjimi podprogrami:

Nalozi(k: Kup) — naloži papir v roki na vrh kupa k,

Snemi(k: Kup) — vzame papir na vrhu kupa k v roko,

Prazen(k: Kup) — vrne true, če je kup k prazen, sicer false,

pri čemer ima k vrednost A, B ali M. Snemi(M) vzame papir z Warove mize, Nalozi(M) pa ga odloži na Bertovo mizo. Vrednost, ki jo vrne Prazen(M), ni definirana.

Ko War ožigosa papir, ga položi na mizo in pokliče proceduro ShraniPapir, ki shrani papir z njegove mize v skladišče, Bert pa pokliče proceduro VzemiPapir in dobi papir iz skladišča na svojo mizo.

Napiši podprograma ShraniPapir in VzemiPapir tako, da bo Bert žigosal papirje v enakem vrstnem redu, kot jih je žigosal War. Na kaj mora še paziti Bert?

1992.2.3

Napiši podprogramsko funkcijo Val, ki dobi kot prvi parameter niz znakov (desetiški zapis števila) in ga pretvori v število, ki ga vrne kot tretji parameter. Dolžino vhodnega niza dobi kot drugi parameter. Vrednost funkcije naj bo true, če je pretvorba uspela, in false, če niz ne predstavlja legalnega števila. R: 144

```
const StrM = 100;
type string = packed array [1..StrM] of char;
function Val(Str: string; StrL: integer; var n: integer): boolean;
```

Legalno število je niz števk (znaki med '0' in '9'), pred katerim lahko stoji predznak (plus ali minus). Presledke pred in za številom naj podprogram ignorira. Niz mora predstavljati število z intervala MinInt ≤ n ≤ MaxInt.

Za predstavitev celih števil uporablja naš računalnik 16-bitne besede:

```
const
  MinInt = −32768;
  MaxInt = 32767;
type integer = MinInt..MaxInt;
```

Med preverjanjem in pretvorbo podprogram ne sme preseči obsega celih števil niti pri nelegalnih številih!

R: 145

1992.2.4

Robotski vrtalnik uporablja različno velike svedre, ki so shranjeni v luknjah na vrtljivem okroglem stojalu. Vrtalnik zna zamenjati sveder v vrtalnem stroju s svedrom, ki je v luknji ob vrtalnem stroju. Vrtalnik lahko vrti stojalo in tako doseže katerokoli luknjo.

Med delom se svedri pomešajo. **Napiši algoritem**, s katerim vrtalnik po opravljenem delu svedre uredi po velikosti.

Upoštevaj, da lahko vrtalnik opravlja samo naslednje preproste operacije:

1. Zavrti vrtljivo stojalo za eno luknjo v smeri urinega kazalca (podprogram Zavrti). V primerjavi z ostalimi tremi operacijami je premik stojala počasen.

2. Zamenja sveder, ki je vpet v vrtalnem stroju, s tistim, ki je v luknji ob vrtalnem stroju (podprogram Zamenjaj). Če je vrtalni stroj ali luknja prazna, se „zamenjata“ prazen prostor in sveder.

3. Podprogram StrojPrazen vrne true, če v vrtalnem stroju ni ničesar, in false, če je v njem vpet kak sveder.

4. Podprogram Primerjaj primerja velikost svedra v vrtalnem stroju s svedrom, ki je v luknji ob stroju. Vrne:

   - true, če je sveder v vrtalniku manjši od svedra v luknji in
   - false, če je sveder v vrtalniku večji od svedra v luknji.

   Vsi svedri so različno veliki. Prazna luknja/vrtalnik se pri primerjanju obnaša kot sveder velikosti 0 (je „manjša“ od vseh svedrov).

Pred začetkom urejanja je vrtalnik prazen in v vsaki luknji na stojalu je en sveder. Algoritem naj čim hitreje uredi svedre po velikosti od najmanjšega do največjega v smeri urinega kazalca.

NALOGE ZA TRETJO SKUPINO

R: 146

1992.3.1

Program za avtomatsko programiranje je iz primerov izluščil relacijo med argumentoma x in y ter vrednostjo funkcije in jo zapisal v obliki naslednjega programa. **Katero relacijo** se je naučil? Odgovor **utemelji**.

```
type pinteger = 0..MaxInt;

function Enostavna(x, y: pinteger): boolean;
begin
```

```
  if x = 0 then
    Enostavna := true
  else
    Enostavna := not Enostavna (y, x − 1);
end; {Enostavna}
```

```
int Enostavna(unsigned int x, unsigned int y)
{
  if (x == 0) return 1;
  else return !Enostavna(y, x − 1);
}
```

1992.3.2

Pri lepem programiranju obsežnejše programe vedno razdelimo na več modulov. V enem modulu (odvisnem modulu) uporabljamo podatke in podprograme, ki so definirani v drugem modulu (podmodulu). Pred prevajanjem odvisnega modula mora prevajalnik prevesti vse njegove podmodule. R: 147

Napiši algoritem, ki bo izpisal pravilni vrstni red prevajanja modulov. Vhodni podatek algoritmu je ime glavnega modula. Pri tem lahko uporabljaš podprogram:

```
function Podmoduli(OdvisniModul: Ime): SeznamImen;
```

ki nam vrne seznam imen podmodulov, ki jih uporablja odvisni modul, katerega ime podamo kot parameter.

1992.3.3

Napiši podprogram Najdaljse, ki v danem zaporedju celih števil poišče najdaljše strogo naraščajoče podzaporedje. Elemente podzaporedja izbiramo iz začetnega zaporedja od leve proti desni, pri tem pa lahko poljubno število originalnih elementov preskočimo. R: 148

Kadar imamo več enakovrednih rešitev, naj podprogram najde poljubno izmed njih.

Dva primera:

6 2 3 4 1	$\longrightarrow$	2 3 4	
1 1 3 2 6 8 1	$\longrightarrow$	1 3 6 8	ali
		1 2 6 8	

```
const N = ...;    { dolžina najdaljšega vhodnega zaporedja }
type Zaporedje = array [1..N] of integer;
procedure Najdaljse(A: Zaporedje;          { dano zaporedje }
                    DolA: integer;         { njegova dolžina }
                    var B: Zaporedje;      { vrne novo naraščajoče podzaporedje }
                    var DolB: integer);    { in njegovo dolžino }
```

R: 153 **1992.3.4** V grafičnem okolju (na primer X Windows, Macintosh, Amiga in MS Windows) so osnovni objekti okna. To so pravokotna območja na zaslonu; lahko se tudi prekrivajo. Del okna, ki leži pod kakim drugim oknom, ni viden. Uporabnik lahko okna premešča po zaslonu, jih zapira in jim spreminja velikost. Zato mora grafični vmesnik ob vsaki taki spremembi osvežiti vsebino zaslona. Če je kakšno okno (ali njegov del) postalo vidno, ga je treba na novo narisati.

Opiši postopek (algoritem), ki osveži vsebino danega okna:

algoritem OsveziOkno
  vhod: okno W
  učinek: osveži vsebino okna W

Tvoj algoritem lahko uporablja naslednje podprograme in podatke:

- Seznam vseh oken na zaslonu. Okna v seznamu so urejena glede na globino — okno, ki je višje (bolj spredaj) na zaslonu, nastopa prej v seznamu.
- Pravokotniki (in okna) na zaslonu so določeni s koordinatami levega zgornjega in desnega spodnjega oglišča. Privzemi, da je izhodišče koordinatnega sistema v levem zgornjem oglišču zaslona.
- Procedura Nariši(W: Okno; P: Pravokotnik) nariše na zaslon del okna W, ki je določen s pravokotnikom P. Pri tem mora biti pravokotnik P vsebovan v oknu W.

REŠITVE NALOG ZA PRVO SKUPINO

N: 135 **R1992.1.1** Program ugotavlja, ali velja relacija $x \leq y$. Prebrani števili izmenično zmanjšuje za 1, dokler ena od spremenljivk ni enaka 0. Kadarkoli je v spremenljivki Kaj vrednost true, je v x ustrezno zmanjšano drugo prebrano število. Zanka se konča, ko manjše od obeh prebranih števil zmanjšamo na 0. Iz spremenljivke Kaj izvemo, katero od obeh števil je to.

N: 135 **R1992.1.2** Informacijo o trenutni dejavnosti telefonistov hranimo v dveh seznamih. Seznam Zasedeni vsebuje številke vseh zasedenih telefonistov, seznam Prosti pa hrani številke vseh prostih telefonistov. Slednji je urejen tako, da so na začetku tisti telefonisti, ki so prosti že najdlje. To urejenost lahko zagotavljamo s tem, da jemljemo proste telefoniste vedno z začetka seznama; zasedene telefoniste, ki se jim je linija sprostila, pa dodamo vedno na konec.

Postopek je takle:

```
Zasedeni := prazen seznam; Prosti := seznam ⟨1, 2, ..., N⟩;
ponavljaj
  če Zvoni in Prosti ni prazen:
    e := prvi iz seznama Prosti; Zveži(e);
    Briši e iz seznama Prosti;
    Dodaj e na konec seznama Zasedeni;
  za vsak t v seznamu Zasedeni ponovi:
    če Počiva(t):
      Briši t iz seznama Zasedeni;
      Dodaj t na konec seznama Prosti;
```

R1992.1.3

N: 136

V spremenljivki Drzi si zapomnimo, ali je mačka v prejšnjem časovnem koraku držala miško. Če je Drzi = true in imata v naslednjem trenutku mačka in miška različen položaj, vemo, da je miška mački ušla in moramo povečati števec Usla. Drugače pa, če imata miška in mačka enak položaj, postavimo Drzi na true, da jo bomo lahko uporabili v naslednjem časovnem koraku.

```
program MackaInMiska(Input, Output);
var
  xMac, yMac, xMis, yMis: integer;
  Usla: integer;
  Drzi: boolean;
begin
  Drzi := false;
  Usla := 0;
  while not Eof do begin
    ReadLn(xMac, yMac, xMis, yMis);
    if (xMac = xMis) and (yMac = yMis) then Drzi := true
    else if Drzi then begin Drzi := false; Usla := Usla + 1 end;
  end; {while}
  WriteLn('Miška je ušla mački ', Usla, '-krat.');
end. {MackaInMiska}
```

R1992.1.4

N: 136

Problem slavne osebe (celebrity problem) ima res elegantno rešitev. Potrebujemo natanko eno vprašanje manj, kot je vseh povabljenih.

Povabljence oštevilčimo od 1 do N. Odgovor na vprašanje, ali a pozna b, nam da funkcija Pozna(a, b). Uporabimo še dve preprosti resnici. Če a pozna b, potem a gotovo ni častni gost, saj ta ne pozna nikogar. Če pa a ne pozna b, potem b gotovo ni častni gost, kajti le-tega vsi poznajo. Gosta 1 vprašamo, ali pozna gosta N. Če ga pozna, potem 1 gotovo ni častni gost in se premaknemo k naslednjemu. Sicer pa N gotovo ni častni gost in vprašamo gosta 1, ali pozna gosta $N - 1$. Postopek ponavljamo, dokler nam ne ostane samo še en gost — ta je iskani častni gost.

Oglejmo si še program:

```
program CastniGost(Input, Output);
const N = ...;
type GostT = 1..N;
var a, b: GostT;

  function Pozna(a, b: GostT): boolean; external;

begin
  a := 1;
  b := N;
  while a <> b do
    if Pozna(a, b) then a := a + 1 { Če a pozna b, potem a ni častni gost. }
    else b := b - 1;               { Če a ne pozna b, potem b ni častni gost. }
  WriteLn('Častni gost je oseba ', a, '.');
end. {CastniGost}
```

REŠITVE NALOG ZA DRUGO SKUPINO

N: 136

R1992.2.1

Program izpiše številko 42, ne glede na prebrani podatek. Zanka pretoči bite iz s v m1 v obratnem vrstnem redu; v m2 pa dobimo enice na tistih mestih binarnega zapisa, kjer se m1 razlikuje od števila $0101010_2 = 42$. Operacijo Xor si lahko razlagamo tudi kot operacijo, ki komplementira vse tiste bite prvega števila, kjer stoji na istoležnem binarnem mestu drugega števila enica. m2 torej v operaciji Xor obrne vse tiste bite v m1, ki ne ustrezajo predpisanemu vzorcu 0101010, tako da je rezultat operacije Xor(m1, m2) vedno 42.

N: 137

R1992.2.2

Pri nalogi gre pravzaprav za simulacijo vrste z dvema skladoma. Vrsta (*queue*) je podatkovna strukture oblike FIFO (*first in — first out*), ki deluje tako kot vrsta v banki — tisti, ki prej pridejo, so tudi prej na vrsti. Sklad (*stack*) pa je primer strukture LIFO (*last in — first out*) — kot če zložimo knjige v stolp. Snamemo jih lahko samo v vrstnem redu, nasprotnem od tistega, v katerem smo jih naložili.

Najprej zapišimo podprogram, ki ga bomo potrebovali nekoliko pozneje:

```
procedure Prekucni(OdKod, Kam: Kup);
{ Podprogram preloži vse, kar je na kupu OdKod, na kup Kam po en papir naenkrat.
  Pri tem se vrstni red papirjev ravno obrne. }
begin
  while not Prazen(OdKod) do begin
    Snemi(OdKod);
    Nalozi(Kam);
  end; {while}
end; {Prekucni}
```

Prvi način, ki nam pade na pamet, je verjetno ta, da papirje spravljamo tako, da jih nalagamo na kup A, pobiramo pa tako, da prekucnemo kup A na kup B, vzamemo vrhnjega ter prekucnemo kup B nazaj na kup A:

```
procedure ShraniPapir1;
begin
  Snemi(M); Nalozi(A);
end; {ShraniPapir1}

procedure VzemiPapir1;
begin
  Prekucni(A, B);
  Snemi(B); Nalozi(M);
  Prekucni(B, A);
end; {VzemiPapir1}
```

Velika slabost tega pristopa je zelo veliko število premikov, saj dvakrat premaknemo vse spravljene papirje za vsak papir, ki ga hočemo vzeti. Postopek lahko dvakrat izboljšamo tako, da prekucnemo kupček samo takrat, ko je to zares potrebno:

```
procedure ShraniPapir2;
begin
  Prekucni(B, A);
  Snemi(M); Nalozi(A);
end; {ShraniPapir2}

procedure VzemiPapir2;
begin
  Prekucni(A, B);
  Snemi(B); Nalozi(M);
end; {VzemiPapir2}
```

Ta metoda se obnese dobro, če spravljamo oz. jemljemo veliko papirjev naenkrat, za izmenično shranjevanje in jemanje pa je prav tako neučinkovita kot metoda 1. Daleč najbolje se odreže metoda 3, saj vsak papir preloži natanko enkrat:

```
procedure ShraniPapir3;
begin
  Snemi(M); Nalozi(A);
end; {ShraniPapir3}

procedure VzemiPapir3;
begin
  if Prazen(B) then Prekucni(A, B);
  Snemi(B); Nalozi(M);
end; {VzemiPapir3}
```

Bert mora tudi paziti, da ne poskuša vzeti papirja, kadar je skladišče prazno, saj ni nikjer določeno, kaj naredi procedura Snemi, če na kupu ni ničesar. Ali je skladišče prazno, se lahko prepriča s funkcijo:

```
function PraznoSkladisce: boolean;
begin
  PraznoSkladisce := Prazen(A) and Prazen(B);
end; {PraznoSkladisce}
```

N: 137

R1992.2.3

Najprej preskočimo presledke na začetku niza, če jih je kaj; nato pogledamo, če je prisoten predznak, in si ga zapomnimo v spremenljivki Sign. Pri branju števk se vrednost doslej prebranega dela števila nabira v n. Ko preberemo novo števko (z vrednostjo Dig), pomnožimo n z 10 in prištejemo Dig (če beremo negativno število, pa Dig odštejemo oz. ga pred prištevanjem pomnožimo z -1). Tako po vsaki dodatni števki n spet vsebuje vrednost števila, ki ga tvorijo doslej prebrane števke (in predznak). Preden dodamo novo števko, moramo še preveriti, da ne bo nova vrednost slučajno prevelika ali premajhna. Ker ne smemo prekoračiti vrednosti 32767, lahko, če je $n < 3276$, mirno pomnožimo n z 10 in prištejemo novo števko, saj bo rezultat v tem primeru največ 32759; če pa je $n = 3276$, bo $10n + d \leq 32767$, če je $d \leq 7$. Če je $n > 3276$, ga nikakor ne smemo pomnožiti z 10. Podobne pogoje lahko postavimo tudi pri negativnih številih. Ko se števke nehajo, lahko preberemo še nič ali več presledkov, če pa naletimo na kakšen drug znak, moramo javiti napako.

```
const
  StrM = 100;
  MinInt = −32768;
  MaxInt = 32767;
type
  integer = MinInt..MaxInt;
  string = packed array [1..StrM] of char;

function Val(Str: string; StrL: integer; var n: integer): boolean;
const
  MaxN = 3276 {MaxInt div 10}; ModMaxN = 7 {MaxInt mod 10};
  MinN = −3276 {−MinInt div 10}; ModMinN = 8 {−MinInt mod 10};
var
  j, Dig: integer;
  ok, Cont: boolean;
  Sign: (Plus, Minus);
begin
  ok := true; Sign := Plus; n := 0; j := 1;
  { Preskočimo presledke na začetku niza. }
  Cont := true;
```

```
  while Cont and (j <= StrL) do
    if Str[j] <> ' ' then Cont := false else j := j + 1;
  { Preberimo predznak (če je prisoten). }
  if Str[j] in ['+', '-'] then
    begin if Str[j] = '-' then Sign := Minus;    j := j + 1 end;
  { Berimo števke; pazimo, da ne bomo dobili prevelike ali premajhne vrednosti. }
  Cont := true;
  while Cont and (j <= StrL) do
    if not (Str[j] in ['0'..'9']) then Cont := false
    else begin
      Dig := Ord(Str[j]) - Ord('0');
      case Sign of
        Plus: if (n > MaxN) or ((n = MaxN) and (Dig > ModMaxN)) then
                                                          ok := false;
        Minus: begin
          if (n < MinN) or ((n = MinN) and (Dig > ModMinN)) then ok := false;
          Dig := -Dig;
        end;
      end; {case}
      if not ok then Cont := false
      else begin n := 10 * n + Dig; j := j + 1 end;
    end; {else, while}
  if Dig = -1 then ok := false; { Namesto števk je tu nek drug znak. }
  { Preskočimo presledke na koncu niza. }
  Cont := true;
  while Cont and (j <= StrL) do
    if Str[j] <> ' ' then Cont := false else j := j + 1;
  if j <= StrL then ok := false; { Poleg presledkov je za številko še nekaj drugega. }
  Val := ok;
end; {Val}
```

R1992.2.4

N: 138

Rešitev je dobro znani postopek za urejanje z mehurčki (*bubble sort*). Večina ostalih postopkov urejanja predpostavlja, da se lahko premikamo po tabeli, ki jo urejamo, v obe smeri ali pa tudi delamo še kakšne daljše skoke, vse to pa bi bilo pri našem mehanizmu premikanja (ki lahko vrti stojalo le v eno smer in vsakič le za eno luknjo) zelo potratno s časom.

Radi bi, da bi bili svedri na stojalu urejeni naraščajoče v smeri urinega kazalca; torej, ko se stojalo premakne za eno mesto v smeri urinega kazalca, vidimo mi zdaj pred vrtalnikom luknjo, ki je za eno mesto pred tisto, ki smo jo videli prej, zato bi morali videti zdaj (če bi imeli svedre že pravilno urejene) manjši sveder kot na prejšnjem mestu. Torej, če imamo nek sveder v vrtalniku, pa opažamo na stojalu same svedre, ki so večji od njega, ni nič narobe, če gremo mimo njih; ko pa pridemo mimo takega, ki je manjši od tistega v vrtalniku, bi bilo škoda, če bi se premaknili naprej, kajti če bomo tistega v vr-

talniku nekje kasneje odložili na stojalo, bo prišel v smeri urinega kazalca pred tistim manjšim, ki ga zdajle vidimo v vrtalniku. Torej v tem primeru raje zamenjajmo sveder v vrtalniku s tistim na stojalu in potem postopek nadaljujmo s tem manjšim svedrom, ki ga imamo zdaj po novem v vrtalniku. Ko pridemo na stojalu do praznega mesta, torej tja, kjer smo začeli, smo pravzaprav na začetku zaporedja; ker smo vsakič, ko smo naleteli na nek sveder, manjši od tistega, ki je bil trenutno v vrtalniku, izvedli zamenjavo, imamo zdaj v vrtalniku očitno najmanjšega izmed vseh svedrov, tako da ne kaže drugega, kot da ga odložimo na trenutno prazno mesto. Tako bo najmanjši sveder prišel na začetek zaporedja, kamor tudi sodi. Zdaj se lahko z praznim vrtalnikom premaknemo spet na naslednje mesto, poberemo tamkajšnji sveder in vse skupaj ponovimo. Če enkrat naredimo poln krog po stojalu, ne da bi pri tem kdaj nesli manjši sveder mimo večjega (z drugimi besedami: če po vsakem koraku naredimo zamenjavo), pomeni, da so svedri urejeni in se postopek lahko konča.

```
procedure Uredi;
var
  Konec: boolean;
begin
  repeat
    Konec := true;
    Zamenjaj;
    Zavrti;
    repeat
      while Primerjaj do begin
        Zavrti;
        Konec := false;
      end; {while}
      Zamenjaj;
      Zavrti;
    until StrojPrazen;
  until Konec;
end; {Uredi}
```

REŠITVE NALOG ZA TRETJO SKUPINO

N: 138

R1992.3.1

Funkcija Enostavna računa relacijo „manjše ali enako“. To se najhitreje vidi, če jo zapišemo kot rekurzivno matematično relacijo $f(x, y)$:

$$f(x,y) = \begin{cases} \mathsf{true}, & \text{če je } x = 0 \\ \neg f(y, x-1), & \text{sicer.} \end{cases}$$

Če sta x in y strogo večja od nič, smemo razviti dva koraka rekurzije:

$$f(x,y) = \neg f(y, x-1) = \neg\neg f(x-1, y-1) = f(x-1, y-1).$$

Za $x, y \geq k \geq 0$ smemo uporabiti zgornjo lastnost k-krat:

$$f(x,y) = f(x-1,y-1) = f(x-2,y-2) = \ldots = f(x-k,y-k).$$

Sedaj obravnavamo tri primere:

1. če je $x < y$, je $f(x,y) = f(x-x,y-x) = f(0,y) = \mathsf{true}$;
2. če je $x = y$, je $f(x,y) = f(x,x) = f(1,1) = f(0,0) = \mathsf{true}$;
3. če je $x > y$, je $f(x,y) = f(x-y,y-y) = f(x-y,0) = \neg f(0,x-y-1) = \neg\mathsf{true} = \mathsf{false}$.

Vidimo, da funkcija vrne vrednost true v prvem in drugem primeru ter false v tretjem. Torej je to res relacija $\leq$.

R1992.3.2

Algoritem VrstniRed uporablja rekurzivni algoritem VrstniRed2. Zapis S $\oplus$ T pomeni „stakni seznama S in T“. Ko pripravljamo vrstni red prevajanja modulov, morajo biti vsi podmoduli, od katerih je nek modul odvisen, v tem seznamu pred njim. Če imajo ti tudi svoje pod-podmodule, morajo biti ti v seznamu še prej in tako dalje. VrstniRed2 zato pokliče najprej samega sebe za vse podmodule trenutnega modula, nato pa doda trenutni modul na konec zaporedja. Pri tem je koristno še voditi seznam odvisnih modulov („nadmodulov“), ki so pripeljali do trenutnega rekurzivnega klica: če bi se kakšen od njih pojavil kot podmodul, pomeni, da so odvisnosti med moduli ciklične in bi bilo dobro sporočiti to kot napako. Seveda moramo paziti tudi, da ne bi istega modula po večkrat dodali v zaporedje, kar bi se drugače čisto lahko zgodilo, če bi se (ali ta modul ali pa nek drug, ki je od njega odvisen) pojavil kot podmodul pri več drugih modulih.

N: 139

algoritem VrstniRed2
vhod: GlavniModul: ime modula;
Odvisni: seznam modulov;
var Zaporedje: seznam modulov;
izhod: V seznam Zaporedje doda spisek modulov, ki jih je treba prevesti, preden prevedemo glavni modul. Nazadnje doda na konec seznama Zaporedje še modul GlavniModul. Pri tem upošteva, da so moduli iz seznama Odvisni odvisni od modula GlavniModul. Tako lahko odkrije ciklične klice podmodulov.
lokalni spremenljivki:
S: seznam;
M: ime modula;
postopek:
S := Podmoduli(GlavniModul);
ponavljaj za vsak modul M iz seznama S:

```
      če je modul M v seznamu Odvisni,
        so odvisnosti med moduli ciklične. Napaka — prekini izvajanje.
      če modula M še ni v seznamu Zaporedje,
        VrstniRed(M, Odvisni ⊕ ⟨M⟩, Zaporedje);
    Zaporedje := Zaporedje ⊕ ⟨GlavniModul⟩;
```

algoritem VrstniRed
vhod: GlavniModul: ime modula, ki ga želimo prevesti;
izhod: seznam modulov, ki določa vrstni red prevajanja;
lokalna spremenljivka: S: seznam modulov;
postopek:
  S := ⟨⟩;
  VrstniRed2(GlavniModul, ⟨⟩, S);
  vrni S;

N: 139

R1992.3.3

Recimo, da imamo vhodno zaporedje A z n elementi. Omejimo se za začetek na vprašanje, kako dolgo je najdaljše strogo naraščajoče (v nadaljevanju: s. n.) podzaporedje, kasneje pa se bomo ukvarjali še s tem, kako priti do elementov tega podzaporedja. Poskusimo pregledovati elemente zaporedja A od leve proti desni in v neki spremenljivki, recimo k, vzdrževati dolžino najdaljšega strogo naraščajočega podzaporedja v doslej pregledanem delu tabele A. Hitro se lahko prepričamo, da se, če tabeli A dodamo en nov element, vrednost k lahko samo poveča 1 ali pa ostane enaka.[26] Naš postopek bo torej nekako tak:

```
k := 0;
for i := 1 to n do
  if se v A[i] konča kakšno s. n. podzaporedje dolžine k + 1 then k := k + 1;
```

Če naj se v $A[i]$ konča kakšno s. n. podzaporedje dolžine $k+1$, mora za predzadnji člen tega podzaporedja (recimo mu $A[j]$) veljati naslednje troje: $j < i$; pri $A[j]$ se mora končati neko s. n. podzaporedje dolžine k; in $A[j] < A[i]$ (ker imamo opravka s strogo naraščajočim podzaporedjem). Koristno bi bilo torej vedeti, kateri je izmed doslej pregledanih elementov tabele A najmanjši tak (recimo mu v), pri katerem se konča neko s. n. podzaporedje dolžine k; če niti ta element ni manjši od $A[i]$, potem tudi nobeden izmed ostalih elementov,

[26] Naj bo $L[i]$ dolžina najdaljšega s. n. podzaporedja tabele $A[1..i]$. Mislimo si zdaj neko najdaljše s. n. podzaporedje tabele $A[1..i]$; če to podzaporedje ne vsebuje elementa $A[i]$, je hkrati že tudi podzaporedje tabele $A[1..i-1]$; če pa vsebuje element $A[i]$, mu ga lahko odvzamemo in dobimo neko s.n. podzaporedje tabele $A[1..i-1]$ (dolgo $L[i]-1$ elementov); v obeh primerih torej vidimo, da tabela $A[1..i-1]$ vsebuje neko s. n. podzaporedje dolžine vsaj $L[i]-1$, torej je $L[i-1] \geq L[i]-1$ oz. $L[i] \leq L[i-1]+1$. Ker je vsako s. n. podzaporedje tabele $A[1..i-1]$ hkrati tudi s. n. podzaporedje tabele $A[1..i]$, mora veljati seveda tudi $L[i-1] \leq L[i]$. Če oboje združimo, vidimo, da mora biti $L[i]$ enak eni od vrednosti $L[i-1]$ in $L[i-1]+1$.

pri katerih se konča kakšno s. n. podzaporedje dolžine k, ne bo manjši od $A[i]$ in bomo vedeli, da se v $A[i]$ ne končuje nobeno s. n. podzaporedje dolžine $k+1$.

```
k := 0; v := −∞;
for i := 1 to n do
  if v < A[i] then begin k := k + 1; v := A[i] end
  else . . . ;
```

Kaj moramo postaviti namesto treh pik? Če pride izvajanje programa do njih, pomeni, da se pri $A[i]$ ne konča nobeno s. n. podzaporedje dolžine $k+1$; je pa še vedno mogoče, da se pri njem konča vsaj kakšno s. n. podzaporedje dolžine k in v tem primeru je $A[i]$, ker je manjši ali enak v, najmanjši doslej odkriti element, pri katerem se konča kakšno s. n. podzaporedje dolžine n; zato si ga moramo v tem primeru zapomniti kot novo vrednost v.

```
k := 0; v := −∞;
for i := 1 to n do
  if v < A[i] then begin k := k + 1; v := A[i] end
  else if se v A[i] konča kakšno s. n. podzaporedje dolžine k then v := A[i];
```

Zdaj smo pri podobnem problemu kot prej: kako preveriti, če se v $A[i]$ konča kakšno s. n. podzaporedje dolžine k? Če bi že poznali najmanjšega izmed doslej pregledanih elementov, pri katerem se konča kakšno s. n. podzaporedje dolžine $k-1$ — recimo temu elementu v' — bi lahko podobno kot prej preverili, ali je $A[i] > v'$. Če je, se pri $A[i]$ res konča neko s. n. podzaporedje dolžine k, drugače pa ne; toda če se ne, se mogoče pri njem konča neko s. n. podzaporedje dolžine $k-1$ in je zato $A[i]$ kandidat za novo vrednost v'.

```
k := 0; v := −∞; v′ := −∞;
for i := 1 to n do
  if v < A[i] then begin k := k + 1; v := A[i] end
  else if v′ < A[i] then v := A[i]
  else if se v A[i] konča kakšno s. n. podzaporedje dolžine k − 1 then v′ := A[i];
```

Vidimo, da smo padli v zanko; potrebovali bi tudi vrednost v'', ki bi povedala najmanjši element, pri katerem se konča kakšno s. n. zaporedje dolžine $k-2$, itd., itd. Zato vpeljimo kar tabelo $v[1..n]$, v kateri je $v[j]$ vrednost najmanjšega doslej prebranega elementa tabele A, pri katerem se končuje kakšno s. n. podzaporedje dolžine j. Namesto zaporedja stavkov **if** pa bomo morali uporabiti zanko.

```
k := 1; v[1] := A[1];
for i := 2 to n do
  if v[k] < A[i] then begin k := k + 1; v[k] := A[i] end
  else begin
```

```
    j := k;
    while j > 1 do
      if v[j − 1] < A[i] then break else j := j − 1;
    v[j] := A[i];
  end; {if}
```

Majhna slabost tega postopka je v notranji zanki (**while**), ki pregleduje tabelo v, da bi našla največji j, pri katerem je $v[j] < A[i]$. Če imamo smolo, bo morala pri tem pogosto pregledati velik del tabele v; primer je zaporedje

$$A = \langle \tfrac{n}{2}+1, \tfrac{n}{2}+2, \ldots, n, 1, 2, 3, 4, \ldots, \tfrac{n}{2} \rangle.$$

Po pregledu prve polovice tega zaporedja bomo imeli $k = n/2$ in $v = \langle \frac{n}{2} + 1, \ldots, n \rangle$, ker pa so preostali elementi tabele A tako majhni, bo morala iti notranja zanka pri vsakem od njih po celi tabeli v, od desne proti levi vse do prvega elementa. Tako vidimo, da je časovna zahtevnost našega postopka v najslabšem primeru $O(n^2)$.

Tej težavi se lahko izognemo, če tabele v ne preiskujemo po vrsti. Če se pri nekem elementu tabele A konča neko zaporedje dolžine j, se konča pri njem seveda tudi neko zaporedje dolžine $j-1$, zato mora biti vrednost $v[j-1]$ manjša ali enaka $v[j]$. Vrednosti v tabeli v so torej urejene nepadajoče, zato lahko v njej za iskanje največjega elementa $v[j]$, ki je še manjši od $A[i]$, uporabimo kar bisekcijo. Pri tem postopku vzdržujemo spodnjo in zgornjo mejo za iskani indeks j, v vsaki iteraciji zanke pa eno od obeh mej premaknemo tako, da se razdalja med njima razpolovi. Zato pridemo do pravega indeksa že po $O(\log n)$ korakih, časovna zahtevnost celotnega postopka pa je tako $O(n \log n)$ namesto $O(n^2)$. Zanko **while** iz prejšnjega postopka moramo zamenjati z nečim takšnim:

```
p := 0; j := k;
while j − p > 1 do begin
  { Na tem mestu velja v[p] < A[i] ≤ v[j]. Za v[0] si mislimo, da je enak −∞. }
  h := (p + j) div 2;
  { Ker je p ≥ 0 in j − p ≥ 2, se ni treba bati, da bi bil h = 0. }
  if v[h] < A[i] then p := h else j := h;
end; {while}
```

Na koncu je $p = j-1$ in torej velja $v[j-1] < A[i] \le v[j]$, torej smo prišli ravno do tistega j, ki ga iščemo: najdaljše s. n. podzaporedje s koncem pri $A[i]$ je dolgo j elementov.

S tem, kar smo naredili doslej, znamo poiskati dolžino najdaljšega s. n. podzaporedja tabele A — to je kar vrednost k ob koncu postopka. Naloga pa zahteva tudi primer takega podzaporedja, ne le njegove dolžine. Do njega lahko pridemo na več načinov. Ena možnost je, da poleg tabele v uvedemo še eno tabelo, recimo w: če v $v[j]$ piše, kateri doslej prebrani element tabele A je najmanjši tak, da se pri njem končuje kakšno s. n. podzaporedje dolžine j,

naj v $w[j]$ piše indeks tega elementa znotraj tabele A. Potem, ko pri nekem novem elementu $A[i]$ ugotovimo, da se pri njem končuje neko s. n. podzaporedje dolžine j, vemo, da je element $A[w[j-1]]$ lahko njegov predhodnik v nekem takem podzaporedju. Te predhodnike si zapisujmo v neko tabelo, pa bomo iz nje na koncu rekonstruirali najdaljše naraščajoče zaporedje.

```
procedure Najdaljse(A: Zaporedje; DolA: integer;
                    var B: Zaporedje; var DolB: integer);
var v, w, Predhodnik: Zaporedje; i, j, h, p, k: integer;
begin
  k := 1; v[1] := A[1]; w[1] := 1; Predhodnik[1] := 0;
  for i := 2 to n do begin
    if v[k] < A[i] then begin k := k + 1; j := k end
    else begin
      p := 0; j := k;
      while j - p > 1 do begin
        h := (p + j) div 2;
        if v[h] < A[i] then p := h else j := h;
      end; {while}
    end; {if}
    v[j] := A[i]; w[j] := i;
    if j = 1 then Predhodnik[i] := 0
    else Predhodnik[i] := w[j - 1];
  end; {for}
  { Rekonstruirajmo najdaljše strogo naraščajoče zaporedje. }
  DolB := k; i := w[k];
  for j := k downto 1 do
    begin B[j] := i; i := Predhodnik[i] end;
end; {Najdaljse}
```

Eleganten način za rekonstrukcijo najdaljšega s. n. podzaporedja pa je tudi ta, da si za vsak i zapomnimo dolžino najdaljšega s. n. podzaporedja s koncem pri $A[i]$; recimo tej dolžini $L[i]$. Ko je ta tabela pripravljena, jo preglejmo od konca proti začetku; za zadnji element našega s. n. podzaporedja vzemimo kar največji i z lastnostjo $L[i] = k$. Če smo že našli j-ti člen naraščajočega zaporedja (recimo na indeksu i), je primeren $(j-1)$-vi člen kar največji i', za katerega velja, da je $i' < i$ in $L[i'] = L[i] - 1$. (Zagotovo obstaja kak tak i', saj smo lahko med pripravo tabele L do vrednosti $L[i]$ prišli le tako, da smo povečali za 1 vrednost $L[i']$ za nek $i' < i$.) Kako se prepričamo, da v tem primeru res velja $A[i'] < A[i]$? Recimo, da bi vendarle veljalo $A[i'] \geq A[i]$; torej pravi predhodnik i-ja v naraščajočem zaporedju, ki ga iščemo, ne more biti i', pač pa nek i''; zanj mora veljati $L[i''] = L[i] - 1$, ker pa takega elementa med i' in i ni (zaradi načina, kako smo prišli do i'), mora biti $i'' < i'$. Obenem je seveda tudi $A[i''] < A[i]$ (saj smo rekli, da je i'' predhodnik i-ja v s. n. podzaporedju) in zato tudi $A[i''] < A[i']$. Toda to pomeni, da

bi lahko naraščajoče zaporedje do i'' nadaljevali z elementom na indeksu i', tako da mora veljati $L[i'] > L[i'']$, to pa je v nasprotju z našimi dosedanjimi predpostavkami, po katerih sta si $L[i']$ in $L[i'']$ enaka (ker sta oba enaka $L[i]-1$). Tako torej vidimo, da je $A[i']$ prav gotovo manjši od $A[i]$ (saj bi sicer prišli v protislovje) in zato primeren za njegovega predhodnika v naraščajočem zaporedju, ki ga skušamo rekonstruirati.

```
procedure Najdaljse2(A: Zaporedje; DolA: integer;
                     var B: Zaporedje; var DolB: integer);
var v, L: Zaporedje; i, j, h, p, k: integer;
begin
  k := 1; v[1] := A[1]; L[1] := 1; Predhodnik[1] := 0;
  for i := 2 to n do begin
    ... { Enak stavek if kot prej. }
    v[j] := A[i]; L[i] := j;
  end; {for}
  { Rekonstruirajmo najdaljše strogo naraščajoče zaporedje. }
  DolB := k; j := k; i := n;
  while j > 0 do begin
    if L[i] = j then begin B[j] := A[i]; j := j - 1 end
    else i := i - 1;
end; {Najdaljse2}
```

Tako porabimo eno tabelo manj, časovna zahtevnost rekonstrukcije najdaljšega s. n. podzaporedja je v obeh primerih enaka $O(n)$.[27]

[27] O problemu najdaljšega naraščajočega podzaporedja je precej literature ("longest increasing subsequence" ali "longest ascending subsequence" ali "longest upsequence"). Lepa razlaga tu opisanega postopka je v E. W. Dijkstra, *Some beautiful arguments using mathematical induction*, Acta Informatica, 13(1):1–8, January 1980. Če v našem zaporedju A nastopajo dolga strnjena naraščajoča podzaporedja, lahko postopek še pospešimo, če tabelo v popravljamo z neke vrste zlivanjem (R. B. K. Dewar, S. M. Merritt, M. Sharir: *Some modified algorithms for Dijkstra's longest upsequence problem*, Acta Informatica 18(1):1–15, November 1982).

Če so v našem zaporedju A same različne vrednosti, recimo cela števila z intervala $1, \ldots, u$, bodo tudi v tabeli v same različne vrednosti; zato lahko njene elemente namesto v tabeli hranimo v stratificiranem drevesu in vsaka iteracija glavne zanke traja le še $O(\log \log u)$ časa (namesto $O(\log n)$ kot doslej zaradi bisekcije po tabeli v). Več o stratificiranih drevesih najdemo v P. van Emde Boas, R. Kaas, E. Zijlstra: *Design and implementation of an efficient priority queue*, Math. Systems Theory, 10:99–127, 1977; P. van Emde Boas: *Preserving order in a forest in less than logarithmic time and linear space*, Inform. Process. Lett. 6(3):80–82, June 1977; K. Mehlhorn, S. Näher: *Bounded ordered dictionaries in $O(\log \log n)$ time and $O(n)$ space*, Inform. Process. Lett. 35(4):183–189, 7 August 1990.

Če kot zaporedja jemljemo permutacije n različnih števil, je povprečna dolžina najdaljšega naraščajočega podzaporedja približno $2\sqrt{n}$. Če bi nas zanimala poljubna monotona podzaporedja (torej tako naraščajoča kot padajoča), pa se izkaže, da ima najdaljše tako podzaporedje zagotovo vsaj $\sqrt{n}$ elementov (Erdös in Szekeres, 1935; preprost dokaz v P. Pritchard, *Another look at the "Longest Ascending Subsequence" problem*, Acta Informatica, 16(1):87–91, August 1981).

R1992.3.4

N: 140

Algoritem najprej najde vse dele okna W, ki so vidni. Hrani jih v seznamu pravokotnikov SeznP. Na začetku postavi v seznam kar cel pravokotnik W.pravokotnik. Potem po vrsti od seznama „odreže“ pravokotnike V.pravokotnik za vsa okna V, ki so pred oknom W. Na koncu seznam SeznP vsebuje natanko tiste dele okna W, ki so vidni. Za vsakega izmed njih izvede Nariši(W, P).

Napišimo vse to bolj podrobno. Uporabimo naslednje oznake:

V.pravokotnik	pravokotnik, s katerim je določeno okno V
V.predhodnik	okno, ki je neposredno pred oknom V
$P.x_1, P.y_1$	koordinati zgornjega levega oglišča pravokotnika P
$P.x_2, P.y_2$	koordinati zgornjega levega oglišča pravokotnika P
(x_1, x_2, y_1, y_2)	pravokotnik, določen s temi koordinatami; na zaslonu mu pripadajo točke (x, y) za $x_1 \le x < x_2$ in $y_1 \le y < y_2$.
$\langle a, b, c \rangle$	seznam z elementi a, b, c

algoritem OdrežiDvaPravokotnika
vhod: pravokotnika A in B
izhod: seznam pravokotnikov, ki jih dobimo, ko odrežemo B od A in ostanek razdelimo na manjše pravokotnike
lokalna spremenljivka: seznam pravokotnikov Sezn

1. Če je $B.x_2 \le A.x_1$ ali $A.x_2 \le B.x_1$ ali $B.y_2 \le A.y_1$ ali $A.y_2 \le B.y_1$, postavi Sezn := $\langle A \rangle$.
   Pojdi na korak 7.
2. Sezn := $\langle\rangle$;
3. Če je $B.y_1 > A.y_1$, dodaj v Sezn pravokotnik $(A.x_1, A.y_1, A.x_2, B.y_1)$;
4. Če je $B.y_2 < A.y_2$, dodaj v Sezn pravokotnik $(A.x_1, B.y_2, A.x_2, A.y_2)$;
5. Če je $B.x_1 > A.x_1$, dodaj v Sezn pravokotnik $(A.x_1, \max\{A.y_1, B.y_1\}, B.x_1, \min\{A.y_2, B.y_2\})$;
6. Če je $B.x_2 < A.x_2$, dodaj v Sezn pravokotnik $(B.x_2, \max\{A.y_1, B.y_1\}, A.x_2, \min\{A.y_2, B.y_2\})$;
7. Vrni Sezn.

algoritem Odreži
vhod: seznam pravokotnikov Sezn in pravokotnik A
izhod: seznam pravokotnikov, ki jih dobimo, ko odrežemo A od vsakega pravokotnika iz seznama Sezn
lokalna spremenljivka: seznam pravokotnikov Sezn2

1. Sezn2 := $\langle\rangle$;
2. ponavljaj za vse pravokotnike B iz seznama Sezn:
   dodaj seznamu Sezn2 seznam OdrežiDvaPravokotnika(B, A).
3. Vrni Sezn2.

algoritem OsveziOkno
vhod: okno W
izhod: osveži vsebino okna W na zaslonu
lokalni spremenljivki: seznam pravokotnikov SeznP, okno V

1. SeznP := ⟨W.pravokotnik⟩; V := W.predhodnik;
2. ponavljaj, dokler je V ≠ **nil**:
   SeznP := Odreži(SeznP, V.pravokotnik);
   V := V.predhodnik;
3. za vsak pravokotnik P iz seznama SeznP izvedi
   Nariši(W, P);

17. državno tekmovanje v znanju računalništva (1993)

NALOGE ZA PRVO SKUPINO

1993.1.1

Našli smo kos papirja, na katerem je zapisan program. Ali lahko ugotoviš, **kaj program izpisuje?** Pri tem upoštevaj, da so vrednosti konstant c1, c2 in c3 dovolj majhne, da pri računanju ne pride do prekoračitve obsega celih števil. Odgovor utemelji!

R: 161

```
program Mac(Output);

const
  c1 = ...;
  c2 = ...;
  c3 = ...;
var
  a, b, c: integer;

begin
  a := c1;
  b := c1 + c2;
  c := c1 + c2 + c3;
  repeat
    WriteLn(a);
    a := b - a;
    b := c - b;
    b := a + b;
  until false;
end.
```

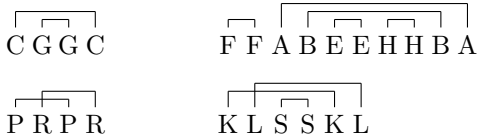
```
#include <stdio.h>

#define c1 ...
#define c2 ...
#define c3 ...

int a, b, c;

void main(void)
{
  a = c1;
  b = c1 + c2;
  c = c1 + c2 + c3;
  do {
    printf("%d\n", a);
    a = b - a;
    b = c - b;
    b = a + b;
  } while (1);
}
```

1993.1.2

Z vhoda preberemo niz znakov. Znaki, ki nastopajo v nizu, se pojavijo *natanko* dvakrat. Vsak par znakov določa povezavo.

R: 162

Primeri povezav:



Napiši program, ki bo preveril, ali pride do križanja povezav.

R: 163

1993.1.3

Slalomska proga z nVrat vratci je podana s koordinatami vratc: vy[v], vxLevi[v], vxDesni[v] (tri tabele realnih števil). Vratca so podana po vrsti od startnih do ciljnih, pri tem vy narašča (vy[v + 1] > vy[v]), velja pa tudi: vxLevi[v] < vxDesni[v].

Smučarjeva pot je podana s tabelama koordinat sx in sy. Med zaporednima točkama smučar vozi v ravni črti. Predpostaviš lahko, da stoji ob startu smučar nad prvimi vratci (sy[1] < vy[1]) in da smuča le navzdol. Tudi vsi sx[i] in sy[i] so realna števila, pri tem pa ni nujno, da so sy podani na mestih, kjer so postavljena vratca.

Napiši del programa, ki izpiše, ali je smučar pravilno prevozil vso progo (da ni zapeljal zunaj nobenih vratic in da je pripeljal do cilja). Predpostaviš lahko, da so podatki že shranjeni v tabelah:

```
const
  nVratMax = 100;
  nPotMax = 100;
var
  vy, vxLevi, vxDesni: array [1..nVratMax] of real;      { koordinate vratic }
  sx, sy: array [1..nPotMax] of real;                  { smučarjeve koordinate }
  nVrat: integer;                                      { dejansko število vratic }
  nPot: integer;          { dejansko število podanih točk v smučarjevi poti }
```

R: 164

1993.1.4

Napiši program, ki bere angleško besedilo in ga pri tem delno uredi. To pomeni, da večkratne presledke med besedami združi v enega in uredi presledke ob ločilih „.“ in „,“ na naslednji način:

- pred ločiloma „.“ in „,“ ni presledka,
- za ločilom „.“ sta natanko dva presledka, če se naslednja beseda začne z veliko začetnico,
- za ločilom „,“ je natanko en presledek,
- med zaporednimi ločili ni presledkov.

Na voljo imaš podprograma:

```
function GetCh(var ch: char): boolean;
procedure PutCh(ch: char);
```

GetCh prebere znak z vhoda in vrne kot funkcijsko vrednost true. Če ni več podatkov na vhodu, vrne false, ch pa ni definiran. PutCh izpiše znak na izhod. Predpostaviš lahko, da GetCh zamenja konce vrstic s presledki, PutCh pa samodejno lomi besedilo na vrstice.

NALOGE ZA DRUGO SKUPINO

1993.2.1

Programer neznanih kvalitet je napisal naslednji program. **Kaj izpiše program** in zakaj izpiše ravno to? **Odgovor utemelji!** R: 165

```
program Kaj(Input, Output);
const
  n = 42;
  n1 = n − 1;
var
  t: array [0..n1] of integer;
  i: integer;

  function Mac(i: integer): integer;
  var x: integer;
  begin
    if i >= n then Mac := 0
    else begin
      x := t[i];
      t[Mac(i + 1)] := x;
      Mac := n − i;
    end;
  end; {Mac}

begin
  for i := 0 to n − 1 do
    Read(t[i]);
  i := Mac(0);
  for i := 0 to n − 1 do
    Write(t[i], ' ');
end.
```

```
#include <stdio.h>

#define n 42

int t[n];

int Mac(int i)
{
  int x;
  if (i >= n) return 0;
  else {
    x = t[i];
    t[Mac(i + 1)] = x;
    return n − i;
  }
}

void main(void)
{
  int i;
  for (i = 0; i < n; i++)
    scanf("%d", &t[i]);
  i = Mac(0);
  for (i = 0; i < n; i++)
    printf("%d ", t[i]);
}
```

1993.2.2

Koordinatni risalnik s peresi različnih barv sprejema risalne ukaze z računalnika in jih sproti izvršuje. Risalnik pozna naslednje ukaze: R: 166

PeroGor	dvigne pero (naslednji Premik ne riše črte)
PeroDol	spusti pero (naslednji Premik riše daljico)
Pero n	zamenja pero — v roko vzame pero številka n
Premik x, y	premakne pero v ravni črti iz trenutne točke v točko (x, y)
Konec	zaključuje vsako risbo.

Zamenjava peresa je zamudna operacija, zato lahko izris večbarvne risbe precej pospešimo, če zberemo skupaj čim večje število ukazov za risanje daljic v isti

barvi. Ker ne moremo spremeniti obstoječe programske opreme tako, da bi pošiljala ukaze risalniku v optimalnem zaporedju, nam preostane le možnost, da med glavni računalnik in risalnik priključimo preprost vmesni računalnik z majhno količino pomnilnika, ki poskrbi za optimizacijo risanja.

Napiši algoritem za vmesni računalnik, ki sprejema ukaze glavnega računalnika in jih pošilja risalniku tako preurejene, da doseže čim hitrejši izris slike. Na vmesnem računalniku obstajajo funkcije za branje ukaza z glavnega računalnika in za pošiljanje ukaza risalniku.

R: 170

1993.2.3

Imamo kvadratno tabelo števil $n \times n$, ki je urejena tako, da števila strogo naraščajo po vrsticah in po stolpcih.

Primer tabele 5×5:

1	3	7	9	11
2	4	8	12	14
3	5	9	13	15
4	6	12	15	17
5	7	15	16	18

Za nek podatek želimo izvedeti, kolikokrat in kje nastopa v tej tabeli. **Napiši podprogram**, ki dobi kot argument iskano število in izpiše koordinate vseh mest v tabeli, kjer se nahaja to število.

Nasvet: obstaja rešitev, pri kateri ni treba pregledati vseh elementov, kar je pomemben prihranek časa pri velikih tabelah.

R: 171

1993.2.4

Znake v Morsejevi abecedi sestavljajo različno dolga zaporedja kratkih in dolgih piskov (od enega do šest piskov).

- Trajanje dolgih piskov (črtic) je trikrat daljše od trajanja kratkih (pik).
- Pavza med dvema piskoma znotraj enega znaka (črke) traja približno toliko, kot traja kratek pisk.
- Znaki (črke) so med seboj ločeni s pavzo, ki traja približno toliko kot dolg pisk.
- Besede so ločene s pavzo, dolgo za več kot dva dolga piska.

Napiši program, ki bo sprejemal besedilo v Morsejevi telegrafiji in **sproti** izpisoval sprejete pike kot „.“ in črtice kot „_“. Znake naj loči med seboj z enim presledkom, vsaka beseda pa naj bo izpisana v novi vrsti. Primer izpisa:

```
··· ·__· ·_· · ·___ · __
__ ___ ·_· ··· · ·___ · ···_ ·
·_ _··· · _·_· · _·· ·
```

Na voljo imaš naslednje podprograme:

Start	postavi štoparico na 0 in jo požene,
Cas: integer	vrne čas v milisekundah od starta štoparice,
Piska: boolean	vrne true, če trenutno sprejemamo pisk, sicer false.

Ker je predpisano le razmerje med dolžino kratkega in dolgega piska, ne pa tudi njuno absolutno trajanje, in ker se lahko hitrost telegrafije tudi nekoliko spreminja, naj se bo program sposoben prilagajati trenutni hitrosti telegrafiranja. Ker v začetku sprejema še ne poznamo hitrosti telegrafiranja, je dopustno, da je prvih nekaj znakov napačno prepoznanih (pike prepoznane kot črte ali obratno) ali celo izgubljenih.

Kako bi zagotovil, da bi bili tudi znaki ob začetku sprejemanja vedno pravilno prepoznani, ne glede na hitrost telegrafiranja? (**Postopek** le **opiši**, dopolnjevanje programa ni potrebno.)

NALOGE ZA TRETJO SKUPINO

1993.3.1

Našli smo kos papirja, na katerem je zapisan program. Ali lahko ugotoviš, **kaj program izpisuje?** Odgovor **utemelji**.

R: 173

```
program Mac(Output);

const
  c1 = ...;
  c2 = ...;
  c3 = ...;
var
  a, b, c: integer;

begin
  a := c1;
  b := c1 xor c2;
  c := c1 xor c2 xor c3;
  repeat
    WriteLn(a);
    a := b xor a;
    b := c xor b;
    b := a xor b;
  until false;
end.
```

```
#include <stdio.h>

#define c1 ...
#define c2 ...
#define c3 ...

int a, b, c;

void main(void)
{
  a = c1;
  b = c1 ^ c2;
  c = c1 ^ c2 ^ c3;
  do {
    printf("%d\n", a);
    a = b ^ a;
    b = c ^ b;
    b = a ^ b;
  } while (1);
}
```

Opomba: operator **xor** ni del standardnega pascala, najdemo pa ga v nekaterih narečjih. Izvede operacijo „ekskluzivni ali“ nad istoležnimi biti v dvojiškem zapisu celih števil — enako kot operator ^ v C-ju.

Velja: 0 **xor** $0 = 0$; 0 **xor** $1 = 1$; 1 **xor** $0 = 1$; 1 **xor** $1 = 0$.
Primer: 9 **xor** $7 = 1001_2$ **xor** $0111_2 = 1110_2 = 14$.

R: 173

1993.3.2

Marvin je poskeniral obris Slovenije, ker bi rad s svojim avtomatskim šivalnim strojem našil obliko slovenske države. Vektorizacijski program je iz bitne slike naredil množico daljic, opisanih z začetno in končno točko, pri tem pa se je malo zapletel — ni ohranil njihovega vrstnega reda.

Zapiši algoritem, ki bo dobljene daljice uredil v pravo zaporedje. Vhod naj bo množica daljic, podanih s krajiščema. Začetno daljico lahko algoritem izbere poljubno, nato pa naj jih izpiše tako, da bosta imeli dve zaporedni daljici vedno eno skupno točko. Eno skupno točko morata imeti tudi prva in zadnja izpisana daljica. Upoštevaj, da je opis Slovenije enostaven in zaključen — v njem ni nobenih presečišč in križanj. Zato se v vsakem krajišču stikata le dve daljici, krajišča pa se popolnoma pokrivajo — k vsaki daljici lahko najdemo natanko dve njeni sosedi.

R: 176

1993.3.3

Za naš mali procesor (CPU) bi radi napisali razhroščevalnik (angl. debugger). Predvsem nas zanima funkcija, ki bo izvajanje v procesorju vrnila za enega ali več ukazov nazaj — pri tem mora v pomnilniku in procesorju vzpostaviti enako stanje, kot je veljalo prej. V razhroščevalniku potrebujemo dva podprograma. Podprogram PredUkazom se samodejno sproži pred izvajanjem vsakega ukaza — njegova naloga je shraniti ustrezne podatke o trenutnem stanju procesorja. Drugi podprogram UkazNazaj lahko uporabnik pokliče kadarkoli in tudi večkrat zaporedoma — njegova funkcija je vrniti izvajanje procesorja po en ukaz nazaj ob vsakem klicu.

Definicija procesorja je taka:

```
type
  BesedaT = 0..65535;
  UkazT = (LDA, LDAI, STA, STAI, MSUB, MSUBI, JPOS, JPOSI);
var
  M: array [BesedaT] of BesedaT;      { pomnilnik }
  P: BesedaT;                         { programski števec }
  A: BesedaT;                         { akumulator }
```

Procesorjevi ukazi so naslednji (n je tipa BesedaT):

```
LDA n     A := n; P := P + 2                          { nalaganje v akumulator }
LDAI n    A := M[n]; P := P + 2
STA n     M[n] := A; P := P + 2                       { shranjevanje akumulatorja }
STAI n    M[M[n]] := A; P := P + 2
MSUB n    A := n − A; P := P + 2                      { računanje z akumulatorjem }
MSUBI n   A := M[n] − A; P := P + 2
JPOS n    if A ≥ 0 then P := n else P := P + 2        { pogojni skok }
JPOSI n   if A ≥ 0 then P := M[n] else P := P + 2
```

Posamezen ukaz je shranjen v dveh pomnilniških celicah:

M[n] := Ord(ukaz); M[n + 1] := operand ukaza;

Napiši podprograma PredUkazom in UkazNazaj.

1993.3.4

Imamo pravokotno mrežo med seboj povezanih procesorjev, ki delujejo vzporedno. Vsak procesor upravlja z eno točko na zaslonu in komunicira s svojimi štirimi sosedi. Na vseh procesorjih tečejo kopije istega programa. R: 178

Neka višja sila enemu izmed procesorjev pove, da je postal središče kroga s polmerom r (točk), ki se mora pobarvati.

Napiši program, ki bo tekel v vseh procesorjih in bo poskrbel za to, da bodo po nekem končnem času procesorji narisali krog.

V programu lahko uporabiš tri podprograme:

Poslji(Sporocilo: SporociloT; vSmer: integer);
: Pošlje sporočilo v poljubni smeri (1: sever; 2: vzhod; 3: jug; 4: zahod).

Sprejmi(**var** Sporocilo: SporociloT; **var** izSmeri: integer);
: Prebere sporočilo in ga vrne. V spremenljivki izSmeri dobimo podatek o tem, iz katere smeri je prišlo sporočilo. Poleg vrednosti 1..4 je možna tudi vrednost 0, ki pomeni, da je sporočilo prišlo „od višje sile“. Zagotovljeno je, da se sporočila ne izgubljajo. Poslji vedno počaka, da sosedov Sprejmi prevzame sporočilo; tudi Sprejmi vedno počaka na sporočilo, če še ni prisotno.

Pobarvaj;
: Pobarva točko, s katero je povezan procesor.

Po želji (in potrebi) lahko dopolniš zapis SporociloT, ki je definiran takole:

```
type
  SporociloT = record
    r: real;
    { prostor za tvoje dodatke }
  end;
```

REŠITVE NALOG ZA PRVO SKUPINO

R1993.1.1

Oglejmo si, kaj se dogaja s spremenljivkami a, b in c med izvajanjem programa. N: 155

	a	b	c	izpis
po inicializaciji	c1	c1 + c2	c1 + c2 + c3	
				c1
a := b − a	c2			
b := c − b		c3		
b := a + b		c2 + c3		
				c2
a := b − a	c3			
b := c − b		c1		
b := a + b		c1 + c3		
				c3
a := b − a	c1			
b := c − b		c2		
b := a + b		c1 + c2		
				c1

Vidimo, da po treh izvajanjih zanke zopet dobimo začetno stanje. Program zato neprestano izpisuje ista tri števila — c1, c2 in c3.

N: 155

R1993.1.2

Recimo, da se pri nekem konkretnem nizu s povezave ne sekajo. Prepričajmo se, da v njem nekje nastopata dva enaka znaka eden tik ob drugem. Naj bo a prvi znak niza; če ne pride takoj za njim še drugi a, pač pa neka druga črka b, se mora druga pojavitev b-ja pojaviti še pred drugo pojavitvijo a-ja, drugače bi se povezavi a-a in b-b križali. Če ti dve pojavitvi b-ja ne nastopata skupaj, mora takoj za prvim b-jem priti neka druga črka, recimo c, in druga pojavitev c-ja mora priti še pred drugo pojavitvijo b-ja. Če tako nadaljujemo, vidimo, da sta si pojavitvi vsake naslednje črke bližje skupaj kot pojavitvi prejšnje, to pa se seveda ne more nadaljevati v nedogled, ampak prej ali slej pridemo do para enakih črk, med katerima ni nobene druge črke. No, če bi zdaj ti dve sosednji enaki črki zbrisali iz niza, je v njem zdaj ena povezava manj kot prej in se torej povezave še vedno ne križajo. Zato bi lahko za novi niz opravili enak razmislek in torej spet našli dve sosednji enaki črki ter ju pobrisali; tako bi lahko nadaljevali, dokler ne bi ostal niz čisto prazen.

Po drugi strani pa, če se v opazovanem nizu s kakšni povezavi križata, je s oblike $s_1as_2bs_3as_4bs_5$. Preden bi lahko postopek iz prejšnjega odstavka pobrisal črki a, bi moral pobrisati vse črke iz podniza s_2bs_3, med drugim torej tudi tisti b; toda preden bi lahko pobrisal b-ja, bi moral pobrisati vse iz podniza s_3as_4, torej tudi tisti a. Tako vidimo, da v resnici ne bo mogel pobrisati niti a-jev niti b-jev, saj bi moral pobrisati a-ja pred b-jema in obratno.

Torej lahko za preverjanje, če se kakšni povezavi križata, uporabimo kar gornji postopek; če nam ta uspe niz v celoti pobrisati, križanja povezav v prvotnem ni bilo, sicer pa je bilo. Za učinkovito izvedbo tega postopka si je

koristno pomagati s skladom. Ko bi preiskovali niz s od leve proti desni, bi v nekem trenutku recimo prvič naleteli na dve sosednji enaki črki; s je torej oblike s_1aas_2 in v s_1 ni nikjer dveh sosednjih enakih črk. Če se zdaj odločimo par aa pobrisati, nam ostane niz s_1s_2 in v njem bi načeloma morali pognati enak postopek. Toda odveč bi bilo, če bi začeli s_1s_2 preiskovati spet na začetku, saj že od prej vemo, da v s_1 ni dveh sosednjih enakih črk; prvi možni položaj, kjer bi se lahko pojavili dve sosednji enaki črki, je torej kot zadnja črka s_1 in prva črka s_2. Zato bo naš program na skladu hranil niz s_1; ko prebere naslednjo črko niza s, jo primerja z zadnjo črko s_1 in če sta enaki, oboje pobriše, sicer pa naslednjo črko odloži na vrh sklada in s tem na konec niza s_1. Da nam ne bi bilo treba vnaprej omejevati dolžine sklada s_1 in s tem dolžine vhodnega niza s, bomo sklad izvedli kar kot seznam elementov, povezanih s kazalci.

```
program Povezave(Input, Output);
type CrkaP = ↑CrkaT;
     CrkaT = record C: char; Nasl: CrkaP end;
var Sklad, P: CrkaP; C: char;
begin
  Sklad := nil;
  while not Eoln do begin
    Read(C);
    if Sklad <> nil then if Sklad↑.C = C then { Zbrišimo C s sklada. }
      begin P := Sklad↑.Nasl; Dispose(Sklad); Sklad := P; continue end;
    { Dodajmo C na sklad. }
    New(P); P↑.C := C; P↑.Nasl := Sklad; Sklad := P;
  end; {while}
  if Sklad = nil then WriteLn('Povezave se ne križajo.')
  else WriteLn('Povezave se križajo.');
end. {Povezave}
```

R1993.1.3

V zanki se bomo z indeksom `v` sprehajali po zaporedju vratic, obenem pa z indeksom `s` po zaporedju smučarjevih položajev. Ko se postavimo v naslednja vratca, indeks `s` po potrebi povečujemo tako dolgo, dokler ne kaže na zadnji položaj smučarja pred trenutnimi vratci. Potem preverimo, če gre daljica od položaja `s` do položaja `s + 1` res skozi vratca; v ta namen moramo izračunati x-koordinato, ki jo ima smučar, ko doseže y-koordinato vratic, in preveriti, če je ta x-koordinata res med levo in desno koordinato teh vratic. Če gre daljica od (x_1, y_1) do (x_2, y_2), pomeni, da se x-koordinata spremeni za $x_2 - x_1$, ko se y-koordinata spremeni za $y_2 - y_1$. Če nas torej zanima, za koliko se spremeni x-koordinata, če se y-koordinata spremeni za $y_3 - y_1$, nam sklepni račun pokaže, da za $(x_2 - x_1) \cdot (y_3 - y_1)/(y_2 - y_1)$. Pri y-koordinati y_3 dosežemo torej x-koordinato

$$x_3 := x_1 + (x_2 - x_1) \cdot (y_3 - y_1)/(y_2 - y_1).$$

N: 156

```
program Slalom(Input, Output);

const
  nVratMax = 100;
  nPotMax = 100;

var
  v: integer;          { zaporedna številka vratic }
  s: integer;          { številka točke v smučarjevi poti }
  vy, vxLevi, vxDesni: array [1..nVratMax] of real;   { koordinate vratic }
  sx, sy: array [1..nPotMax] of real;                 { smučarjeve koordinate }
  nVrat: integer;      { število vratic }
  nPot: integer;       { število podanih točk v smučarjevi poti }
  x: real;             { x smučarja med vratci }
  Odstop: boolean;

begin
  ReadLn(nVrat);
  for v := 1 to nVrat do ReadLn(vy[v], vxLevi[v], vxDesni[v]);
  ReadLn(nPot); for s := 1 to nPot do ReadLn(sy[s], sx[s]);
  Odstop := false; v := 1; s := 1;
  if nPot <= 1 then WriteLn('Odstopil na startu.')
  else begin
    while not Odstop and (v <= nVrat) do begin
      while not Odstop and (sy[s + 1] < vy[v]) do begin
        { do naslednjih smučarjevih koordinat ni vratic, premaknemo smučarja }
        s := s + 1;
        if s >= nPot then begin WriteLn('Odstopil.'); Odstop := true end;
      end; {while}
      { prevozi vratca v }
      while not Odstop and (v <= nVrat) and (sy[s + 1] >= vy[v]) do begin
        { preverimo, če res pelje skozi vratca }
        x := sx[s] + (sx[s + 1] - sx[s]) / (sy[s + 1] - sy[s]) * (vy[v] - sy[s]);
        if (x >= vxLevi[v]) and (x <= vxDesni[v]) then v := v + 1
        else begin WriteLn('Izpustil vratca ', v, '.'); Odstop := true end;
      end; {while}
    end; {while}
    if not Odstop then WriteLn('Smučar je uspešno prevozil progo.');
  end; {if}
end. {Slalom}
```

N: 156

R1993.1.4

Znake, ki jih beremo, lahko sproti izpisujemo, razen če niso presledki. Pri teh si le zapomnimo, da so se pojavili (spremenljivka Presledek v spodnjem programu). V spremenljivki Locilo si zapomnimo, če je bil zadnji doslej prebrani ne-presledek slučajno pika ali vejica. Če res naletimo na piko ali vejico, presledkov pred njo tako ali tako ne smemo izpisati, ločilo pa si zapomnimo v spremenljivki Locilo. Če naletimo na

kak drug znak, ki ni presledek, pa se moramo le še odločiti, koliko presledkov izpisati pred njim: med piko in veliko začetnico sta vedno dva, za vejico je točno eden, v ostalih primerih pa izpišemo enega ali pa nobenega, odvisno od vrednosti spremenljivke Presledek.

```
program Besedilo(Input, Output);

var Znak, Locilo: char;
    Presledek: boolean;

  function GetCh(var ch: char): boolean; external;
  procedure PutCh(ch: char); external;

begin
  Locilo := ' ';
  Presledek := false;
  while GetCh(Znak) do begin
    if Znak = ' ' then Presledek := true
    else if (Znak = '.') or (Znak = ',') then begin
      Locilo := Znak;
      Presledek := false;
      PutCh(Znak);
    end else begin
      if (Locilo = '.') and (Znak in ['A'..'Z']) then
        begin PutCh(' '); PutCh(' ') end
      else if Locilo = ',' then PutCh(' ')
      else if Presledek then PutCh(' ');
      Locilo := ' ';
      Presledek := false;
      PutCh(Znak);
    end; {if}
  end; {while}
end. {Besedilo}
```

REŠITVE NALOG ZA DRUGO SKUPINO

R1993.2.1

Program najprej prebere n števil v tabelo t, nato v funkciji Mac obrne vrstni red števil v tabeli in jih nazadnje v obrnjenem vrstnem redu izpiše. N: 157

V nalogi je zanimiva predvsem funkcija Mac, ki obrne vrstni red števil. Osnovno idejo smo prenesli iz prologovega programa naive-reverse:[28]

[28] Gl. npr. Ivan Bratko, *Prolog Programming for Artificial Intelligence*, 3. izd. (2001), razd. 8.5.4, str. 188. Opisani prologov program je „naiven“ zato, ker moramo iti pri običajnih prologovih seznamih prek celega seznama, preden lahko dodamo na koncu nek nov element. Naš postopek za obračanje seznama dodaja elemente na konec izhodnega seznama enega po enega, zato ima z obračanjem seznama n elementov kar $O(n^2)$ dela; se pa zato včasih

```
reverse([], []).              % prazen seznam obrnemo v prazen seznam
reverse([X | L1], L3) :-      % vhodni seznam razdelimo na
                              %    glavo X in preostanek seznama L1
  reverse(L1, L2),            % seznam L1 obrnemo v seznam L2
  append(L2, [X], L3).        % obrnjenemu seznamu L2 dodamo na konec element X
```

Na podoben način deluje funkcija Mac. Tabelo najprej razdeli na trenutni element (shranimo ga v x) in preostanek, ki ga obrnemo s klicem Mac(i + 1). Na koncu shranjeni element postavimo na ustrezno mesto (tega spotoma izračuna rekurzivni klic Mac(i + 1)).

N: 157

R1993.2.2

Naloga bi lahko kaj bolj natančno povedala, kaj naredi risalnik, ko prejme ukaz o zamenjavi peresa: Ali pred premikom do vrtiljaka s peresi (*carousel*) dvigne pero, če je bilo prej spuščeno? Ali se, ko vzame novo pero, premakne na položaj, kjer je bil, ko je prejel ukaz za zamenjavo peresa? Ali pa le obstane na nekem vnaprej definiranem položaju? Ali novo pero spusti, če je bilo spuščeno tudi staro pero pred zamenjavo? Te reči moramo poznati, če hočemo pošteno simulirati vsa možna zaporedja ukazov. Lahko bi na primer predpostavili, da ukazi, ki jih dobivamo z računalnika, pred zamenjavo peresa vsebujejo tudi ukaz za dvig peresa, po zamenjavi pa se eksplicitno premaknejo na nek nov položaj in šele tam spustijo pero. Če pa kaj od tega manjka, bi morali načeloma za te reči poskrbeti mi. Recimo za primer, da ima program nekaj ukazov za eno pero, nato preklopi na drugo, spet nekaj riše, preklopi nazaj na prvo in nariše še nekaj črt. Če bi zdaj mi prvo in tretjo skupino ukazov hoteli združiti (ker se obe nanašata na isto pero), je npr. mogoče, da se prva ne konča z ukazom za dvig peresa, ker je uporabnik vedel, da bo naslednji ukaz (za preklop na drugo pero) itak implicitno tudi dvignil pero; tretja skupina ukazov pa se mogoče začne s premikom, ki bi se opravil z dvignjenim peresom, če velja, da je pero po zamenjavi vedno dvignjeno; v tem primeru bi morali mi, ko združujemo prvo in tretjo skupino ukazov, vmes vriniti ukaz za dvig peresa.

Predpostavili bomo, da risalnik, ko dobi ukaz za zamenjavo peresa, najprej dvigne pero (če je bilo prej spuščeno), se odpelje do vrtiljaka s peresi, odloži dosedanje pero, vzame novo in nato obstane z dvignjenim novim peresom na nekem fiksnem in vnaprej znanem položaju. To je smiselno, saj prav gotovo nihče, ko pošlje risalniku ukaz za zamenjavo peresa, od njega ne pričakuje, naj nariše spotoma še črto v stari barvi od trenutnega položaja do vrtiljaka in nato v novi barvi od vrtiljaka nazaj do istega položaja ali kaj podobno neumnega. Predpostavili bomo tudi, da v zaporedju ukazov, ki prihajajo z računalnika, takoj za ukazom za zamenjavo peresa vedno pride ukaz za premik. Tudi to je precej smiselno, saj je po zamenjavi peresa trenutni položaj nekje pri vrtiljaku in verjetnost, da hoče uporabnik risati črto ravno od tam, je neznatno majhna.

---

uporablja kot benchmark pri merjenju hitrosti interpreterjev prologa.

Vprašanje je še, kaj risalnik naredi, če dobi ukaz, naj zamenja pero s tistim, ki ga že zdaj drži. Glede tega bomo predpostavili, da bi ravnal enako, kot če bi moral res vzeti neko drugo pero; skratka, da bo dvignil pero, se odpeljal do vrtiljaka, ga odložil in takoj spet pobral.

Naš program bo ukaze, ki prihajajo z računalnika, odlagal v nek vmesni pomnilnik (v vrsto), razen če se ne nanašajo na pero, ki ga risalnik trenutno drži v roki — take pa lahko pošljemo naravnost risalniku. Zato moramo ločeno voditi podatek o peresu, ki je res v risalniku (PeroRisalnika), in o tistem, za katerega računalnik misli, da je v risalniku (PeroRacunalnika — to je pero, ki bi tudi res bilo v risalniku, če se ne bi vmešal naš program). Da poraba pomnilnika ne bo naraščala v nedogled, se dogovorimo za neko največjo dovoljeno velikost vrste (MaxVrsta ukazov); ko se vrsta napolni, bomo zamenjali pero in takoj izvedli tiste ukaze iz vrste, ki se nanašajo na novo pero. Da bi bilo treba peresa menjati čim redkeje, si vedno izberimo tako pero, na katerega se nanaša največ ukazov iz vrste, tako da se bo vrsta čim bolj spraznila. Zato bomo v tabeli Kolikokrat za vsako pero hranili podatek o tem, kolikokrat se pojavlja v vrsti.

Podprogram ObdelajUkaz izvede dani ukaz, če se nanaša na trenutno pero, sicer pa ga doda v vrsto. Podprogram PocistiPero pa zamenja pero v risalniku in še enkrat prebere vse ukaze iz vrste ter jih pošilja skozi ObdelajUkaz. Ob tem se bodo torej vsi ukazi, ki se nanašajo na novo pero risalnika, tudi res izvedli, ostali pa bodo prišli nazaj v vrsto. Posebej opozorimo na dejstvo, da je pri spodnjem programu ves čas, razen v okviru izvajanja podprograma PocistiPero, na začetku vrste nek ukaz za menjavo peresa.[29] Zato ni pomembno, kakšna je vrednost spremenljivke PeroRacunalnika tik pred klicem podprograma PocistiPero; ta bo v vsakem primeru pravilno izvedel natanko tiste ukaze, ki se nanašajo na novo barvo peresa, na koncu pa bo imela tudi PeroRacunalnika spet tako vrednost kot na začetku (torej vrednost, ki je ostala po zadnjem ukazu

[29]Res: (1) Na začetku, do prvega ukaza za menjavo peresa, predpostavimo, da hoče uporabnik risati s peresom, ki je že od prej v risalniku, pa kakršne koli barve že pač je; zato postavimo na začetku tako PeroRacunalnika kot PeroRisalnika na 0 in do prvega ukaza za menjavo peresa sproti izvajamo vse ukaze, tako da v vrsto ne pride nič. Ko torej ukaze začnemo dodajati v vrsto, se prav gotovo v njej prvi znajde nek ukaz za menjavo peresa. (2) Kasneje pa, ko ukaze jemljemo iz vrste, se to vedno počne le znotraj podprograma PocistiPero, ta pa iz vrste pobere vse ukaze in jih posreduje podprogramu ObdelajUkaz. Tu lahko ločimo dve možnosti: ali se ukaz za menjavo peresa z začetka vrste ujema s peresom NovoPero ali pa ne. (2a) Če se ujema, bi ObdelajUkaz takoj izvedel tega in vse nadaljnje ukaze do prve naslednje menjave peresa, tako da bi se vsebina vrste nato spet začela z ukazom za menjavo peresa. (2b) Če pa se ne ujema, bi ta ukaz takoj spet dodali v vrsto in ko bi zanka v PocistiPero prišla do konca, bi bil ta ukaz, torej nek ukaz za menjavo peresa, spet na začetku vrste. (3) Do težave bi lahko prišlo le še pri dodajanju v prazno vrsto, toda če je vrsta prazna, pomeni, da smo ali šele na začetku ali pa se je ravnokar izvedla PocistiPero in se je pred tem cela vrsta nanašala na pero v eni sami barvi. V vsakem primeru je bilo torej po tistem pero računalnika enako peresu tiskalnika, tako da, če se je zdaj pojavila potreba po tem, da bi v vrsto nekaj dodali, pomeni, da se je moralo pero računalnika spremeniti, torej smo dobili tudi ukaz za menjavo peresa in ravno ta bo šel prvi v vrsto.

za menjavo peresa), kot je tudi prav.

Če hoče ObdelajUkaz dodati ukaz v vrsto, pa opazi, da je ta že polna, pokliče PocistiPero, kar bo iz vrste zanesljivo odstranilo vsaj en ukaz. Ko PocistiPero opravi svoje delo, lahko ObdelajUkaz obdela trenutni ukaz na enak način, kot da bi ga ravnokar šele prejel: zaradi klica PocistiPero je zdaj mogoče pero računalnika isto kot pero risalnika in lahko pošljemo novi ukaz naravnost risalniku, če pa ga bo treba dodati v vrsto, je v njej po vrnitvi iz PocistiPero zagotovo dovolj prostora še vsaj za en ukaz. Ko dodamo ukaz v vrsto, seveda ne smemo pozabiti povečati števca, koliko ukazov v vrsti se nanaša na to pero (Kolikokrat[PeroRacunalnika]).

Če dobi ObdelajUkaz ukaz za menjavo peresa z istim, ki je že zdaj v risalniku, namesto tega pošlje risalniku le ukaz za dvig peresa. V skladu s prej navedenimi predpostavkami si namreč mislimo, da bi risalnik ukaz za zamenjavo res izvedel in s tem po nepotrebnem tratil čas, zato mu tega ukaza raje ne pošiljamo; po drugi strani pa bi po zamenjavi veljalo, da je pero dvignjeno, zato ga moramo tudi mi zdaj dvigniti: pero je bilo namreč doslej mogoče spuščeno in ker ukazu za zamenjavo verjetno sledi ukaz za premik, ne bi bilo dobro, če bi risalnik vlekel črto do novega položaja (saj se uporabnik zanaša na to, da se je pred tem izvedel ukaz za menjavo peresa, ki je pustil pero dvignjeno). Zanašamo se na to, da je ukaz za dvig peresa hiter, zato se ne bomo obremenjevali s tem, če ga kdaj pa kdaj slučajno izvedemo brez potrebe (ker je bilo pero že itak dvignjeno) — drugače bi morali poleg podatka o trenutnem peresu risalnika vzdrževati pač tudi podatek o trenutnem položaju peresa (dvignjeno ali spuščeno).

Ko dobimo od računalnika ukaz za konec risbe, moramo, preden ga pošljemo risalniku, še izprazniti vrsto. V ta namen si tudi lahko pomagamo s podprogramom PocistiPero, tako da tudi zdaj ne bomo menjali peres večkrat, kot je nujno potrebno (po enkrat za vsako pero, ki ga sploh potrebujemo).

Program bi lahko še malo izboljšali, da bi pospeševal izvajanje kakšnih patoloških zaporedij ukazov (npr. takih, ki pridno menjajo peresa, vmes pa le mahajo z njimi po zraku, ne da bi jih res spustili na papir in z njimi kaj narisali — očitno je, da bi lahko tako menjavo peresa in vse pripadajoče gibe čisto opustili).

Še ena izboljšava bi bila, da bi imeli za vsako pero ločeno vrsto oz. poseben seznam ukazov za to pero. Pri tem bi lahko še vedno hranili podatke o dolžini teh seznamov in pazili, da skupna dolžina ne preseže neke dogovorjene meje. Lepo pri tej rešitvi bi bilo, da nam ob klicu PocistiPero ne bi bilo treba pregledati vseh ukazov v vrsti, pač pa le tiste, ki se nanašajo na izbrano pero.

```
program KrmiljenjeRisalnika;
const StPeres = 10;
type OperacijaT = (opPeroGor, opPeroDol, opPero, opPremik, opKonec);
     UkazT = record
```

```
        case Kaj: OperacijaT of
          opPero: (n: integer);
          opPremik: (x, y: integer);
      end;

  procedure BeriUkaz(var Ukaz: UkazT); external;
  procedure PosljiUkaz(Ukaz: UkazT); external;

const MaxVrsta = 100;
var Vrsta: array [0..MaxVrsta - 1] of UkazT;
    Glava, DolVrste: integer;

  procedure VrstaVzemi(var Ukaz: UkazT);
  begin
    Ukaz := Vrsta[Glava]; Glava := Glava + 1; DolVrste := DolVrste - 1;
    if Glava = MaxVrsta then Glava := 0;
  end; {VrstaVzemi}

  procedure VrstaDodaj(Ukaz: UkazT);
  begin
    Vrsta[(Glava + DolVrste) mod MaxVrsta] := Ukaz;
    DolVrste := DolVrste + 1;
  end; {VrstaDodaj}

var Kolikokrat: array [1..StPeres] of integer;
    PeroRisalnika, PeroRacunalnika: integer;

  procedure PocistiPero(NovoPero: integer); forward;

  procedure ObdelajUkaz(Ukaz: UkazT);
  var Pero, NajPero: integer;
  begin
    if Ukaz.Kaj = opPero then { Zapomnimo si novo pero. }
      PeroRacunalnika := Ukaz.n;
    if PeroRacunalnika = PeroRisalnika then begin
      if Ukaz.Kaj = opPero then Ukaz.Kaj := opPeroGor;
      PosljiUkaz(Ukaz);
    end else begin
      if (DolVrste = MaxVrsta) then begin
        { Za katero pero imamo največ ukazov? }
        NajPero := 1;
        for Pero := 2 to StPeres do
          if Kolikokrat[Pero] > Kolikokrat[NajPero] then NajPero := Pero;
        { Preklopimo na to pero in takoj izvedimo ukaze zanj. }
        PocistiPero(NajPero);
        ObdelajUkaz(Ukaz);
      end else begin
        VrstaDodaj(Ukaz);
```

```
            Kolikokrat[PeroRacunalnika] := Kolikokrat[PeroRacunalnika] + 1;
          end; {if}
        end; {if}
    end; {ObdelajUkaz}

    procedure PocistiPero(NovoPero: integer);
    var Pero, StUkazov: integer; Ukaz: UkazT;
    begin
      { Ukaze bomo na novo prebirali in posiljali podprogramu ObdelajUkaz.
        Zato postavimo števce na 0, saj jih bo ObdelajUkaz ponovno povečeval. }
      for Pero := 1 to StPeres do Kolikokrat[Pero] := 0;
      { Preklopimo na to pero. }
      Ukaz.Kaj := opPero; Ukaz.n := NovoPero; PosljiUkaz(Ukaz);
      PeroRisalnika := NovoPero;
      { Pošljimo vse ukaze iz vrste še enkrat skozi podprogram
        ObdelajUkaz. Tisti, ki se nanašajo na novo pero, se bodo
        pri tem zares izvedli, ostali pa bodo prišli nazaj v vrsto. }
      StUkazov := DolVrste; Pero := 0;
      while StUkazov > 0 do begin
        VrstaVzemi(Ukaz);
        ObdelajUkaz(Ukaz);
        StUkazov := StUkazov - 1;
      end; {while}
    end; {PocistiPero}

var Ukaz: UkazT; Pero: integer;
begin {KrmiljenjeRisalnika}
    for Pero := 1 to StPeres do Kolikokrat[Pero] := 0;
    PeroRacunalnika := 0; PeroRisalnika := 0;
    while true do begin { Neskončna zanka. }
      BeriUkaz(Ukaz);
      while Ukaz.Kaj <> opKonec do begin
        ObdelajUkaz(Ukaz);
        BeriUkaz(Ukaz);
      end; {while}
      { Risbe je konec; pošljimo risalniku še ukaze, ki čakajo v vrsti. }
      for Pero := 1 to StPeres do
        if Kolikokrat[Pero] > 0 then PocistiPero(Pero);
      PosljiUkaz(Ukaz); { Pošljimo mu še ukaz za konec risbe. }
    end; {while}
end. {KrmiljenjeRisalnika}
```

N: 158

R1993.2.3

Najhitrejša rešitev naloge je naslednja: začnemo v levem spodnjem oglišču. Če je trenutni element manjši od iskanega, se premaknemo v desno. Če je večji, se premaknemo navzgor. Če je element enak iskanemu, je vseeno, ali se premaknemo navzgor ali v desno. Postopek skoraj brez sprememb deluje tudi na pravokotni tabeli.

```
var t: array [1..n, 1..n] of integer;

procedure Poisci(Element: integer);
var x, y, Stevec: integer;
begin
  x := 1;
  y := n;
  Stevec := 0;
  while (x <= n) and (y >= 1) do begin
    if Element > t[x, y] then x := x + 1
    else if Element < t[x, y] then y := y - 1
    else begin
      WriteLn('Našel sem iskano vrednost ', Element,
              ' v celici t[', x, ',', y, '].');
      Stevec := Stevec + 1;
      x := x + 1;
    end; {if}
  end; {while}
  WriteLn('Število najdenih primerkov vrednosti ', Element,
          ': ', Stevec, '.');
end; {Poisci}
```

O pravilnosti tega postopka se lahko prepričamo z naslednjo invarianto: na začetku vsakega izvajanja glavne zanke **while** v gornjem programu vsebuje spremenljivka Stevec število takih pojavitev vrednosti Element, ki imajo ali $X < x$ ali $Y > y$. Na začetku to očitno velja, saj pri $x = 1$ in $y = n$ ustreznih položajev (X, Y) sploh ni, Stevec pa je enak 0. Če potem opazimo, da je element (x, y) manjši od iskane vrednosti, pomeni, da so tudi tisti nad njim v istem stolpcu manjši od te vrednosti (saj vemo, da elementi dol po stolpcu naraščajo); torej lahko povečamo x za 1, pa bo invarianta še vedno veljala. Podobno pa, če opazimo, da je element (x, y) večji od iskane vrednosti, so tisti desno od njega v isti vrstici tudi večji od nje, tako da lahko odpišemo celo vrstico: ko zmanjšamo y za 1, invarianta še vedno velja. Ob koncu izvajanja podprograma je $x \geq n$ ali pa $y \leq 1$ in pogoju „$X < x$ ali $Y > y$“ ustrezajo vsi položaji (X, Y) v tabeli, tako da je v Stevec očitno res pravo število pojavitev vrednosti Element v celi tabeli.

R1993.2.4

N: 158

Spodnji program hrani v spremenljivki dt1 dolžino kratkega piska, kot jo je ocenil iz dosedanjih meritev. Naslednji pisk naj bi bil torej dolg ali dt1 ali pa trikrat toliko. Ker naloga pravi, da se lahko dolžina piska počasi tudi spreminja, bomo dolžino naslednjega piska primerjali z dvakratnikom enote dt1 — če je pisk krajši, si ga bomo razlagali kot kratek pisk, drugače pa kot dolg pisk. V vsakem primeru lahko zdaj iz tega piska ocenimo novo enotsko dolžino kratkega piska (ndt1) in jo potem skombiniramo z dosedanjo (dosedanja ima težo Utez, nova pa težo 1); to nam zagotovi,

da referenčne dolžine dt1 ne bomo prehitro spremenili le zaradi enega ali dveh neobičajno dolgih ali kratkih piskov, če pa so takšne spremembe trajnejše, bo dt1 sčasoma lahko prišla poljubno blizu nove dolžine piska. Ko smo uspešno prepoznali pisk, moramo izmeriti še trajanje pavze za njim, da ugotovimo, če gre morebiti za konec črke ali celo konec besede.

```
program Morse(Input, Output);

const Pika = '.';
      Crta = '_';

type CasT = integer;

var
  dt: CasT;       { izmerjeni časovni interval }
  dt1: CasT;      { referenčna osnovna enota — trajanje pike }
  ndt1: CasT;     { osnovna enota, kot smo jo izmerili pri zadnjem znaku }
  Utez: integer;  { stopnja nespremenljivosti referenčne enote }
  p: boolean;
  Znak: char;

  procedure Start; external;
  function Cas: CasT; external;
  function Piska: boolean; external;

begin
  repeat until not Piska;
  dt1 := 0; Utez := 0;
  while true do begin
    repeat until Piska; { počakamo začetek piska }
    Start;
    repeat until not Piska; { počakamo konec piska }
    dt := Cas;
    Start;
    if dt < 2 * dt1 { primerjamo trajanje piska z referenco }
      then begin Znak := Pika; ndt1 := dt end{ pika traja eno enoto }
      else begin Znak := Crta; ndt1 := dt div 3 end; { črtica traja tri enote }
    Write(Znak);
    dt1 := (Utez * dt1 + ndt1) div (Utez + 1); { prilagodimo časovno enoto }
    if Utez < 5 then Utez := Utez + 1; { referenčna enota postaja zanesljivejša }
    { počakamo začetek piska ali dovolj dolgo, da velja konec črke }
    repeat p := Piska until p or (Cas > 2 * dt1);
    if not p then Write(' ');
    { počakamo začetek piska ali dovolj dolgo, da velja konec besede }
    repeat p := Piska until p or (Cas > 5 * dt1);
    if not p then WriteLn;
  end; {while}
end. {Morse}
```

Da bi pravilno prepoznali tudi znake na začetku telegrama (ko je vrednost referenčne časovne enote še neznana), moramo trajanja piskov shranjevati in odlašati z njihovo kategorizacijo v pike ali črtice ter z izpisom tako dolgo, da se med shranjenimi časi pojavita dve kategoriji trajanj — kratko in dolgo. Na podlagi teh kategorij lahko določimo začetno vrednost časovne enote (reference) in glede na njo kategoriziramo in izpišemo shranjene znake. Nadaljnji postopek je enak tukaj sprogramiranemu, shranjevanje ni več potrebno.

Mimogrede, pike in črtice v primeru iz besedila naloge pomenijo: „SPREJEM MORSEJEVE ABECEDE“.

REŠITVE NALOG ZA TRETJO SKUPINO

R1993.3.1

Oglejmo si, kaj se dogaja s spremenljivkami a, b in c med izvajanjem programa: N: 159

	a	b	c	izpis
po inicializaciji	c1	c1 **xor** c2	c1 **xor** c2 **xor** c3	
				c1
a := b **xor** a	c2			
b := c **xor** b		c3		
b := a **xor** b		c2 **xor** c3		
				c2
a := b **xor** a	c3			
b := c **xor** b		c1		
b := a **xor** b		c1 **xor** c3		
				c3
a := b **xor** a	c1			
b := c **xor** b		c2		
b := a **xor** b		c1 **xor** c2		
				c1

Pri tem smo upoštevali eno najlepših lastnosti operacije **xor**: a **xor** a = 0.

Vidimo, da po treh izvajanjih zanke zopet dobimo začetno stanje. Program zato neprestano izpisuje ista tri števila — c1, c2 in c3.

R1993.3.2

Različnih pristopov k reševanju te naloge je zelo veliko. Razložili bomo enega najhitrejših, ki je žal rahlo razsipen s prostorom. Osnovnih principov, ki smo jih uporabili tu, ni težko prenesti v manj potraten, a počasnejši algoritem. N: 160

Resnično pomembna v spodnjem programu sta le podprograma PreberiUredi in Izpisi. Prvi prebere podatke o daljicah in vsako posebej vstavi v seznam. Drugi se sprehodi po tem seznamu in izpiše daljice v pravem vrstnem redu.

Namesto da bi v seznam vstavljali daljice, bomo raje vstavljali njihova krajišča. Ko preberemo novo daljico, poiščemo obe krajišči v seznamu. Če krajišča ne najdemo, ga v seznam vstavimo. Nato v seznam zapišemo še povezavo med njima.

Iskanje in vstavljanje v spodnjem programu zelo učinkovito počne prodprogram Vstavi. Za razumevanje je malce težji, ker uporablja razpršene tabele, prav tako dober (le počasnejši) pa bi bil kar navaden linearni seznam.

Na koncu se le zapeljemo čez povezave, ki smo jih naredili med gradnjo seznama, in izpišemo vse točke. Pazimo le, da med dvema možnima nadaljevanjema (vsako krajišče ima dve povezavi) izberemo pravo (napačna povezava je seveda tista, ki smo jo ravnokar uporabili).

```
program Daljice(Input, Output);

const Najvec = 1020;
type
  KoordT = integer;
  EnaT = record
    x, y: KoordT;
    Link1, Link2: integer;
  end;
  TockeT = array [0..Najvec] of EnaT;
var
  Tocke: TockeT;
  Zadnji: integer;

  procedure Init;
  var i: integer;
  begin
    for i := 0 to Najvec do with Tocke[i] do Link1 := -2;
  end; {Init}

  function Preberi(var x1, y1, x2, y2: KoordT): boolean;
  begin
    if Eof then Preberi := false
    else begin ReadLn(x1, y1, x2, y2); Preberi := true; end;
  end; {Preberi}

  function Vstavi(x1, y1: KoordT): integer;
  var Indeks, Prvi, Dodatek: integer;
      Nasel: boolean;
  begin
    Indeks := (x1 + y1) mod (Najvec + 1);
    if Tocke[Indeks].Link1 <> -2 then begin
      Nasel := (Tocke[Indeks].x = x1) and (Tocke[Indeks].y = y1);
      if not Nasel then begin
        Prvi := Indeks;
```

```
        Dodatek := 1 + ((x1 + y1) mod (Najvec − 1));
        repeat
          Indeks := (Indeks + Dodatek) mod (Najvec + 1);
          Nasel := (Tocke[Indeks].x = x1) and (Tocke[Indeks].y = y1);
        until (Tocke[Indeks].Link1 = −2) or (Indeks = Prvi) or Nasel;
        if (Indeks = Prvi) and (not Nasel) then
          begin WriteLn('Tabela je polna!'); Halt end;
      end; {if}
    end
    else Nasel := false;
    if not Nasel then with Tocke[Indeks] do begin
      Link1 := −1; Link2 := −1;
      x := x1; y := y1;
    end; {if}
    Vstavi := Indeks;
end; {Vstavi}

procedure PreberiUredi;
var xz, yz, xk, yk: KoordT;
    tz, tk: integer;
begin
  Zadnji := −1;
  while Preberi(xz, yz, xk, yk) do begin   { preberemo podatke o daljici }
    tz := Vstavi(xz, yz);                  { poiščemo/vstavimo podatke o prvem }
    tk := Vstavi(xk, yk);                  { in drugem krajišču }
    with Tocke[tz] do                      { povežemo prvo krajišče z drugim }
      if Link1 = −1 then Link1 := tk else Link2 := tk;
    with Tocke[tk] do                      { in drugo s prvim }
      if Link1 = −1 then Link1 := tz else Link2 := tz;
    Zadnji := tk;                          { zapomnimo si zadnje vstavljeno krajišče }
  end; {while}
end; {PreberiUredi}

procedure Izpisi;
var Tren, Nas: integer;
begin
  if Zadnji <> −1 then begin
    Tren := Zadnji;                        { izpisovati začnemo pri zadnjem krajišču }
    Nas := Tocke[Tren].Link1;
    if Nas <> −1 then repeat
      WriteLn('(', Tocke[Tren].x, ',', Tocke[Tren].y,
              ') - (', Tocke[Nas].x, ',', Tocke[Nas].y, ')');
      { premik v naslednje krajišče; pazimo, da nadaljujemo v pravi smeri }
      if Tren = Tocke[Nas].Link1
        then begin Tren := Nas; Nas := Tocke[Nas].Link2 end
        else begin Tren := Nas; Nas := Tocke[Nas].Link1 end;
    until (Nas = −1) or (Tren = Zadnji);
```

```
      if Nas = −1 then WriteLn('Link ni zaprt!');
    end; {if}
  end; {Izpisi}

begin {Daljice}
  Init;
  PreberiUredi;
  Izpisi;
end. {Daljice}
```

Podprogram Vstavi uporablja tabelo Tocke kot „zaprto“ razpršeno tabelo. Ko naletimo na neko točko (x, y) (kot krajišče neke daljice), si želimo čim hitreje ugotoviti, če smo na to točko že naleteli pri kakšni drugi daljici (kajti če je tako, moramo ti dve daljici povezati). Da nam ne bi bilo treba preiskovati cele tabele in primerjati vseh točk v njej z našo novo točko (x, y), se dogovorimo, da bomo točko (x, y) v tabeli vedno (če se bo le dalo) hranili na indeksu $(x+y) \bmod (n+1)$, pri čemer je $n+1$ dolžina naše tabele (indekse pa štejemo od 0 do n). Težava nastopi zaradi možnosti, da bi morali več točk hraniti na istem indeksu (na primer: (x, y) in (y, x), pa tudi $(x+1, y-1)$ in tako naprej). V takem primeru se premikajmo naprej po tabeli, dokler ne pridemo do prazne celice; gornji program dela skoke po $1 + ((x + y) \bmod (n - 1))$. Lepo je, če pri tem skakanju vemo, da bomo preizkusili vse možne indekse, preden bomo prišli nazaj na začetnega in nad vsem skupaj obupali; gornji program ima tabelo z 1021 elementi, kar je dobro, ker je 1021 praštevilo, tako da je dolžina skoka gotovo tuja številu 1021 in bomo res pristali na vseh možnih indeksih, preden bomo prišli nazaj na prvega. Kakorkoli že, med tem skakanjem po tabeli bomo prej ali slej prišli do prazne celice (razen če ni tabela že čisto polna), kamor lahko zdaj vpišemo svojo novo točko; lahko pa med tem skakanjem to točko tudi najdemo (in se s tem izkaže, da je točka že bila v tabeli, le da je takrat, ko smo jo vanjo dodajali, že nismo več mogli vpisati na prvotni indeks, pač pa smo takrat s skakanjem po tabeli prišli do neke prazne celice in jo vpisali tja).

Vsaka celica tabele ima prostor za indeksa Link1 in Link2, ki bosta na koncu pri vsaki točki kazala na indeksa drugih dveh krajišč tistih dveh daljic, ki se stikata v tej točki. To, da je celica še prazna, prepoznamo po tem, da imata Link1 in Link2 vrednost -2, če pa smo vanjo že vpisali neko točko, ki pa je še nismo povezali z drugima dvema krajiščema, sta Link2 in mogoče tudi Link1 enaka -1.

N: 160

R1993.3.3

Osnovna ideja za nalogo izhaja iz sistema Dynascope, katerega avtor je Rok Sosič. Sistem je namenjen predvsem opazovanju poljubnih programov med njihovim izvajanjem. Ena od funkcij je tudi vračanje izvajanja opazovanega programa na neko predhodno točko. Dejanska izvedba te funkcije je podobna kot v tem programu.

Pred izvajanjem vsakega ukaza moramo shraniti vrednosti tistih elementov računalnika, ki se pri tem ukazu spreminjajo (zapis PopravekT). Sem vsekakor sodi vrednost programskega števca (P), mogoče pa tudi vrednost akumulatorja (A) ali pa neke pomnilniške celice (M in njen Naslov). Kaj od tega je res treba shraniti, je odvisno od ukaza. Pri podprogramu UkazNazaj je treba le vpisati stare vrednosti (polje TipPopravka pove, katere vrednosti v zapisu so veljavne). Zapise hranimo v tabeli Sled, ki jo pravzaprav uporabljamo kot sklad — PredUkazom dodaja zapise na konec, UkazNazaj pa jih od tam briše.

```
program Dynascope(Output);

const MaxDolzinaSledi = 100; { največja dolžina shranjene sledi izvajanja }
type
  BesedaT = 0..65535;
  UkazT = (LDA, LDAI, STA, STAI, MSUB, MSUBI, JPOS, JPOSI);
  tpT = (tpAP, tpMNP, tpP);
  PopravekT = record                    { podatki, ki jih spreminjajo ukazi }
    P: BesedaT;                         { vedno shranimo programski števec }
    case TipPopravka: tpT of
      tpAP: (A: BesedaT);               { shranimo tudi akumulator }
      tpMNP: (M, Naslov: BesedaT);      { shranimo naslov in vsebino pomn. celice }
      tpP: ( );                         { shranimo le programski števec }
  end;
var
  M: array [BesedaT] of BesedaT;        { pomnilnik }
  P: BesedaT;                           { programski števec }
  A: BesedaT;                           { akumulator }
  Sled: array [1..MaxDolzinaSledi] of PopravekT; { sled izvajanja }
  DolzinaSledi: integer;                { število shranjenih ukazov v tabeli Sled }

  procedure Napaka(Sporocilo: string); external;

procedure PredUkazom;
var Korak: PopravekT;
    Ukaz, Operand: BesedaT;
begin
  Ukaz := M[P]; Operand := M[P + 1];
  Korak.P := P; { vsak ukaz spremeni vsaj P }
  case Ukaz of
    Ord(LDA), Ord(LDAI), Ord(MSUB), Ord(MSUBI): { ukazi spremenijo A in P }
      begin Korak.TipPopravka := tpAP; Korak.A := A end;
    Ord(STA), Ord(STAI): begin { ukaza spremenita pomnilniško celico in P }
      Korak.TipPopravka := tpMNP;
      if Ukaz = Ord(STA) then Korak.Naslov := Operand
      else Korak.Naslov := M[Operand];
      Korak.M := M[Korak.Naslov];
    end;
```

```
      Ord(JPOS), Ord(JPOSI): Korak.TipPopravka := tpP; { ukaza spremenita le P }
      else Napaka('neznan ukaz');
    end; {case}
    if DolzinaSledi >= MaxDolzinaSledi then Napaka('sled predolga');
    DolzinaSledi := DolzinaSledi + 1;          { podaljšamo sled izvajanja }
    Sled[DolzinaSledi] := Korak;               { spremembo stanja dodamo v sled }
end; {PredUkazom}

procedure UkazNazaj;
var Korak: PopravekT;
begin
    if DolzinaSledi < 1 then Napaka('sledi ni');
    Korak := Sled[DolzinaSledi];               { vzamemo zadnjo spremembo stanja }
    DolzinaSledi := DolzinaSledi - 1;          { skrajšamo sled izvajanja }
    P := Korak.P;                              { vedno popravimo P }
    case Korak.TipPopravka of
      tpAP: A := Korak.A;                      { popravimo A }
      tpMNP: M[Korak.Naslov] := Korak.M;       { popravimo pomnilniško celico }
      tpP: ;                                   { P smo že popravili }
    end; {case}
end; {UkazNazaj}
```

Mogoče bi bilo koristno program spremeniti tako, da v primeru, ko je tabela Sled že polna, ne bi javil napake, pač pa bi uporabljal Sled kot krožno tabelo in s podatki o novem ukazu preprosto povozil najstarejši zapis sledi. Tako bi sled vedno omogočala sledenje nazaj za zadnjih DolzinaSledi ukazov.

N: 161

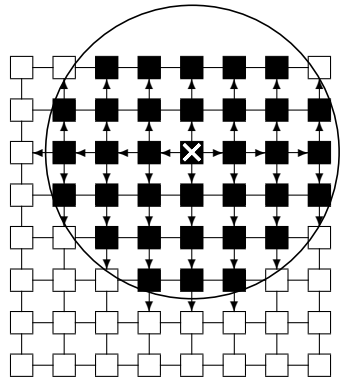
R1993.3.4

Uporabili bomo Pitagorov izrek. Točka pripada krogu, če velja $x^2 + y^2 \leq r^2$. Potrebujemo torej koordinati x in y vsake točke. Pare koordinat si procesorji sporočajo med seboj, zato smo v zapis SporociloT dodali polji x in y.

Procesorju v središču kroga dodelimo koordinati $(0, 0)$. Vsak procesor pošlje svoje koordinate sosedom, ki s pomočjo sprejetih koordinat in smeri, iz katere je sporočilo prispelo, izračunajo svoj položaj. r^2 izračuna samo procesor v središču kroga.

Sporočila pošiljamo pametno, tako da noben procesor ne sprejme več kot enega sporočila. Središčni procesor razširi sporočilo v vse štiri smeri, njegovi zahodni in vzhodni sosedje ga pošljejo naprej v vse smeri, razen v tisto, iz katere je sporočilo prišlo. Vsi ostali procesorji sporočilo le posredujejo severano (ali južno). Procesorji, ki ne ležijo v območju kroga, sporočila ne posredujejo dalje.

Širjenje sporočila si lahko ogledamo na naslednji sliki (s križcem je označen središčni procesor):



```
program Krog;

type SporociloT = record
                    r: real;
                    x, y: integer;
                  end;

var Sprejeto: SporociloT;
    Smer, Stevec: integer;

  procedure Poslji(Sporocilo: SporociloT; vSmer: integer); external;
  procedure Sprejmi(var Sporocilo: SporociloT; var izSmeri: integer); external;
  procedure Prizgi; external;

  { Ta podprogram bo tudi prižgal našo točko, če res leži v krogu. }
  function VKrogu: boolean;
  begin
    if Sprejeto.x * Sprejeto.x + Sprejeto.y * Sprejeto.y > Sprejeto.r
      then VKrogu := false
      else begin Prizgi; VKrogu := true end;
  end; {VKrogu}

begin
  repeat
    Sprejmi(Sprejeto, Smer);
    case Smer of
      0: begin { središče kroga }
        Sprejeto.x := 0; Sprejeto.y := 0;
        Sprejeto.r := Sprejeto.r * Sprejeto.r;
        Prizgi;
        for Stevec := 1 to 4 do Poslji(Sprejeto, Stevec);
      end;
      2, 4: begin { vzhodno ali zahodno od središča }
        Sprejeto.x := Sprejeto.x + Smer - 3;
```

```
          if VKrogu then { obvestimo tri sosede }
            for Stevec := 1 to 4 do if Stevec <> Smer then
              Poslji(Sprejeto, Stevec);
        end;
        1, 3: begin { severno ali južno }
          Sprejeto.y := Sprejeto.y + Smer - 2;
          if VKrogu then Poslji(Sprejeto, 4 - Smer);
        end;
      end; {case}
    until false;
end. {Krog}
```

18. državno tekmovanje v znanju računalništva (1994)

NALOGE ZA PRVO SKUPINO

1994.1.1 Recimo, da je naš računalnik tak, kot so jih imeli pred mnogimi leti, in pozna samo števke od 0 do 9. Z njimi lahko seveda zapišemo tudi večja števila, vendar pri črkah že odpove. R: 187

Naloga zahteva, da naš računalnik naučimo zapisovati tudi črke. Ker črk ne pozna, si bomo pomagali s trikom in črke zapisali s številkami. **Opiši postopek**, kako bi zapisoval črke s pomočjo števk in to tako, da bi porabil za neko besedilo kar najmanj števk.

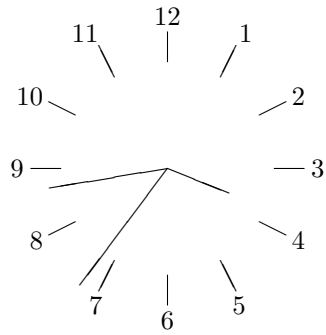
1994.1.2 **Napiši podprogram** Pretvori, ki dobi kot parameter tabelo znakov in njeno dolžino in ju predela tako, da spremeni ubežna zaporedja v znake. Ubežno zaporedje se začne z znakom `'\'`, ki mu sledijo do 3 števke (znaki od `'0'` do `'9'`). Števke sestavljajo kodo znaka, s katerim moramo nadomestiti ubežno zaporedje. Če znaku `'\'` ne sledi števka, ga nadomestimo z znakom s kodo 0. R: 188

Primeri: `'abc\100efg'` → `'abcdefg'`; `'\1000'` → `'d0'`; `'A\66C'` → `'ABC'`.

Uporabljaj naslednje deklaracije:

```
const MaksDolzinaNiza = ...; { največja dolžina niza }
type NizT = array [1..MaksDolzinaNiza] of char;
procedure Pretvori(var Niz: NizT; var DolzinaNiza: integer);
```

1994.1.3 **Napiši program**, ki riše uro s kazalci, ki teče. Ura naj bi izgledala podobno kot televizijska ura. Na uri morajo biti sekundni, minutni in urni kazalec. Dimenzije zaslona so: $X = 1..1000$, $Y = 1..1000$. Na začetku je zaslon prazen. Na razpolago imaš naslednje podprograme: R: 189



- **procedure** Cas(**var** Ura, Minuta, Sekunda: integer);
  Vrne trenutni čas.
- **procedure** PojdiNa(X, Y: integer);
  Pero na zaslonu postavi na točko (X, Y).
- **procedure** Naprej(K: real; D: integer);
  Pero premakne pod kotom K za dolžino D naprej. Način risanja pove, ali pero pri tem riše ali briše.
- **procedure** Risi; **procedure** Brisi;
  Način risanja peresa postavita na risanje oz. brisanje.

R: 191

1994.1.4

Sestavi funkcijski podprogram PodNiz(t, s), ki vrne *resnično* (true) natanko takrat, ko lahko dobimo besedo t iz besede s, če v besedi s prečrtamo nekaj (nič ali več) znakov. Lahko predpostavimo, da se beseda v tabeli konča s presledkom. Primera:

PodNiz('banana ', 'ljubljančanka ') = true
PodNiz('pero ', 'koper ') = false

Uporabljaj naslednje deklaracije:

```
const MaksDolzinaNiza = ...; { največja dolžina niza }
type NizT = array [1..MaksDolzinaNiza] of char;
function PodNiz(t, s: NizT): boolean;
```

NALOGE ZA DRUGO SKUPINO

R: 192

1994.2.1

Napiši funkcijski podprogram, ki za dano besedo vrne število zlogov v njej. Besedo vedno sestavlja vsaj en zlog. Vsak samoglasnik določa svoj zlog. Črko *r*, ki jo obdajata dva soglasnika, štejemo kot samoglasnik.[30] Primeri:

Stej('matematika ') = 5; Stej('z ') = 1; Stej('prstan ') = 2.

Uporabljaj naslednje deklaracije:

```
const MaksDolzinaNiza = ...; { največja dolžina niza }
type NizT = array [1..MaksDolzinaNiza] of char;
function Stej(Niz: NizT): integer;
```

Predpostaviš lahko, da je v tabeli Niz takoj za koncem besede vsaj en presledek (znak ' ').

[30] Malo bolj problematični so primeri, ko se beseda začne na *r*, temu pa sledi soglasnik. Na primer, *rjast* ima dva zloga in ne enega, *rjavolas* pa verjetno tri in ne štiri. Tvoj podprogram naj take *r*-je razglasi ali za samoglasnike ali pa za soglasnike, kakor se ti zdi pač bolj prikladno.

1994.2.2

Imamo števili $n > 0$ in $k \geq 0$. Sestavimo tabelo z n elementi, ki imajo vrednosti $-k, \ldots, k$. Če je vsota elementov tabele 0, imenujemo ta vektor *n-k-sestavljanka*. R: 193

Opiši postopek, ki prebere števili n in k ter izpiše vse n-k-sestavljanke. Postopek naj deluje čim hitreje. Primer vseh 3-2-sestavljank:

$$\begin{array}{lllll} (-2,0,2); & (-2,1,1); & (-2,2,0); & (-1,-1,2); & (-1,0,1); \\ (-1,1,0); & (-1,2,-1); & (0,-2,2); & (0,-1,1); & (0,0,0); \\ (0,1,-1); & (0,2,-2); & (1,-2,1); & (1,-1,0); & (1,0,-1); \\ (1,1,-2); & (2,-2,0); & (2,-1,-1); & (2,0,-2). & \end{array}$$

1994.2.3

Z računalnikom krmilimo uro v zvoniku. Mehanizem skrbi za usklajeno pomikanje obeh kazalcev, naloga računalnika pa je, da vsako minuto ukaže mehanizmu, naj premakne kazalce za eno minuto, ter da ob vsaki polni uri odbije uro (število udarcev zvona mora biti enako uri: točno ob enih en udarec, ob dveh dva, ..., opoldne in opolnoči 12 udarcev; udarci naj si sledijo v razmiku treh sekund). R: 194

Na voljo imaš naslednja podprograma:

- PremakniKazalce ob vsakem klicu premakne kazalce za eno minuto naprej;
- UdariNaZvon sproži en udarec mehanizma na zvon.

Operacijski sistem računalnika vsako sekundo pokliče podprogram VsakoSekundo, ki upravlja z uro. **Napiši ta podprogram.**

Zaradi enostavnosti predpostavimo, da se program požene opolnoči (prvi klic tvojega podprograma bo že kar takoj opolnoči in že takoj je treba odbiti uro strahov) in da ob startu programa kazalci kažejo polnoč. Tvoj podprogram lahko uporablja globalne (ali lastne statične) spremenljivke, ki jim lahko tudi predpišeš začetno vrednost.

1994.2.4

Napiši program ali podroben algoritem, ki pobarva grafični zaslon s črnimi pikami v naključnem vrstnem redu. Na voljo imaš naslednje parametre in pomožne podprograme: R: 195

- Velikost zaslona je XMax × YMax.
- Podprogram Pobarvaj(x, y) pobarva točko s koordinatami (x, y) s črno barvo.
- Funkcija Pobarvana(x, y) vrne true, če je točka (x, y) že pobarvana s črno barvo, sicer pa false.
- Funkcija Random(n) vrne naključno število med vključno 0 in n − 1.

Program mora vsako točko pobarvati natanko enkrat, ko so vse točke pobarvane črno, pa naj se ustavi. Privzeti smeš, da na začetku na zaslonu ni črnih točk. Točke mora barvati **enakomerno** in nakljućno. Program naj bo hiter in naj ne porabi veliko dodatnega pomnilnika.[31] Upoštevaj, da je tipična velikost zaslona 1280×1024.

NALOGE ZA TRETJO SKUPINO

R: 200

1994.3.1

Zaradi enostavnosti se domenimo, da „datoteka dolžine N“ pomeni niz bitov dolžine N (datoteke pri tej nalogi torej nimajo imen).

a) Program P za kompresijo podatkov je tak program, ki vsaki datoteki D priredi neko drugo datoteko $P(D)$. Seveda programu za kompresijo pripada tudi program $\bar{P}$ za dekompresijo podatkov, ki kompresirani datoteki $P(D)$ priredi prvotno datoteko $D = \bar{P}(P(D))$. **Ugotovi**, kakšnim **minimalnim** zahtevam morata zadostovati programa P in $\bar{P}$ za pravilno delovanje, to je, za vsako datoteko D mora veljati $\bar{P}(P(D)) = D$.

b) S programi za kompresijo podatkov skušamo prihraniti prostor na disku. Naj bo P program za kompresijo podatkov. Njegovo učinkovitost na datotekah, krajših od N, ocenimo takole: naj bo $\mathcal{D}$ množica *vseh* datotek, katerih dolžina ne presega N. Skupna dolžina vseh datotek v $\mathcal{D}$ naj bo M. Vsako datoteko iz $\mathcal{D}$ skomprimiramo s programom P in dobimo novo množico komprimiranih datotek $\mathcal{E}$. Njihova skupna dolžina naj bo m. Razmerju $(M - m)/M$ pravimo *učinkovitost programa P na datotekah, krajših od N*. **Ugotovi**, kakšna je najboljša možna učinkovitost programov za kompresijo.

c) Upoštevaje rezultat iz točke (b), **opiši** kak program za kompresijo, ki ima najboljšo možno učinkovitost.

d) Ali je definicija učinkovitosti programov za kompresijo iz točke (b) uporabna v praksi? **Odgovor** utemelji. Predlagaj, kako bi v praksi izmerili učinkovitost programa za kompresijo.

R: 202

1994.3.2

Na pošti so prejeli napravo, ki za posamezno pošiljko določi njeno poštnino (skupna vrednost znamk, ki morajo biti na pošiljki).

[31] Radi bi na primer, da bi program porabil veliko manj pomnilniških celic, kot pa je točk na zaslonu. Z enakomernostjo barvanja pa hočemo reči, naj bodo točke, ki jih program do posameznega trenutka pobarva, približno enakomerno razpršene po zaslonu (delež pobarvanih točk naj torej ne bo npr. na enem koncu zaslona znatno večji kot na drugem).

Napiši algoritem, ki bo iz danih vrednosti znamk in poštnine izpisal niz vrednosti znamk (lahko tudi več enakih), ki jih je potrebno uporabiti.

Če ne obstaja niz znamk, ki točno pokrije poštnino, potem je rešitev niz, katerega skupna vrednost znamk preseže poštnino za najmanjši možni znesek. V primeru, da različni nizi znamk pokrivajo enako poštnino, je rešitev niz, ki vsebuje najmanj znamk. Primeri:

če so vrednosti znamk: 2, 7, 14, 17, 22, 63, 98,

so rešitve nekaterih poštnin takšne:

72 — 2, 7, 63; 86 — 2, 7, 14, 63; 143 — 17, 63, 63; 5 — 2, 2, 2.

1994.3.3

R: 205

Vhod v garažno hišo je opremljen z velikimi drsnimi vrati, ki so namenjena vstopu in izstopu avtomobilov in pešcev v garažno hišo. Za prehod pešcev je dovolj, da se vrata odprejo do polovice širine, za avtomobil pa je potrebno odpreti vrata v celotni širini.

Pri izstopu iz objekta pešec pritisne tipko za odpiranje vrat; prisotnost avtomobila, ki želi odpeljati iz garažne hiše, pa začuti tipalo v asfaltu (ki ni občutljivo na pešce).

Pri vstopu imamo en sam način aktiviranja odpiranja vrat — s kontaktno ključavnico. Pri tem ne moremo ločiti vstopajočega pešca od avtomobila, zato vrata vedno odpremo v celotni širini.

Tik ob vratih je varovalno optično tipalo, ki v primeru ovire med vrati zagotavlja, da ostanejo vrata odprta oziroma da se ponovno razprejo do prvotne širine (pešec/avto). Popolnoma zaprtih vrat ni mogoče odpreti z aktiviranjem tega tipala.

Vrata naj ostanejo odprta 15 sekund, potem naj se samodejno zaprejo. Vsaka prisotnost ovire v tem času ali nov ukaz za odpiranje nastavi preostali čas odprtosti ponovno na 15 sekund.

Napiši program, ki bo ob upoštevanju tipal krmilil motor za pomik vrat. Delovanje naj bo smiselno tudi v primeru tesno si sledečih vstopov/izstopov pešcev in avtomobilov.

Na voljo imaš naslednje prodprograme:

- OdprtostVrat je funkcija, ki vrne realno število med 0 in 1, ki pove lego vrat:
  0 — zaprta vrata;
  0,5 ali več — vrata so dovolj odprta za prehod pešca;
  1 — popolnoma odprta vrata, primerna za vstop/izstop avtomobila;

- IzstopPesec je funkcija, ki vrne true, dokler je pritisnjena tipka za odpiranje vrat z notranje strani, sicer vrne false;

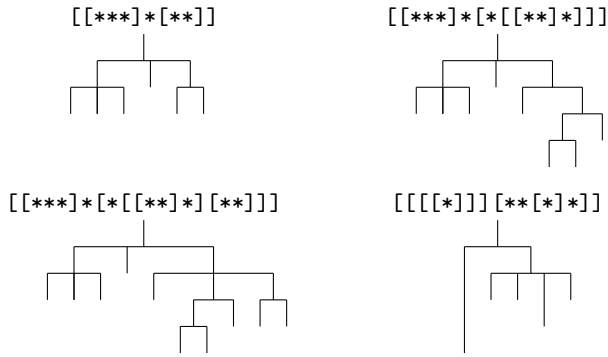
- IzstopAvto je funkcija, ki vrne true, dokler tipalo na notranji strani vrat čuti prisotnost avtomobila, ki čaka na izvoz (sicer vrne false);
- Vstop je funkcija, ki vrne true, če je aktivirana kontaktna ključavnica za odpiranje vrat z zunanje strani, sicer vrne false;
- Ovira je funkcija, ki vrne true, če je med vrati prisotna ovira (avto, pešec ali kaj drugega), sicer vrne false;
- Motor(Ukaz) je podprogram, ki upravlja z motorjem:
  Motor(Stoj) — izklopi motor, vrata se ustavijo v trenutni legi;
  Motor(Odpiraj) — vklopi motor, vrata se odpirajo (ali ostajajo odprta);
  Motor(Zapiraj) — vklopi motor, vrata se zapirajo (ali ostajajo zaprta);
- NastaviBudilko(s) je podprogram, s katerim nastavimo odštevalno uro na s sekund (glej podprogram PreostaliCas);
- PreostaliCas je funkcija, ki vrne preostali čas (v sekundah) od začetnega, ki smo ga nastavili s podprogramom NastaviBudilko(s). Če se je čas že iztekel, vrača 0.

R: 207

1994.3.4 Sestavi program

, ki nariše drevo danega oklepajnega izraza tako, kot je prikazano v primerih. Oklepajni izraz je niz znakov `[`, `]` in `*`, pri katerem so oklepaji in zaklepaji pravilno gnezdeni. Predpostavi, da so vhodni nizi, s katerimi boš delal, pravilno oblikovani. Za risanje je na voljo podprogram Crta(x1, y1, x2, y2: real), ki nariše črto od točke (x1, y1) do točke (x2, y2).[32]

Primeri:



[32] Mimogrede, še ena zanimiva naloga z risanjem dreves (v tekstovnem načinu) je "BUT we need a diagram" z ACMovega študentskega tekmovanja v programiranju za azijsko regijo (Tokio, 23. nov. 1998, problem F; #692 v zbirki na `online-judge.uva.es`).

REŠITVE NALOG ZA PRVO SKUPINO

R1994.1.1

N: 181

Črke bomo desetiško kodirali (danes to delamo seveda binarno). Odločiti se moramo, ali bomo kodirali samo velike (25 črk) ali velike in male črke (50 črk). No, v obeh primerih je rešitev podobna.

Prva rešitev, ki nam pride na misel, je, da črke označimo s številkami od 1 do 25 (oziroma od 1 do 50). V tem primeru porabimo za n črk dolgo besedilo $2n$ števk. Temu bi lahko rekli dekadno kodiranje. Obstajajo pa še boljše rešitve in v teh je vsa umetnost kodiranja in hkrati tudi komprimiranja.

Uredimo vse črke po parih; tako dobimo $25 \times 25 = 625$ parov. Te lahko zapišemo s številkami od 1 do 625, to je s trimestnimi števili. V tem primeru porabimo za n črk dolgo besedilo $(n/2) \cdot 3 = 1{,}5\,n$ števk, kar je že lep prihranek v primerjavi s prvo rešitvijo.

Če uredimo črke po trojicah, dobimo $25 \times 25 \times 25 = 15625$ trojic. Le-te zapišemo s petmestnimi števili od 1 do 15625. Vendar, glej ga zlomka, za n črk dolgo besedilo bomo sedaj porabili $(n/3) \cdot 5 = 1{,}66\,n$ številk, kar pa je slabše kot v prvem primeru.

Če tako nadaljujemo, ugotovimo, da je črke še bolje urediti po peterkah, ki jih je 9 765 625 in jih torej lahko zapišemo s števili od 1 do 9 765 625. Za n črk dolgo besedilo bomo porabili $(n/5) \cdot 7 = 1{,}4\,n$ števk.

Ta razmislek lahko tudi posplošimo: če združujemo po k črk v skupine, je povprečna poraba prostora $f(k) = \lceil \log 25^k \rceil / k$ števk na vsako črko. Zgoraj smo ugotovili, da je $f(9) = 1{,}4$, vendar se dá pri večjih k najti še ugodnejša razmerja. Naslednjih nekaj k, pri katerih doseže $f(k)$ svojo najmanjšo vrednost doslej:

k	$f(k)$		približna vrednost
98	137/98	$\approx$	1,397959184
583	815/583	$\approx$	1,397941681
1068	1493/1068	$\approx$	1,397940075
14369	20087/14369	$\approx$	1,397940010
teoretična spodnja meja	$\log 25$	$\approx$	1,397940009

Iz definicije f in dejstva, da je $x \le \lceil x \rceil < x + 1$, sledi $\log 25 \le f(k) \le \log 25 + 1/k$; torej se lahko spodnji meji $\log 25$ približamo poljubno natančno, če smo le pripravljeni vzeti dovolj velik k. Ne moremo pa je čisto doseči, kajti to bi zahtevalo, da je neka potenca števila 25 hkrati tudi potenca števila 10, torej liha in soda obenem. Podobno lahko razmišljamo tudi, če moramo kodirati poleg velikih tudi male črke (namesto 25 vzamemo 50). Vsekakor pa bi takšno

kodiranje, če bi vzeli prevelik k, postalo nepraktično, saj bi zahtevalo preveč računanja z velikimi celimi števili.

Še ena možnost za varčevanje s prostorom pri kodiranju črk bi bila, da bi pogostejšim črkam (ali skupinam k črk) dodelili krajše kode (take z manj števkami), redkejšim pa daljše. To je koristno, če lahko (in običajno lahko) s krajšimi kodami pri pogostejših skupinah črk več pridobimo, kot pa bomo z daljšimi kodami pri redkejših skupinah črk izgubili. Paziti moramo le na to, da ne bo nobena koda podaljšek kakšne druge, saj bi bilo drugače tako kodirano besedilo težko (ali pa sploh nemogoče) nedvoumno dekodirati. Znan primer postopka, ki poišče takšne kode, se imenuje *Huffmanovo kodiranje*. Slabost takega kodiranja je, da moramo poznati pogostosti posameznih črk (ali skupin črk), te pa so odvisne od tega, na kakšnem besedilu (ali, še bolje: skupini besedil) smo jih merili; zato kod, ki je dober za eno besedilo, mogoče ni tako dober za neko drugo, v katerem so pogostosti črk drugačne (npr. ker je v drugem jeziku).

N: 181

R1994.1.2

S številcem i se sprehodimo po danem nizu, številec j pa nam kaže, koliko znakov se je že nabralo v predelani različici niza. (Ker se niz ob predelovanju lahko le krajša, nikoli pa se ne podaljšuje, lahko odlagamo znake kar v isti niz, saj vemo, da bomo s tem vedno pisali čez tiste dele niza, ki smo jih že prebrali in nas ne zanimajo več.) Ko pri sprehajanju po nizu naletimo na znak `'\'`, preberemo še naslednjih nekaj števk in izluščimo celo število, ki ga predstavljajo, ter dodamo pripadajoči znak v predelani niz.

```
program UbezniZnaki;
const
  MaksDolzinaNiza = 10; { največja dolžina niza }
  MaksStevk = 3;        { največje število števk za znakom '\' }
type
  NizT = array [1..MaksDolzinaNiza] of Char;
var
  Niz: NizT;            { niz znakov za pretvorbo }
  DolzinaNiza: integer; { dolžina niza }

procedure Pretvori(var Niz: NizT; var DolzinaNiza: integer);
var
  i, j, k: integer;     { števci }
  c: integer;           { koda znaka }
begin
  i := 1; j := 0;
  while i <= DolzinaNiza do begin
    j := j + 1;
    if Niz[i] = '\' then begin
      i := i + 1; c := 0; k := 1;
```

```
        while (i <= DolzinaNiza) and (k <= MaksStevk) and
              (Niz[i] in ['0'..'9']) do begin
          c := c * 10 + Ord(Niz[i]) - Ord('0');
          i := i + 1; k := k + 1;
        end; {while};
        Niz[j] := Chr(c);
      end else begin
        Niz[j] := Niz[i]; i := i + 1;
      end; {if}
    end; {while}
    DolzinaNiza := j;
end; {Pretvori}

procedure IzpisiNiz(Niz: NizT; DolzinaNiza: integer);
var i: integer;
begin
    for i := 1 to DolzinaNiza do Write(Niz[i]);
    WriteLn;
end; {IzpisiNiz}

begin {UbezniZnaki}
    Niz := 'abc\100efg'; DolzinaNiza := 10;
    IzpisiNiz(Niz, DolzinaNiza); { izpiše: 'abc\100efg' }
    Pretvori(Niz, DolzinaNiza);
    IzpisiNiz(Niz, DolzinaNiza); { izpiše: 'abcdefg' }
end. {UbezniZnaki}
```

R1994.1.3

Vsako sekundo narišemo uro na zaslon, počakamo do konca te sekunde in nato uro zbrišemo; potem jo lahko narišemo s kazalci v novem položaju. Na začetku lahko narišemo še črtice na vsakih 30 stopinj in ob njih še številke za ure; dobro je, da so dovolj daleč od središča ure, tako da jih ob risanju in brisanju kazalcev ne bomo poškodovali. Spodnji program vsebuje tudi implementacijo podprogramov za delo z grafiko, ki jih omenja naloga (za Turbo Pascal).

N: 181

```
program UraSKazalci;
uses Dos, Graph, Crt;
var
    PeroX, PeroY: integer;
    Ura, Minuta, Sekunda, NovaUra, NovaMinuta, NovaSekunda: integer;

procedure Cas(var Ura, Minuta, Sekunda: integer);
var WUra, WMinuta, WSekunda, WStotinka: word;
begin
    GetTime(WUra, WMinuta, WSekunda, WStotinka);
    Ura := WUra; Minuta := WMinuta; Sekunda := WSekunda;
end; {Cas}
```

```
procedure OdpriGraficniNacin;
var GDrv, GMode: integer;
begin
  GDrv := Detect; InitGraph(GDrv, GMode, 'c:\bp\bgi');
  if GraphResult <> grOk then Halt(1);
  PeroX := 1; PeroY := 1; SetColor(White);
end; {OdpriGraficniNacin}

procedure ZapriGraficniNacin;
begin
  CloseGraph;
end; {ZapriGraficniNacin}

procedure PojdiNa(X, Y: integer);
begin
  PeroX := X; PeroY := Y;
end; {PojdiNa}

procedure Naprej(K: real; D: integer);
var NoviPeroX, NoviPeroY: integer;
begin
  NoviPeroX := PeroX + Round(D * Cos(Pi * (K - 90) / 180));
  NoviPeroY := PeroY + Round(D * Sin(Pi * (K - 90) / 180));
  Line(PeroX, PeroY, NoviPeroX, NoviPeroY);
  PeroX := NoviPeroX; PeroY := NoviPeroY;
end; {Naprej}

procedure Pisi(Stevilo: integer);
var Niz: string;
begin
  Str(Stevilo, Niz);
  OutTextXY(Trunc(Round(PeroX - 0.5 * TextWidth(Niz))),
            Trunc(Round(PeroY - 0.5 * TextHeight(Niz))), Niz);
end; {Pisi}

procedure Risi;
  begin SetColor(White) end;

procedure Brisi;
  begin SetColor(Black) end;

var
  Sirina, Visina, X0, Y0, R: integer;
  KazU, KazM, KazS: integer;                    { dolžine kazalcev }
begin {UraSKazalci}
  OdpriGraficniNacin;
  Sirina := GetMaxX + 1; Visina := GetMaxY + 1;  { velikost zaslona }
  X0 := Sirina div 2; Y0 := Visina div 2;        { središče zaslona (in ure) }
```

```
  if X0 > Y0 then R := Y0 else R := X0;
  R := Trunc(Round(0.95 * R));                    { to bo zunanji polmer ure }
  { Določimo velikost kazalcev. }
  KazU := 10 * R div 20; KazM := 12 * R div 20; KazS := 13 * R div 20;

  for Ura := 1 to 12 do begin                     { Narišimo številke in črtice ob njih. }
    PojdiNa(X0, Y0);
    Brisi; Naprej(Ura * 30, 14 * R div 20);
    Risi; Naprej(Ura * 30, 2 * R div 20);
    Brisi; Naprej(Ura * 30, R div 20);
    Risi; Pisi(Ura);
  end; {for}

  Cas(Ura, Minuta, Sekunda);
  while not KeyPressed do begin
    Risi;                                         { Narišimo kazalce v trenutnem položaju. }
    PojdiNa(X0, Y0); Naprej((Ura + Minuta / 60) * 30, KazU);
    PojdiNa(X0, Y0); Naprej(Minuta * 6, KazM);
    PojdiNa(X0, Y0); Naprej(Sekunda * 6, KazS);

    repeat                                        { Počakajmo do konca te sekunde. }
      Cas(NovaUra, NovaMinuta, NovaSekunda);
    until NovaSekunda <> Sekunda;

    Brisi;                                        { Pobrišimo kazalce v starem položaju. }
    PojdiNa(X0, Y0); Naprej((Ura + Minuta / 60) * 30, KazU);
    PojdiNa(X0, Y0); Naprej(Minuta * 6, KazM);
    PojdiNa(X0, Y0); Naprej(Sekunda * 6, KazS);
    Ura := NovaUra; Minuta := NovaMinuta; Sekunda := NovaSekunda;
  end; {while}

  ZapriGraficniNacin;
end. {UraSKazalci}
```

R1994.1.4

Zahteva, naj bo mogoče dobiti t iz s z brisanjem nekaj črk, je enakovredna zahtevi, da se morajo v s-ju pojavljati vse črke iz t-ja (in to v pravem vrstnem redu), vmes pa so lahko še kakšne druge črke (ki bi jih morali iz s-ja zbrisati, da bi dobili t). Torej se lahko z zanko (števec i v spodnjem programu) premikamo po nizu s in si zapomnimo (števec j), koliko prvih črk niza t smo že zagledali. Vsakič, ko zagledamo naslednjo črko niza t, povečamo vrednost j. Če pridemo do konca s-ja prej kot do konca t-ja, vemo, da t ni s-jev podniz, sicer pa je.

N: 182

```
program Banana;

const MaksDolzinaNiza = 15; { največja dolžina niza }
type NizT = array [1..MaksDolzinaNiza] of char;

function PodNiz(t, s: NizT): boolean;
var i, j: integer;
```

```
begin
  i := 1; j := 1;
  while (s[i] <> ' ') and (t[j] <> ' ') do begin
    if s[i] = t[j] then j := j + 1;
    i := i + 1;
  end; {while};
  PodNiz := t[j] = ' ';
end; {PodNiz}

begin
  WriteLn(PodNiz('banana ', 'ljubljančanka '));
  WriteLn(PodNiz('pero ', 'koper '));
end. {Banana}
```

REŠITVE NALOG ZA DRUGO SKUPINO

N: 182

R1994.2.1

Spodnji podprogram si poenostavi delo s predpostavko, da se beseda, ki nas zanima, začenja šele v drugi celici tabele s, v prvi celici pa je presledek (pravzaprav je dovolj že, če ni samoglasnik); predpostavi tudi, da je za besedo v tabeli vsaj en presledek. To nam poenostavi preverjanje, kakšni črki stojita ob r-ju, v primerih, ko je r na začetku ali na koncu besede. Uporabljeni pogoj, da ob r-ju ne sme biti samoglasnikov, če naj nastane nov zlog, pomeni, da bomo tudi r na začetku besede razglasili za samoglasnik, če takoj za njim stoji soglasnik.

```
program Zlogi;
const MaksDolzinaNiza = 15; { največja dolzina niza }
type NizT = array [1..MaksDolzinaNiza] of char;

function Samoglasnik(Crka: char): boolean;
begin
  Samoglasnik := Crka in ['a', 'e', 'i', 'o', 'u'];
end; {Samoglasnik}

function StZlogov(s: NizT): integer;
var i, z: integer;
begin
  i := 2; z := 0;
  while s[i] <> ' ' do begin
    if Samoglasnik(s[i]) then z := z + 1
    else if (s[i] = 'r') and
            not (Samoglasnik(s[i - 1]) or Samoglasnik(s[i + 1])) then z := z + 1;
    i := i + 1;
  end; {while}
  if z = 0 then StZlogov := 1 else StZlogov := z;
end; {Stej}
```

```
begin {Zlogi}
  WriteLn(StZlogov(' matematika '));
  WriteLn(StZlogov(' z '));
  WriteLn(StZlogov(' prstan '));
end. {Zlogi}
```

R1994.2.2

N: 183

Recimo, da smo določili v tabeli že prvih $m-1$ števil in dobili vsoto s; če bi zdaj za naslednje število vzeli i, bi bila vsota $s+i$. Ostane nam še $n-m$ števil, ki morajo biti vsa iz množice $\{-k,\ldots,k\}$, torej bo vsota vseh n števil na koncu vsaj $s+i-(n-m)k$ in kvečjemu $s+i+(n-m)k$. Mi pa bi radi, da bi bila vsota na koncu lahko 0. Torej mora veljati

$$s+i-(n-m)k \le 0 \le s+i+(n-m)k,$$

iz česar sledi pogoj

$$-(n-m)k-s \le i \le (n-m)k-s.$$

Drugih vrednosti i torej nima smisla postavljati na m-to mesto, ker v nadaljevanju nikakor ne bi mogli dobiti vsote 0. S to neenakostjo si lahko spodnji program pomaga, da se izogne nepotrebnim rekurzivnim klicem.

```
program Sestavljanka;

const MaxN = 20;
var n, k, i: integer;
    Tabela: array [1..MaxN] of integer;

function Min(a, b: integer): integer;
  begin if a < b then Min := a else Min := b end;

function Max(a, b: integer): integer;
  begin if a > b then Max := a else Max := b end;

procedure Pregled(Globina, Vsota: integer);
var
  Meja1, Meja2, i: integer;
begin
  if Globina = n + 1 then begin
    Write('(');
    for i := 1 to n do begin
      Write(Tabela[i]);
      if i < n then Write(',');
    end; {for}
    WriteLn(')');
  end else begin
```

```
      Meja1 := Max((Globina − n) * k − Vsota, −k);
      Meja2 := Min((n − Globina) * k − Vsota, k);
      for i := Meja1 to Meja2 do begin
        Tabela[Globina] := i;
        Pregled(Globina + 1, Vsota + i);
      end; {for}
    end; {if}
end; {Pregled}

begin {Sestavljanka}
  ReadLn(n, k);
  for i := −k to k do begin
    Tabela[1] := i;
    Pregled(2, i);
  end; {for}
end. {Sestavljanka}
```

Kot kaže spodnja tabela, število n-k-sestavljank kar hitro narašča:

$n=$	0	1	2	3	4	5	6	7	8	9	10
$k=1$	1	1	3	7	19	51	141	393	1 107	3 139	8 953
2	1	1	5	19	85	381	1 751	8 135	38 165	180 325	856 945
3	1	1	7	37	231	1 451	9 331	60 691	398 567	2 636 263	17 538 157

Če jih hočemo le prešteti, ni treba, da vse naštejemo, tako kot to počne gornji program. Lahko pridemo kar do rekurzivne formule za število sestavljank, le problem moramo malo posplošiti: naj bo $f(n,k,s)$ število n-k-sestavljank z vsoto s. Razmislek, ki smo ga uporabili tudi pri pisanju programa, nam pokaže, da je

$$f(n,k,s) = \sum_{t=\max\{-(n-1)k,\,s-k\}}^{\min\{(n-1)k,\,s+k\}} f(n-1,k,t)$$

za $n > 0$; trivialna primera sta $f(0,k,0) = 1$ in $f(n,k,s) = 0$ za $|s| > nk$. Z metodo rodovnih funkcij se lahko prepričamo, da je $f(n,k,s)$ ravno koeficient pri členu x^{s+nk} v polinomu $(1 + x + x^2 + \ldots + x^{2k})^n$.

N: 183

R1994.2.3

Program si lahko v globalnih spremenljivkah zapomni, koliko manjka do naslednje polne ure ali pa do naslednjega udarca na zvon in kolikokrat je še treba pozvoniti.

```
var
  DoPomika: integer value 60;      { sekund do pomika kazalcev }
  DoPolneUre: integer value 0;     { sekund do polne ure }
  DoUdarca: integer value 0;       { sekund do udarca na zvon }
  Udarcev: integer value 0;        { potrebno odbiti še toliko udarcev }
```

```
  Ura: integer value 11;                { koliko je ura }

  procedure PremakniKazalce; external;
  procedure UdariNaZvon; external;

procedure VsakoSekundo;
begin
  if DoPomika > 0 then DoPomika := DoPomika - 1
  else begin PremakniKazalce; DoPomika := 59 end;
  if DoPolneUre > 0 then DoPolneUre := DoPolneUre - 1
  else begin
    if Ura < 12 then Ura := Ura + 1 else Ura := 1;
    Udarcev := Ura; DoPolneUre := 3600 - 1;
  end; {if}
  if DoUdarca > 0 then DoUdarca := DoUdarca - 1
  else if Udarcev > 0 then begin
    UdariNaZvon; Udarcev := Udarcev - 1; DoUdarca := 2;
  end; {if}
end; {VsakoSekundo}
```

R1994.2.4

N: 183

Če bi bil zaslon majhen, recimo velikosti 80×25, bi lahko uporabili naslednji algoritem. Točke na zaslonu uredimo po vrsti tako, da uredimo vrstice od zgoraj navzdol in točke v vrstici od leve proti desni. Nato izvajamo naslednje korake:

(1) i := število točk na zaslonu;
(2) r := Random(i);
(3) Po vrsti prebiraj točke na zaslonu.
    Ko naletiš na r-to nepobarvano točko, jo pobarvaj.
(4) i := i − 1;
(5) Če je i > 0, skoči na korak (2), sicer končaj.

Za velik zaslon se ta algoritem ne obnese, ker traja korak (3) predolgo. Pomagamo si tako, da si zapomnimo, koliko točk je treba še pobarvati v vsaki posamezni vrstici. V koraku (3) lahko torej zelo hitro določimo vrstico, v kateri je r-ta točka.

```
const XMax = 1280; YMax = 1024;

function Random(n: integer): integer; external;
procedure Pobarvaj(x, y: integer); external;
function Pobarvana(x, y: integer): boolean; external;

procedure PobarvajZaslon1;
var i, r, x, y: integer;
    Vrstica: array [0..YMax - 1] of integer;
begin
```

```
  for i := 0 to YMax − 1 do Vrstica[i] := XMax;
  for i := XMax * YMax downto 1 do begin
    r := Random(i); y := 0;
    while r >= Vrstica[y] do
      begin r := r − Vrstica[y]; y := y + 1 end;
    x := 0; Vrstica[y] := Vrstica[y] − 1;
    while r >= 0 do begin
      if Pobarvana(x, y) then r := r − 1;
      x := x + 1;
    end; {while}
    Pobarvaj(x − 1, y);
  end; {for}
end; {PobarvajZaslon1}
```

Časovna zahtevnost barvanja zaslona $X \times Y$ s tem postopkom je $O(XY(X + Y))$ (četrtina točk, torej $O(XY)$ točk, leži na spodnji desni četrtini zaslona in pri njih porabi prva zanka **while** $O(Y)$, druga pa $O(X)$ časa).

Pravzaprav bi bilo koristneje, če ne bi bili tako sistematični. Zelo preprost postopek bi bil, da bi naključno izbirali točke, dokler ne bi naleteli na kakšno nepobarvano; to bi potem pobarvali in nadaljevali na enak način. Vendar pa nas zdaj lahko zaskrbi, da bo mogoče potrebnih veliko takšnih naključnih poskusov, preden bomo naleteli na naslednjo nepobarvano točko, sploh proti koncu izvajanja postopka, ko bo že večina točk pobarvanih. Recimo, da je na celem zaslonu $n = X \cdot Y$ točk in da smo jih pobarvali že k; potem je verjetnost, da je neka naključno izbrana točka nepobarvana, enaka $p = (1 - k/n)$. Izkaže se, da bo v povprečju potrebnih $1/p = n/(n-k)$ naključnih poskusov, preden bomo zadeli kakšno nepobarvano točko.[33] Če to seštejemo po vseh točkah, imamo $s = \sum_{k=0}^{n-1} n/(n-k) = n \sum_{k=1}^{n} 1/k = nH_n$ poskusov. Vsota $H_n = \sum_{k=1}^{n} 1/k$ se imenuje „n-to harmonično število“ in velja $H_n \approx \ln n + \gamma$, pri čemer je $\gamma \approx 0{,}577$. Preden pobarvamo vse točke, potrebujemo torej $s \approx n \ln n$ operacij.

Ta postopek porabi največ časa proti koncu, ko je že večina točk pobarvanih in je zato potrebnih veliko naključnih poskusov, preden najde naslednjo nepobarvano. Recimo, da bi se ustavili takrat, ko ostane še a nepobarvanih točk; potem lahko prečešemo ves zaslon (kar zahteva še dodatnih $O(n)$ operacij), poiščemo te nepobarvane točke, jih vpišemo v neko tabelo, nato pa to tabelo premešamo in tako v naključnem vrstnem redu pobarvamo še te preostale točke. Barvanje do trenutka, ko ostane še a nepobarvanih točk, pa zdaj zahteva v povprečju $\sum_{k=0}^{n-a+1} n/(n-k) = n \sum_{k=a+1}^{n} 1/k = n(H_n - H_a) \approx$

[33] Verjetnost, da bo potrebnih točno i poskusov, je $(1-p)^{i-1}p$ (ker nam mora $i-1$ poskusov spodleteti, za kar je verjetnost vsakič $1-p$, i-ti poskus pa mora uspeti, kar se zgodi z verjetnostjo p). Pričakovano število potrebnih poskusov je zato $\sum_{i=1}^{\infty} i((1-p)^{i-1}p)$, kar se z nekaj telovadbe izkaže za enako $1/p$. Več o tem najdemo v učbenikih verjetnostnega računa pod „geometrijska porazdelitev“; ali pa v MathWorld *s. v.* “Geometric Distribution”.

$n(\ln n - \ln a) = n\ln(n/a)$ poskusov. Če na primer vzamemo $a = \sqrt{n}$, imamo $n\ln\sqrt{n} = \frac{1}{2}n\ln n$, torej pol manj poskusov kot prej, pri tem pa je poraba pomnilnika narasla le za $O(\sqrt{n})$, ker moramo pač na koncu hraniti tabelo a nepobarvanih točk. Spodnji podprogram vzame za a širino zaslona, kar je ponavadi še več kot $\sqrt{n}$, saj je zaslon ponavadi širši kot višji. Njegova časovna zahtevnost je torej $\Theta(XY \ln Y)$.

```
procedure PobarvajZaslon2;
var i, r, x, y: integer;
    Ostale: array [1..XMax] of integer;
begin
  for i := 1 to XMax * YMax − XMax do begin
    repeat
      x := Random(XMax); y := Random(YMax);
    until not Pobarvana(x, y);
    Pobarvaj(x, y);
  end; {for}
  { Ostalo je še XMax nepobarvanih točk. Poglejmo, katere so to. }
  i := 0; for y := 0 to YMax − 1 do for x := 0 to XMax − 1 do
    if not Pobarvana(x, y) then
      begin i := i + 1; Ostale[i] := y * XMax + x end;
  { Pobarvajmo te preostale točke v naključnem vrstnem redu. }
  for i := XMax downto 1 do begin
    r := Random(i) + 1;
    Pobarvaj(Ostale[r] mod XMax, Ostale[r] div XMax);
    Ostale[r] := Ostale[i];
  end; {for}
end; {PobarvajZaslon2}
```

Če smo pripravljeni žrtvovati več pomnilnika, lahko prihranimo še nekaj časa. Če je na zaslonu n točk, potrebujemo $\lceil \lg n \rceil$ bitov pomnilnika za indeks vsake točke. Za tabelo a nepobarvanih točk potrebujemo torej približno $a \lg n$ bitov pomnilnika. Če smo pripravljeni porabiti n bitov pomnilnika, torej po en bit za vsako točko (kar je sicer glede na besedilo naloge najbrž že preveč), si lahko privoščimo $a = n/\lg n$. Pri barvanju prvih $n - a$ točk imamo zato le še $n\ln(n/a) = n\ln\lg a$ klicev funkcije Pobarvana.

Lahko pa bi se pri barvanju zaslona zgledovali tudi po generatorjih naključnih števil. Preprost način za generiranje psevdonaključnih števil je formula $x_{i+1} = (ax_i+1) \bmod m$, pri čemer pa moramo primerno izbrati konstanti a in m. S to formulo bomo dobivali števila od 0 do $m-1$; če oštevilčimo točke na zaslonu z $0, \ldots, n-1$, lahko pri vsakem x_i pogledamo, če je manjši od n, in če je, pobarvamo točko x_i. Za m lahko vzamemo kakšno potenco števila 2 (lahko bi vzeli kar $2^{\lceil \lg n \rceil}$, torej najmanjšo potenco števila 2, ki je $\geq n$, vendar je bilo pri naših poskusih videti barvanje zaslona še bolj naključno, če smo

vzeli dvakrat ali štirikrat tolikšen m), za a pa je potem koristno vzeti kakšno število oblike $8k + 5$. To med drugim zagotavlja, da bomo od generatorja dobili vsa števila med 0 in $m - 1$, še preden se bo kakšno od njih ponovilo. Koristno je tudi, če a ni niti preblizu 1 niti preblizu m (sicer bi na primer majhnemu številu x_i sledilo vedno tudi majhno število x_{i+1}). Ni nujno, da vsak a, ki ustreza tem zahtevam, daje že tudi res dober generator psevdonaključnih števil, vendar pri našem problemu barvanja zaslona to niti ni tako zelo pomembno.[34] Lepo pri tem razmisleku je, da imamo le $O(m)$ računskih operacij, saj moramo spremljati le eno periodo našega psevdonaključnega generatorja; in če je $m = 2^{\lceil \lg n \rceil}$, je $m < 2n$, tako da je časovna zahtevnost našega algoritma le $O(n)$ in ne več $O(n \ln n)$ kot pri prejšnjem postopku.

```
procedure PobarvajZaslon3;
var StPobarvanih, x, y, r, a, m: integer;
begin
  { m := najmanjša potenca števila 2, ki je ≥ XMax * YMax. }
  m := 1; while m < XMax * YMax do m := m * 2;
  { Zdi se, da je pri večjem m barvanje zaslona videti še malo bolj
    naključno. Brez naslednjega stavka ima program na začetku barvanja
    nekatere stolpce rajši kot nekatere druge. }
  m := m * 4;
  { a := oblike 8k + 5, niti preblizu 1 niti preblizu m. }
  a := ((m div 2) div 8) * 8 + 5;
  { Načeloma je vseeno, s katerim r začnemo, saj bo imelo naše zaporedje }
  r := Random(XMax * YMax);                          { periodo m. }
  StPobarvanih := 0;
  while StPobarvanih < XMax * YMax do begin
    if r < XMax * YMax then begin
      Pobarvaj(r mod XMax, r div XMax);
      StPobarvanih := StPobarvanih + 1;
    end; {if}
    r := (a * r + 1) mod m;
  end; {while}
end; {PobarvajZaslon3}
```

Pri računanju `(a * r + 1) mod m` moramo biti pazljivi: vrednost `a * r` utegne biti prevelika za 32-bitna števila, tako da moramo ali uporabiti 64-bitne spremenljivke (če jih naš prevajalnik podpira) ali pa pisati svoje podprograme za računanje z velikimi celimi števili. (Na primer: pri zaslonu 1280×1024 bi gornji program dobil $m = 2^{23}$; a je blizu $m/2$ in če je r velik, blizu m, bo $a \cdot r$ lahko blizu 2^{45}.)

[34] Dober uvod v generiranje psevdonaključnih števil je Knuth, *The Art of Computer Programming*, 2. knjiga, 3. poglavje, še posebej razdelek 3.6, od koder smo povzeli tudi zgoraj navedene napotke. Glej tudi W. H. Press *et al.*, *Numerical Recipes in C*, 2. izd., § 7.1.

Še preprostejši (in tudi hitrejši) pa je naslednji postopek. Mislimo si, da bi točke na zaslonu oštevilčili po vrsticah in nato pri vsaki vrstici od leve proti desni z vrednostmi $0, \ldots, n-1$. Zahteva, naj vse točke pobarvamo v naključnem vrstnem redu, je pravzaprav enakovredna zahtevi, naj ta števila $0, \ldots, n-1$ zdaj naključno premešamo. To pa lahko enostavno naredimo tako, da nekajkrat ponovimo naslednja dva koraka: (1) vsak stolpec ciklično zamaknemo za neko naključno število točk navzdol („ciklično“ pomeni, da vsako točko, ki pade spodaj čez rob zaslona, postavimo nazaj na vrh, na začetek stolpca); (2) vsako vrstico ciklično zamaknemo za neko naključno število točk v desno. Prvi korak nam točko (x, y) premakne na $(x, (y+z_1[x]) \bmod Y)$, drugi pa (x, y) premakne na $((x + z_2[y]) \bmod X, y)$, pri čemer sta $z_1[0..X-1]$ in $z_2[0..Y-1]$ tabeli, ki nam povesta, kolikšen je zamik pri posamezem stolpcu in vrstici. Pri naših poskusih je bilo videti barvanje zaslona že čisto naključno, če smo takšno zamikanje izvedli dvakrat.

```
procedure PobarvajZaslon4;
const StZamikov = 2;
var ZamikX: array[1..StZamikov, 0..XMax − 1] of integer;
    ZamikY: array[1..StZamikov, 0..YMax − 1] of integer;
    x, y, xx, yy, i: integer;
begin
  for i := 1 to StZamikov do begin
    { V vsaki od tabel ZamikX[i] in ZamikY[i] pripravimo
      nek primeren naključen zamik. }
    for x := 0 to XMax − 1 do ZamikX[i, x] := Random(YMax);
    for y := 0 to YMax − 1 do ZamikY[i, y] := Random(XMax);
  end; {for i}
  for x := 0 to XMax − 1 do for y := 0 to YMax − 1 do begin
    xx := x; yy := y;
    for i := 1 to StZamikov do begin
      { Ciklično zamaknemo stolpec xx za ZamikX[i, xx] vrstic navzdol. }
      yy := yy + ZamikX[i, xx]; if yy >= YMax then yy := yy − YMax;
      { Ciklično zamaknemo vrstico yy za ZamikY[i, yy] stolpcev v desno. }
      xx := xx + ZamikY[i, yy]; if xx >= XMax then xx := xx − XMax;
    end; {for i}
    Pobarvaj(xx, yy);
  end; {for x, y}
end; {PobarvajZaslon4}
```

Lepo pri tem postopku je, da ne vsebuje toliko dragih računskih operacij, kot sta množenje in deljenje 64-bitnih števil pri prejšnji rešitvi.

Na oko je barvanje zaslona pri tem postopku čisto naključno, nič slabše kot pri prejšnjih rešitvah; je pa v enem pogledu ta rešitev vendarle slabša od njih. Če je na našem zaslonu n točk, obstaja $n!$ možnih vrstnih redov barvanja točk. Pri podprogramih PobarvajZaslon1 in PobarvajZaslon2 se hitro vidi, da ima vsak

od teh $n!$ vrstnih redov enako verjetnost, da bo uporabljen[35] (če je funkcija Random(r) res pošten generator naključnih števil, torej če res z enako verjetnostjo vrne vsako od števil $0, \ldots, r-1$). Pri podprogramu PobarvajZaslon4 pa mnogi vrstni redi barvanja sploh niso možni. Pri vsakem od X stolpcev imamo Y možnosti za zamik tega stolpca, pri vsaki od Y vrstic pa X možnosti za zamik te vrstice; vsega skupaj torej z eno izvedbo zamikanja vrstic in zamikanja stolpcev dosežemo $X^Y Y^X$ različnih vrstnih redov. Po k zamikanjih (zgoraj smo predlagali $k = 2$) lahko dosežemo največ $(X^Y Y^X)^k$ različnih vrstnih redov (v resnici mogoče še manj, ker lahko različna zaporedja zamikov pripeljejo do istega vrstnega reda barvanja točk). Če pišemo $X = cY$ in upoštevamo, da je gotovo $c \ll Y$, je $(X^Y Y^X)^k = (c^Y Y^Y Y^{cY})^k \ll (Y^Y Y^Y Y^{cY})^k = Y^{(c+2)kY}$. Vseh možnih vrstnih redov pa je $n!$ za $n = XY = cY^2$; znano je (Stirlingova formula), da je $n! \approx n^n e^{-n}\sqrt{2\pi n} = n^n n^{-n/\ln n}\sqrt{2\pi n}$, to pa je naprej enako $(cY^2)^{cY^2(1-1/(cY^2))}\sqrt{2\pi c}Y$, kar je (če upoštevamo $c \geq 1$, $\sqrt{2\pi c} \geq 1$ in dejstvo, da pri dovolj velikih Y gotovo velja $1/(cY^2) \leq 1/2$) naprej $\geq (Y^2)^{cY^2(1-1/2)}Y = Y^{1+cY^2}$. Preden bi imelo število dosegljivih vrstnih redov, ki je $\leq Y^{(c+2)kY}$, sploh kakšne možnosti, da bi se približalo številu vseh vrstnih redov, ki je $\geq Y^{1+cY^2}$, bi morali torej imeti dovolj velik k, da bi bilo $(c+2)kY = 1 + cY^2$, torej $k = (1+cY^2)/((c+2)Y) \approx \frac{c}{c+2}Y$. Pri tolikšnem k bi bila časovna zahtevnost našega postopka že $O(kXY) = O(XY^2)$, torej pravzaprav nič boljša kot pri PobarvajZaslon1, prostorska zahtevnost (za tabeli ZamikX in ZamikY) pa $O(k(X+Y)) = O(XY+Y^2)$, kar je sploh odločno preveč.

Opisane postopke smo primerjali pri barvanju „zaslona“ 1280×1024 točk. Da risanje po zaslonu ne bi preveč vplivalo na čas izvajanja, program v resnici ni ničesar prikazoval na zaslonu, pač pa sta podprograma Pobarvaj in Pobarvana simulirala zaslon z globalno spremenljivko — tabelo 1280×1024 bitov. Rezultate prikazuje tabela na str. 201.

REŠITVE NALOG ZA TRETJO SKUPINO

N: 184

R1994.3.1

(a) Edina zahteva, ki ji mora zadoščati program P, je, da priredi dvema različnima datotekama različni komprimirani datoteki. Z drugimi besedami, program P mora biti injektivna preslikava na množici vseh datotek. Program za dekompresijo $\bar{P}$ je pač inverz

[35] Naloga je pravzaprav zahtevala, naj zaslon barvamo enakomerno in naključno; med temi $n!$ vrstnimi redi pa je tudi nekaj takih, ki ga ne barvajo enakomerno, ampak npr. sistematično od zgoraj navzdol, torej točke, ki so v določenem trenutku že pobarvane, nikakor niso enakomerno razpršene po celem zaslonu. To je na nek način pomanjkljivost prejšnjih rešitev, vendar lahko do takšnih neenakomernih razporedov pride tudi PobarvajZaslon4; v praksi je verjetnost takšnega neenakomernega razporeda pri vseh opisanih rešitvah zanemarljivo majhna.

	Postopek	Časovna zahtevnost	Prostorska zahtevnost	Čas izvajanja	Št. klicev Pobarvana (mio.)
	PobarvajZaslon1	$O(XY(X+Y))$	$O(Y)$	69 s	419,2
	PobarvajZaslon2				
	— brez tabele Ostale	$O(XY\ln(XY))$	$O(1)$	23,1 s	19,66 (19,22⋆)
	— tabela velikosti Y	$O(XY\ln Y)$	$O(Y)$	10,8 s	10,35 (10,39⋆)
	— tabela $XY/\ln(XY)$	$O(XY\ln\ln(XY))$	$O(XY/\ln(XY))$	4,3 s	4,78 (4,78⋆)
	PobarvajZaslon3				
(†)	— $m = 2^{\lceil \lg(XY)\rceil}$	$O(XY)$	$O(1)$	6,5 s	0
(†)	— $m = 2\cdot 2^{\lceil \lg(XY)\rceil}$	$O(XY)$	$O(1)$	9,4 s	0
	— $m = 4\cdot 2^{\lceil \lg(XY)\rceil}$	$O(XY)$	$O(1)$	15,0 s	0
	PobarvajZaslon4				
(‡)	— StZamikov $= 1$	$O(XY)$	$O(X+Y)$	0,18 s	0
	— StZamikov $= 2$	$O(XY)$	$O(X+Y)$	0,26 s	0
	— StZamikov $= 3$	$O(XY)$	$O(X+Y)$	0,34 s	0
(‡)	sistematično barvanje zaslona (od zgoraj navzdol, od leve proti desni)	$O(XY)$	$O(1)$	0,10 s	0
	naivna rešitev: v tabeli pripravimo naključno permutacijo števil $0..(XY-1)$ (in torej porabimo preveč pomnilnika)	$O(XY)$	$O(XY)$	1,14 s	0

⋆ V oklepajih je pričakovano število klicev podprograma Pobarvana, torej $XY\ln(XY/a)$, če je a velikost tabele Ostale; če tabele nimamo, si mislimo $a = 1$.
† Barvanje ni videti dovolj naključno. ‡ Barvanje ni niti najmanj naključno.

Primerjava različnih rešitev naloge 1994.2.4 za $X = 1280$, $Y = 1024$.

programa P. Njegovo delovanje je nedefinirano, če ga poženemo nad datoteko, ki ni nastala s kompresijo kakšne datoteke s programom P.

(b) Naj bo k število vseh datotek, krajših od N.[36] Ker program za kompresijo P priredi različnim datotekam različne komprimirane datoteke, vsebuje množica $\mathcal{E}$ natančno k datotek. Torej je skupna dolžina m vseh datotek iz $\mathcal{E}$ najmanjša tedaj, ko množica $\mathcal{E}$ vsebuje prvih k najkrajših datotek. To pa so ravno vse datoteke iz $\mathcal{D}$. Tako smo ugotovili, da vedno velja $m \geq M$, torej je učinkovitost programa vedno manjša ali enaka 0.

(c) Najboljša učinkovitost programa ni večja od 0. Program za kompresijo P z najboljšo možno učinkovitostjo 0 je zato preprosto program, ki z datoteko ničesar ne naredi: $P(D) = D$. Pripadajoči program za dekompresijo $\bar{P}$ prav tako ničesar ne naredi: $\bar{P}(E) = E$.

(d) Seveda ima takole teoretično definirana učinkovitost programov za kompresijo v praksi majhno uporabno vrednost. V resnici nas ne zanima, kako dobro komprimira program vse možne datoteke; zanima nas učinkovitost kompresije za datoteke, s katerimi dejansko imamo opravka v praksi. Te datoteke pa so le majhen delček vseh možnih datotek. Pri datotekah iz vsakdanjega življenja se kažejo določene pravilnosti, ki jih programi za kompresijo uspešno

[36] Hitro se vidi, da je število dvojiških nizov, krajših od N, ravno $2^N - 1$ (če štejemo tudi prazni niz), njihova skupna dolžina pa je $M = 2 + (N-2)\cdot 2^N$, vendar to za nadaljevanje našega razmisleka pravzaprav ni pomembno.

izkoriščajo.

V praksi je pomembno, kakšne datoteke želimo komprimirati. Na primer, na datotekah, ki vsebujejo besedila, se bolje obnesejo eni programi za kompresijo, na datotekah, ki vsebujejo slike, pa drugi (tam se pogosto tudi odpovemo zahtevi po injektivnosti iz točke (a): ne moti nas, če se slika po kompresiji in dekompresiji malo spremeni, zato se lahko več podobnih slik skomprimira v enako zaporedje bitov). Zato je smiselno meriti učinkovitost programa za kompresijo glede na dani tip datotek (npr. besedila, programi ali bitne slike). Učinkovitost programa izmerimo statistično na zadosti velikem vzorcu datotek danega tipa. Tako dobimo dosti uporabnejšo oceno o tem, koliko prostora bomo dejansko prihranili s kompresijo.

N: 184

R1994.3.2

Ker nas zanima med ugodnimi poštninami (takimi, ki čim manj presegajo zahtevno vrednost) taka z najmanj znamkami, bomo poštnine pregledovali po naraščajočem številu znamk. Na začetku vemo, da lahko z 0 znamkami sestavimo poštnino 0. Nato v vsakem koraku poskusimo vsako od poštnin, ki se jih je dalo sestaviti s k znamkami, dopolniti s še eno znamko; tako dobimo vse poštnine, ki se jih da sestaviti s $k+1$ znamkami. Na začetku bomo tako iz poštnine 0 dobili vse možne poštnine, ki se jih da dobiti z eno samo znamko; nato bomo iz teh dobili vse poštnine, ki se jih da dobiti z dvema znamkama; itd. Takemu preiskovanju pravimo *iskanje v širino*. Da ne bi po nepotrebnem izgubljali časa z neobetavnimi poštninami, si zapomnimo, za koliko presega zahtevano vrednost najboljša doslej najdena rešitev. Če kasneje odkrijemo kakšno poštnino, ki zahtevano vrednost presega za več kot toliko, jo lahko kar takoj pozabimo, saj vemo, da to zanesljivo ne bo najboljša rešitev. Na začetku vzamemo za ta presežek kar vrednost najdražje znamke, saj gotovo obstaja kakšna poštnina, ki presega zahtevano vrednost kvečjemu za toliko (sicer bi lahko iz nje vzeli kakšno znamko in dobili še boljšo rešitev). Še en pogoj, s katerim lahko zavržemo nekatere neobetavne rešitve, je naslednji: če smo neko vrednost poštnine uspeli sestaviti že nekoč prej z manj znamkami kot zdaj, nima smisla razvijati naprej trenutne poštnine z več znamkami, saj se bo dalo vse, kar lahko naredimo iz te poštnine, narediti tudi iz tiste prejšnje z enako vrednostjo in še manj znamkami. Strategija, ki izloča delne rešitve na opisani način, se imenuje *dinamično programiranje*.

```
program Znamke;

const NajvecjiNiz = 100;

type
  NizT = array [1..NajvecjiNiz] of integer;
  { PostnineT[1] je v bistvu seznam delnih poštnin.
    PostnineT[2, i] je vrednost zadnje znamke, ki smo jo dodali
    pri tvorjenju poštnine, da smo dobili poštnino PostnineT[1, i]. }
```

```
  PostnineT = array [1..2] of NizT;
  MnozicaT = set of 1..NajvecjiNiz;

var
  VrednostiZnamk : NizT;       { vrednosti znamk, iz katerih sestavljamo poštnino }
  NizZnamk: NizT;              { niz vrednosti znamk, ki tvorijo poštnino }
  Postnine: PostnineT;         { delne poštnine in zadnje dodane znamke }
  Postnina: integer;           { poštnina, ki jo želimo pokriti }
  Presezek: integer;           { najmanjši dosedanji presežek preko poštnine }
  OstanekPostnine: integer;    { poštnina brez zadnje znamke }
  PokriteVrednosti: MnozicaT; { poštnine, ki smo jih pokrili }

procedure BeriNizVrednosti(var Niz: NizT);
var i: integer;
begin
  i := 1;
  while not Eoln do
    begin Read(Niz[i]); i := i + 1 end;
  ReadLn;
  Niz[i] := -1;
end; {BeriNizVrednosti}

procedure Inicializacija;
var i : integer;
begin
  { Izračunajmo največji možni presežek: to je kar vrednost najdražje znamke,
    kajti gotovo se da sestaviti poštnino, ki želeno vrednost presega kvečjemu
    za vrednost ene znamke. }
  Presezek := VrednostiZnamk[1];
  i := 2;
  while VrednostiZnamk[i] <> -1 do begin
    if Presezek > VrednostiZnamk[i] then
      Presezek := VrednostiZnamk[i];
    i := i + 1;
  end; {while}
  { Na začetku ni pokrita še nobena poštnina. }
  PokriteVrednosti := [];
  Postnine[1, 1] := 0; Postnine[1, 2] := -1;
  NizZnamk[1] := -1;
end; {Inicializacija}

procedure NajdiNizZnamk(TrenPostnine: PostnineT; PokriteVrednosti: MnozicaT;
          var OstanekPostnine: integer; var NizZnamk: NizT; IndeksZnamke: integer);
var NovePostnine: PostnineT;
    TrenPostnina: integer;
    i, j, k: integer;
begin
```

```
if (TrenPostnine[1, 1] <> −1) and (Presezek > 0) then begin
  i := 1; k := 1;
  while TrenPostnine[1, i] <> −1 do begin
    { Za vsako trenutno poštnino naredimo vse možne „naslednice“,
      ki jih lahko dobimo, če trenutni poštnini dodamo novo znamko. }
    j := 1;
    while VrednostiZnamk[j] <> −1 do begin
      { Izračunamo naslednico trenutne poštnine. }
      TrenPostnina := TrenPostnine[1, i] + VrednostiZnamk[j];
      if not (TrenPostnina in PokriteVrednosti) then begin
        { Trenutna poštnina še ni pokrita. }
        PokriteVrednosti := PokriteVrednosti + [TrenPostnina];
        if TrenPostnina < Postnina + Presezek then begin
          { Trenutna poštnina ne presega trenutno najboljše rešitve. }
          NovePostnine[1, k] := TrenPostnina;
          NovePostnine[2, k] := VrednostiZnamk[j];
          k := k + 1;
          if TrenPostnina >= Postnina then
            { Trenutna poštnina je izboljšala presežek. }
            Presezek := TrenPostnina − Postnina;
        end; {if}
      end; {if}
      j := j + 1;
    end; {while}
    i := i + 1;
  end; {while}
  NovePostnine[1, k] := −1;

  { Rekurzivni klic podprograma z novimi poštninami. }
  NajdiNizZnamk(NovePostnine, PokriteVrednosti, OstanekPostnine,
                NizZnamk, IndeksZnamke + 1);

  if NovePostnine[1, 1] <> −1 then begin
    { Katera znamka, dodana v tem koraku, vodi do najboljše poštnine? }
    i := 1;
    while NovePostnine[1, i] <> OstanekPostnine do i := i + 1;
    NizZnamk[IndeksZnamke] := NovePostnine[2, i];
    OstanekPostnine := OstanekPostnine − NovePostnine[2, i];
  end else begin
    { Smo v najbolj zunanjem rekurzivnem klicu, torej smo našteli že }
    NizZnamk[IndeksZnamke] := −1;     { vse znamke v tej poštnini. }
  end; {if}
end

else begin
  { Prejšnji rekurzivni klic ni ustvaril nobene nove poštnine, torej se lahko
    naše iskanje konča; rekonstruirajmo znamke v najboljši najdeni poštnini. }
  OstanekPostnine := Postnina + Presezek;
```

```
      NizZnamk[IndeksZnamke] := −1;
    end; {if}
end; {NajdiNizZnamk}

procedure IzpisiNizZnamk(NizZnamk: NizT);
var i : integer;
begin
  i := 1;
  while NizZnamk[i] <> −1 do
    begin Write(NizZnamk[i], ' '); i := i + 1 end
  WriteLn;
end; {IzpisiNizZnamk}

begin {Znamke}
  BeriNizVrednosti(VrednostiZnamk);
  ReadLn(Postnina);
  Inicializacija;
  NajdiNizZnamk(Postnine, PokriteVrednosti, OstanekPostnine, NizZnamk, 1);
  IzpisiNizZnamk(NizZnamk);
end. {Znamke}
```

V gornjem programu opravi glavnino dela podprogram NajdiNizZnamk. Ta v strukturi TrenPostnine dobi seznam poštnin, ki se jih je dalo sestaviti z IndeksZnamke − 1 znamkami (ne pa z manj). Za i-to izmed teh poštnin je v TrenPostnine[1, i] njena vrednost, v TrenPostnine[2, i] pa vrednost zadnje znamke, dodane k tej poštnini (slednje pride prav, da lahko točno rekonstruiramo skupino znamk, ki tvori neko poštnino). Konec seznama je označen s tem, da je TrenPostnine[1, i] = −1.

Podprogram NajdiNizZnamk poskuša vsaki od poštnin iz TrenPostnine na vse možne načine dodati še po eno znamko; če je nova poštnina obetavna (torej če je še nismo videli in če še ne poznamo kakšne boljše rešitve), jo doda v strukturo NovePostnine (ki ima enako zgradbo kot TrenPostnine). Nove poštnine, ki se nam tako naberejo, uporabimo kot izhodišče za nov rekurzivni klic, v katerem bomo poskušali dodati še po eno znamko. Če nismo uspeli sestaviti nobene nove poštnine, se rekurzija neha, podprogram NajdiNizZnamk pa v tem primeru shrani vrednost najboljše doslej najdene poštnine v spremenljivko OstanekPostnine.

Klicatelj lahko preveri, katera je bila nazadnje dodana znamka v poštnino z vrednostjo OstanekPostnine in to znamko doda v tabelo NizZnamk, njeno vrednost pa odšteje od OstanekPostnine, tako da bodo lahko tudi višje ležeči rekurzivni klici ugotovili, kako se nadaljuje sestava najboljše poštnine.

R1994.3.3

V spremenljivki StanjeMotorja si bomo zapomnili, kaj smo motorju nazadnje naročili, v spremenljivki Ukaz pa, kaj bi radi od njega zdaj. Vrednost MinSirina pove, kako na široko bi radi odprli

N: 185

vrata. Na primer, če hoče kdo vstopiti ali pa hoče izstopiti avtomobil, moramo naročiti odpiranje vrat do polne širine; če hoče izstopiti pešec, pa je dovolj, če naročimo odpiranje do polovične širine (če pa je že naročeno odpiranje do polne širine, npr. ker je hotel tik pred tem nekdo vstopiti, nam ni treba spreminjati ničesar). Ko se vrata pri odpiranju dovolj odprejo, moramo motor ustaviti, enako pa tudi, ko se pri zapiranju zaprejo do konca. Ko se motor ustavi in vrata niso zaprta, nastavimo budilko, ponovno pa jo nastavimo tudi, če so vrata še vedno odprta in se pojavi ovira ali pa kakšna zahteva za vstop ali izstop. Ko se čas budilke izteče, začnemo vrata zapirati. Če zaznamo oviro, vrata pa niso čisto zaprta, jih moramo začeti spet odpirati (do stare širine, ki je še vedno v MinSirina — šele ko se čisto zaprejo, postavimo MinSirina na 0).

```
program Vrata;
type MotorT = (Stoj, Odpiraj, Zapiraj);
var
  MinSirina: real;
  Ukaz, StanjeMotorja: MotorT;
  UraTece: boolean;

  function OdprtostVrat: real; external;
  function IzstopPesec: boolean; external;
  function IzstopAvto: boolean; external;
  function Vstop: boolean; external;
  function Ovira: boolean; external;
  procedure Motor(Ukaz: MotorT); external;
  procedure NastaviBudilko(s: integer); external;
  function PreostaliCas: integer; external;

begin
  MinSirina := 0; UraTece := false;
  StanjeMotorja := Stoj; Motor(Stoj);
  repeat
    Ukaz := StanjeMotorja;
    if Vstop or IzstopAvto then
      begin Ukaz := Odpiraj; MinSirina := 1 end;
    if IzstopPesec and ((MinSirina = 0) or (Ukaz = Zapiraj)) then begin
      if MinSirina = 0 then MinSirina := 0.5;
      Ukaz := Odpiraj;
    end; {if}
    if Ovira and (OdprtostVrat > 0) then Ukaz := Odpiraj;
    if UraTece and (PreostaliCas = 0) then
      begin Ukaz := Zapiraj; UraTece := false end;

    if (Ukaz = Odpiraj) and (OdprtostVrat >= MinSirina) then Ukaz := Stoj;
    if (Ukaz = Zapiraj) and (OdprtostVrat = 0) then
      begin Ukaz := Stoj; MinSirina := 0 end;
    if Ukaz <> StanjeMotorja then begin
```

```
        Motor(Ukaz); StanjeMotorja := Ukaz;
        if (Ukaz = Stoj) and (OdprtostVrat > 0) then
          begin NastaviBudilko(15); UraTece := true end;
      end; {if}

      if StanjeMotorja <> Stoj then
        begin NastaviBudilko(0); UraTece := false end;
      if (StanjeMotorja = Stoj) and (OdprtostVrat > 0) and
         (Vstop or IzstopAvto or IzstopPesec or Ovira) then
        begin NastaviBudilko(15); UraTece := true end;
  until false;
end. {Vrata}
```

R1994.3.4

N: 186

Nalogo lahko rešimo z rekurzijo. Oklepajni izraz je ali oblike $*$ ali pa $[A_1A_2\cdots A_k]$, pri čemer so $A_1,\ldots,A_k$ spet oklepajni izrazi. i-temu znaku $*$ našega oklepajskega izraza pripada na sliki navpična črtica enotske dolžine na x-koordinati i (oz. $i-1$, če hočemo, da se drevo začne na x-koordinati 0), y-koordinata pa je odvisna od globine gnezdenja (v koliko oklepajev je ovit opazovani znak $*$). Vsakemu podizrazu A_j pa pripada neko manjše poddrevo celotnega drevesa; y-koordinata, na kateri se nahaja vrh tega drevesa, je spet odvisna od globine gnezdenja, x-koordinata pa od tega, katere znake $*$ pokriva podizraz A_j. Da narišemo drevo izraza $[A_1A_2\cdots A_k]$, moramo narisati vsa poddrevesa za izraze $A_1,\ldots,A_k$ ter nato njihove vrhove povezati z vodoravno črto in na sredi nad njo še kratko navpično črto.

Podprogram RisiDrevo v spodnjem programu nariše zaporedje dreves za izraze $A_1,\ldots,A_k$ in prek parametrov LeviX in DesniX vrne x-koordinato vrha prvega ter zadnjega drevesa (torej drevesa za A_1 in drevesa za A_k). x-koordinato, na kateri je treba narisati naslednji znak $*$, lahko vzdržujemo prek globalne spremenljivke X, y-koordinato pa povečamo ob rekurzivnem klicu (parameter Globina). Rekurzivni klic izvedemo, čim zagledamo [; v okviru tega klica naj bi se izrisalo vse do pripadajočega zaklepaja], nato pa klicatelj prek vrednosti LeviPodX in DesniPodX tudi ve, po katerih koordinatah mora segati vodoravna črta nad poddrevesi.

```
program IzrazVDrevo;
uses Graph;

const MaksDolzinaNiza = 25; { največja dolžina niza }
type NizT = packed array [1..MaksDolzinaNiza] of char;
var Izraz: NizT;
    Z: integer;
    X, LeviX, DesniX : real;

procedure OdpriGraficniNacin;
```

```
var GDrv, GMode: integer;
begin
  GDrv := Detect;
  InitGraph(GDrv, GMode, 'c:\bp\bgi');
  if GraphResult <> grOk then Halt(1);
end; {OdpriGraficniNacin}

procedure ZapriGraficniNacin;
begin
  CloseGraph;
end; {ZapriGraficniNacin}

procedure Crta(X1, Y1, X2, Y2: real);
begin
  Line(Round(X1 * 50), Round(Y1 * 50),
       Round(X2 * 50), Round(Y2 * 50));
end; {Crta}

procedure RisiDrevo(Globina: integer; var LeviX, DesniX: real);
var
  LeviPodX, DesniPodX: real;
  Prvic: boolean;
begin
  Prvic := true;
  repeat
    if Izraz[Z] = '[' then begin
      Z := Z + 1; RisiDrevo(Globina + 1, LeviPodX, DesniPodX);
      Crta(LeviPodX, Globina, DesniPodX, Globina);
      DesniX := LeviPodX + (DesniPodX - LeviPodX) / 2;
      if Prvic then begin LeviX := DesniX; Prvic := false end;
      Crta(DesniX, Globina, DesniX, Globina - 1);
    end
    else if Izraz[Z] = '*' then begin
      Z := Z + 1;
      Crta(X, Globina, X, Globina - 1);
      DesniX := X;
      if Prvic then begin LeviX := DesniX; Prvic := false end;
      X := X + 1;
    end; {if}
  until (Izraz[Z] = ']') or (Izraz[Z] = ' ');
  Z := Z + 1;
end; {RisiDrevo}

procedure Demo(Niz: NizT);
begin
  ClearViewPort; Izraz := Niz; Z := 1; X := 1;
  RisiDrevo(1, LeviX, DesniX); ReadLn;
```

```
end; {Demo}

begin {IzrazVDrevo}
  OdpriGraficniNacin;
  Demo('[[***]*[**]] ');
  Demo('[[***]*[*[[**]*][**]]] ');
  Demo('[[***]*[*[[**]*]]] ');
  Demo('[[[[*]]][**[*]*]] ');
  ZapriGraficniNacin;
end. {IzrazVDrevo}
```

Za ljubitelje skladovnega računanja pa je tule še rešitev v postscriptu:[37]

```
/xKorak 5 25.4 div 72 mul def          % 5 mm
/yKorak 5 25.4 div −72 mul def         % negativna, da ne bodo drevesa rastla navzgor

% Razcepi niz na vrhu sklada: s razcepi → (prva črka s) (preostanek s)
/razcepi
{
  /s exch def                          % poberimo parameter s sklada
  s 0 1 getinterval                    % prva črka
  s 1   s length 1 sub   getinterval   % preostanek
} def

% Nariše poddrevo: globina xPrvegaListaTegaDrevesa opis drevo1 →
/drevo1                                % xPrvegaListaZaTemDrevesom ostanekOpisa xKorena
{
  10 dict begin       % Odprimo nov slovar za lokalne spremenljivke.

  % Poberimo parametre s sklada.
  razcepi   /ostanekOpisa exch def   /prviZnak exch def
  /xLista exch def   /globina exch def

  prviZnak ([) eq    % Smo pri listu ali pri notranjem vozlišču?
  {
    /prviOtrok true def
    {
      % Narišimo naslednje poddrevo.
      globina yKorak add   xLista   ostanekOpisa   drevo1

      % Zapomniti si moramo x-koordinato korena prvega in zadnjega poddrevesa.
      prviOtrok
      { /prviOtrok false def
        dup /xPrvegaPoddrevesa exch def } if
      /xZadnjegaPoddrevesa exch def

      % Tole sta spremenljivki, ki bi ju radi pravzaprav prenašali „po referenci“.
```

[37] Za več o postscriptu glej npr. Glenn C. Reid, *Thinking in PostScript*, Addison-Wesley, 1990; Adobe Systems Inc., *PostScript Language Reference Manual*, 3. izd., Addison-Wesley, 1999.

```
      /ostanekOpisa exch def
      /xLista exch def
      % Smo pregledali vsa poddrevesa?
      ostanekOpisa razcepi exch
      (]) eq { /ostanekOpisa exch def  exit } { pop } ifelse
    } loop
    % Narišimo vodoravno prečko.
    xPrvegaPoddrevesa  globina yKorak add  moveto
    xZadnjegaPoddrevesa  globina yKorak add  lineto
    % Narišimo koren.
    /xKorena  xPrvegaPoddrevesa xZadnjegaPoddrevesa add  2 div  def
    xKorena  globina yKorak add  moveto
    xKorena  globina  lineto
    % Odložimo na sklad podatke, ki jih bo uporabljal klicatelj.
    xLista ostanekOpisa xKorena
  }
  {
    % Opraviti imamo z listom — narišimo le navpično črtico.
    xLista globina moveto
    xLista globina yKorak add lineto
    % Odložimo na sklad podatke, ki jih bo uporabljal klicatelj.
    xLista xKorak add  ostanekOpisa  xLista
  }
  ifelse
  end % zaprimo slovar
} def

/drevo
{
  /opis exch def  /y exch def  /x exch def      % Poberimo parametre s sklada.
  /Courier findfont  12 scalefont  setfont      % Izpišimo opis drevesa.
  x  y 5 add  moveto
  opis show
  newpath  y x opis drevo1  stroke              % Narišimo drevo.
  pop pop pop                                   % Počistimo sklad.
} def

100 100 ([[***]*[**]]) drevo
100 200 ([[***]*[*[[**]*][**]]]) drevo
100 300 ([[***]*[*[[**]*]]]) drevo
100 400 ([[[[*]]][**[*]*]]) drevo

showpage
```

19. državno tekmovanje v znanju računalništva (1995)

NALOGE ZA PRVO SKUPINO

1995.1.1

Tekmovalna komisija je v naglici napisala spodnji program. R: 222
Kaj izpiše program? Odgovor primerno utemelji!

```
program Enigma(Output);
type
  z = packed array [1..255] of char;
var
  p, a, r: z;
  LP, LA, LR: integer;
  i, j, rks, k: integer;
begin
  p := 'ba'; LP := 2; { dolžina p }
  r := 'baba'; LR := 4; { dolžina r }
  a := 'alibaba in štirideset barbarov.'; LA := 31; { dolžina a }
  { v preostanku tabel r, p in a se nahajajo presledki }
  j := 0; rks := 1;
  while rks <> 0 do begin
    i := j + 1; k := 1;
    repeat
      if a[i] = p[k] then begin
        i := i + 1; k := k + 1;
      end else begin
        i := i - k + 2; k := 1;
      end;
    until (k > LP) or (i > LA);
    if k > LP then begin
      i := i - LP; rks := i; j := i;
      if LP < LR then begin
        for k := LA + 100 downto i + LP do a[k] := a[k - (LR - LP)];
        for k := 1 to LR do a[k + i - 1] := r[k];
      end else begin
        for k := i to LA - (LP - LR) do a[k] := a[k + (LP - LR)];
        for k := 1 to LR do a[k + i - 1] := r[k];
      end;
    end else begin
      rks := 0; j := 0;
    end; {if}
  end; {while}
  WriteLn(a:LA);
end. {Enigma}
```

Opomba: pri delu z nizi se različna narečja pascala med seboj razlikujejo v raznih podrobnostih, zato bi zgornji program pri nekaterih prevajalnikih deloval drugače, kot je bil zamišljen. Prireditev, kot je `p := 'ba'`, v standardnem pascalu pravzaprav sploh ni dovoljena, ker niza nimata enakega števila elementov (`p` ima 255 elementov, konstanto `'ba'` pa jezik šteje za tabelo z dvema elementoma). Mnogi prevajalniki pa bi takšno prireditev vendarle dovolili in bi preostanek tabele `p` zapolnili s presledki (znaki `' '`); na takšno delovanje se zanaša tudi naš zgornji program.

Klic oblike `WriteLn(a:n)` pa naj bi, če je `a` tipa **packed array** [1..m] **of** char in je `m > n`, izpisal le prvih `n` znakov tabele `a`, ne glede na to, kaj je v preostanku te tabele. Takšno obnašanje predpisuje standardni pascal in ga uporablja tudi zgornji program.[38]

R: 224

1995.1.2

Podano imamo množico delavcev, ki jih označimo s celimi števili od 1 do `n`. Vsak delavec ima lahko enega ali več neposrednih šefov. Dana je tudi funkcija `Sef(x, y: integer): boolean`, ki vrne `true`, če je `x` neposredni šef delavcu `y`. **Napiši del programa** (ali podprogram), ki za danega delavca (njegovo številko preberemo) izpiše vse njegove šefe (posredne in neposredne).

Primer: v spodnjem primeru sta Tonetova neposredna šefa Janez in Gregor, njegov posredni šef pa je Dare.

Delavec	*Šef*
Tone	Janez
Tone	Gregor
Marjan	Gregor
Marjan	Janez
Janez	Dare
Gregor	Dare
Tadej	Dare

R: 225

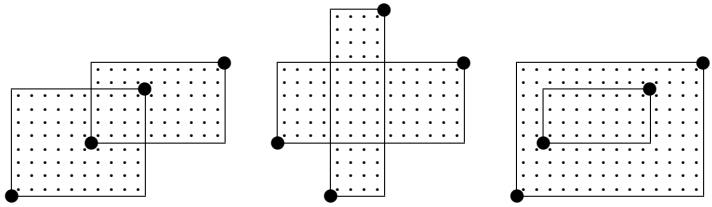
1995.1.3

Napiši program, ki bo izračunal površino unije dveh pravokotnikov v ravnini. Stranice pravokotnikov so vzporedne s koordinatnima osema.

Pravokotniki so podani s koordinatama spodnjega levega in zgornjega desnega oglišča pravokotnika. Točke so podane kot pari realnih števil (koordinati `X` in `Y`), ki jih program prebere.

Nekaj primerov: Površina unije je šrafirana s pikicami. Podani točki za posamezni pravokotnik sta označeni s črno piko.

[38] Nekateri nestandardni prevajalniki pa bi v takem primeru izpisali kar vse znake tabele `a`, ne glede na predpisano širino polja `n`. Nekateri prevajalniki z izpisovanjem znakov prenehajo že tudi pred `n`-tim znakom, če naletijo v nizu na znak `Chr(0)`.



1995.1.4

R: 226

Računalnik je sestavljen iz enega glavnega in petdesetih med seboj enakih pomožnih procesorjev. V nevihti je strela udarila v omrežje prav blizu računalnika in odpovedalo je nekaj pomožnih procesorjev. Kateri?

Napisati je potrebno **program**, s katerim bo glavni procesor ugotovil, kateri pomožni procesorji so pokvarjeni. Na voljo je funkcija Enaka(a, b: integer): boolean, ki na pomožnih procesorjih s številkama a in b izvede določene operacije in vrne true, če so dale na obeh procesorjih enak rezultat in false, če ne.

Predpostavimo lahko, da je pokvarjenih manj kot pol procesorjev in da je funkcija Enaka dovolj natančna, da ne more po naključju spregledati pokvarjenega procesorja (torej, če primerjamo pokvarjenega in pravilno delujočega, gotovo vrne false, če primerjamo dva pravilno delujoča, pa gotovo vrne true). Žal pa ni nujno, da so vsi pokvarjeni procesorji pokvarjeni na enak način; napake so lahko različne ali enake, torej funkcija Enaka pri primerjanju dveh pokvarjenih procesorjev lahko vrne true ali false.

NALOGE ZA DRUGO SKUPINO

1995.2.1

R: 227

Kaj vrne funkcija KajVrnem v odvisnosti od parametra x? Parameter t je tabela naključno izbranih števil. x je enak nekemu elementu tabele t. Odgovor utemelji!

```
const Dolz = 10000;
type Tabela = array [1..Dolz] of integer;

function KajVrnem(var t: Tabela; x: integer): integer;
var
  a, b, c: integer;
begin
  a := 1; b := Dolz;
  while a < b do begin
    while t[a] < x do a := a + 1;
    while t[b] > x do b := b - 1;
    if a < b then begin
      c := t[a]; t[a] := t[b]; t[b] := c;
```

```
    end; {if}
  end; {while}
  KajVrnem := a;
end; {KajVrnem}
```

R: 228

1995.2.2

Računalniki so iz dneva v dan hitrejši in imajo več pomnilnika. Pri resnem delu pa je ozko grlo še vedno branje in pisanje na trdi disk, ki je približno 100000-krat počasnejše kot branje in pisanje v pomnilnik. Zato skoraj vedno žrtvujemo del pomnilnika za t.i. priročni pomnilnik (angl. cache), ki ima del podatkov iz diska vedno shranjen v pomnilniku in če uporabnik zahteva kakšnega od njih, ni potrebno dosegati podatkov na disku. To seveda zelo pohitri delo na računalniku.

Kako vse skupaj deluje? Disk je razdeljen na manjše enote, ki jih imenujmo bloki in tipično vsebujejo 512 (ali več) znakov. Do operacijskega sistema prihajajo zahtevki za branje in pisanje blokov na disk. Ker se v praksi pokaže, da v nekem obdobju uporabnik pogosto prebere isti blok večkrat in čez isti blok večkrat zapiše različno vsebino, si pomagamo s priročnim pomnilnikom, kamor shranjujemo uporabnikove zahtevke in podatke. Če hoče uporabnik zapisati blok podatkov na disk, tega ne zapišemo zares, ampak ga shranimo v priročni pomnilnik. Tam je, dokler nam ne zmanjka prostora v priročnem pomnilniku in se izkaže, da je bil ta blok največ časa neuporabljen — takrat ga dejansko zapišemo na disk. Če v času, ko se blok nahaja v pomnilniku, pride še en zahtevek po tem, da ga prepišemo, ga preprosto prepišemo le v pomnilniku. Če pride zahtevek po branju bloka, ki se ne nahaja v priročnem pomnilniku, ga moramo najprej prebrati z diska, nato pa ga vrnemo uporabniku in ga še shranimo v priročni pomnilnik. V primeru, ko uporabnik zahteva blok, ki je že v priročnem pomnilniku, mu vrnemo kar tega.

V rešitvi **opiši postopek**, ki ga morata izvajati podprograma za pisanje in branje z diska,

```
procedure ZapisiBlok(StBloka: integer; P: Podatki);
procedure PreberiBlok(StBloka: integer; var P: Podatki);
```

ki pa imata za pomoč priročni pomnilnik, v katerega lahko shraniš do MaksPPBlokov blokov. Opiši tudi podatkovne strukture, ki omogočajo čim hitrejše delovanje obeh podprogramov. Ko je podatek zares potrebno fizično zapisati oz. prebrati z diska, uporabi podprograma:

```
procedure ZapisiBlokNaDisk(StBloka: integer; P: Podatki); external;
procedure PreberiBlokZDiska(StBloka: integer; var P: Podatki); external;
```

Predpostavi, da ima vsak blok na disku svojo številko (celo število med 1 in MaksBlokov) in tabelo znakov, ki predstavljajo podatke:

```
type Znak = 0..255;
type Podatki = array [1..512] of Znak;
```

1995.2.3

R: 230

Prek računalniškega omrežja želimo razširjati zaporedje znakov, npr. besedila dokumentov, ki stalno, a neenakomerno sproti nastajajo. Naš računalnik sprejema znake po eni vhodni zvezi, posredovati pa jih mora sproti, ne da bi najprej čakal na konec dokumenta, naprej drugim računalnikom, ki jih je nekaj ducatov (neka konstanta), do vsakega od njih pa vodi ena izhodna zveza.

Na voljo imamo dva podprograma:

function Beri(**var** ch: char): boolean;

- če je na vhodni liniji na voljo kakšen nov znak, ga prebere in vrne kot parameter, vrednost funkcije je true;
- če na vhodu ni nobenega znaka, je vrednost funkcije false, ch pa ni definiran.

Funkcija se vedno izvede takoj (ne čaka na znake). Če funkcije nekaj časa ne pokličemo, se tok prihajajočih znakov začasno ustavi (znaki se ne bodo izgubili).

function Pisi(iz: integer; ch: char): boolean;

Na izhodno linijo iz (številka med 1 in številom izhodnih linij) poskusi poslati znak ch. Če je bilo pošiljanje uspešno, vrne true, sicer pa false. Funkcija se vedno izvede takoj (npr. ne čaka, da se linija sprosti).

Zveze niso enako hitre, vendar tudi najpočasnejša večino časa dohaja dotok novih besedil — če pa ga kdaj ne dohaja, se znaki ne smejo izgubljati, prenos se lahko le upočasni. Da prenosi po hitrejših linijah ne bi trpeli na račun počasnejših (razen izjemoma, npr. pri prenašanju zelo dolgih besedil), uporabimo v programu večji kos pomnilnika (vrsto/buffer) za začasno hranjenje še neodposlanih znakov. Na razpolago je le toliko pomnilnika, da gre vanj večina dokumentov v celoti, ne pa najdaljši.

Napiši program, ki bo skrbel za razširjanje besedil.

1995.2.4

R: 231

V mreži računalnikov se nahaja sto vozlišč (računalnikov), ki si med seboj lahko pošiljajo sporočila. Posamezno vozlišče je lahko povezano z največ petimi sosednjimi vozlišči. Vsak računalnik ima svoj enoten mrežni naslov, ki je neko pozitivno celo število, vendar pa ne ve, s katerimi sosednjimi računalniki in po katerih povezavah je povezan.

Da bi prenos sporočil potekal kar najhitreje, mora vsak računalnik poznati najkrajše poti do ostalih računalnikov v mreži. Za to naj poskrbi **program**, ki se bo hkrati pognal in izvajal na vseh računalnikih. Njegova naloga je izvedeti,

v katero izmed petih smeri naj posamezen računalnik pošlje sporočilo, če ga želi poslati nekemu drugemu računalniku. Program naj si zgradi tabelo najkrajših smeri za vseh sto računalnikov v mreži.

Pri pisanju programa si pomagajte z naslednjimi podprogrami:

function MojNaslov: integer;
Vrne naslov računalnika, ki je število od 1 do 100.

function Poslji(VSmer: integer; Sporocilo): boolean;
Pošlje sporočilo v želeno smer. Sporočilo definirajte sami. Če računalnik ne more poslati sporočila, ker v dani smeri ni povezave, funkcija vrne false, sicer pa true. Upoštevajte, da se sporočila nikoli ne izgubljajo.

function Sprejmi(**var** IzSmeri: integer; **var** Sporočilo): boolean;
V primeru, da je prispelo novo sporočilo, ga zapiše v parametra IzSmeri in Sporočilo ter vrne true. Če sporočila ni, vrne false.

Pri tem upoštevajte, da smeri pomenijo številke povezav med računalniki; to so števila od 1 do 5.

NALOGE ZA TRETJO SKUPINO

R: 232

1995.3.1

Preveč samozavesten programer je napisal naslednji program. **Kaj ta program izpiše?** Odgovor primerno utemelji! Kdaj izbrani algoritem ne bi deloval pravilno?

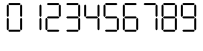
```
program Enigma(Output);
type z = packed array [1..255] of char;
var
  p, a, r: z;
  j: integer;

function Length(s: z):integer;
var i: integer;
begin
  i := 255; while (i > 1) and (s[i] = ' ') do i := i - 1;
  if (i = 1) and (s[1] = ' ') then Length:= 0 else Length:= i;
end;

procedure Replace(LP, LR, LA: integer; var a, r: z);
var k: integer;
begin
  if LP < LR then begin
    for k := LA + 100 downto j + LP do a[k] := a[k - (LR - LP)];
    for k := 1 to LR do a[k + j - 1] := r[k];
  end else begin
```

```
    for k := j to LA − (LP − LR) do a[k] := a[k + (LP − LR)];
    for k := 1 to LR do a[k + j − 1] := r[k];
  end;
end;

function RKSearch(var a, p, r: z):integer;
const
  q = 33554393; { praštevilo }
  d = 32;
var
  h1, h2, dM, i, LP, LA, LR: integer;
begin
  LP := Length(p); LA := Length(a); LR := Length(r);
  dM := 1; for i := 1 to LP − 1 do dM := (d * dM) mod q;
  h1 := 0; for i := 1 to LP do h1 := (h1 * d + Ord(p[i])) mod q;
  h2 := 0; for i := 1 to LP do h2 := (h2 * d + Ord(a[i + j])) mod q;
  i := j + 1;
  while (h1 <> h2) and (i <= LA − LP) do begin
    h2 := (h2 + d * q − Ord(a[i]) * dM) mod q;
    h2 := (h2 * d + Ord(a[i + LP])) mod q;
    i := i + 1;
  end; {while}
  if h1 = h2 then begin
    RKSearch := i; j := i; Replace(LP, LR, LA, a, r);
  end else begin
    RKSearch := 0; j := 0;
  end; {if}
end;

begin
  r := 'baba'; p := 'ba'; a := 'alibaba in štirideset barbarov.';
  { v preostanku tabel r, p in a se nahajajo presledki }
  j := 0;
  while RKSearch(a, p, r) <> 0 do begin
    WriteLn(a:Length(a)); WriteLn('^':j);
    j := j + Length(r) − Length(p);
  end;
end.
```

Opomba: pri delu z nizi se različna narečja pascala med seboj razlikujejo v nekaterih podrobnostih; glej opombo pri prvi nalogi za prvo skupino, str. 212.

1995.3.2

Na zaslonu je niz N sedemsegmentnih (digitalnih) števk. R: 234
Predstavitev števk s segmenti je naslednja:



Za prižgane segmente vemo, da so pravilni, medtem ko za ugasnjene segmente ne vemo, ali so pravilni ali pokvarjeni. Poleg tega vemo, da je vsota vseh števk deljiva z 10.

Opiši postopek, ki najde tak niz vrednosti števk, da ustreza omejitvam in ima najmanj popravljenih segmentov.

Primer:  ima možni rešitvi 3-7 in 8-2. Rešitev je 3-7, ker ima samo en popravljen segment, medtem ko ima 8-2 pet popravljenih segmentov.

R: 243

1995.3.3

Podano imamo množico delavcev, ki jih označimo s celimi števili od 1 do `n`. Vsak delavec ima lahko enega ali več neposrednih šefov. Dana je tudi funkcija **function** JeSef(x, y: integer): boolean, ki vrne `true`, če je `x` šef delavcu `y` (neposredni ali posredni). **Napiši algoritem**, ki za dano množico delavcev izračuna množico vseh njihovih najbližjih skupnih šefov.

Najbližji skupni šef množice delavcev `S` je delavec:

1. ki je šef vsakemu delavcu iz množice `S` in
2. ne obstaja njemu podrejeni delavec, ki bi bil prav tako šef vsem delavcem iz množice `S`.

Primer: V primeru, da je relacija **function** JeSef(x, y: integer): boolean definirana z naslednjo tabelo, sta Gregor in Janez najbližja skupna šefa Tonetu in Marjanu.

Delavec	*Šef*
Tone	Janez
Tone	Gregor
Marjan	Gregor
Marjan	Janez
Janez	Dare
Gregor	Dare
Tadej	Dare

R: 245

1995.3.4

V mreži računalnikov se nahaja poljubno število vozlišč (računalnikov), ki si med seboj lahko pošiljajo sporočila. Posamezno vozlišče je lahko povezano z največ enajstimi sosednjimi vozlišči. Računalniki ob vključitvi nimajo informacije o tem, po katerih povezavah in s katerimi sosednjimi računalniki so povezani, niti o tem, kje se nahajajo. Zato želimo vse računalnike, ki so priključeni v mrežo, označiti z enotnimi identifikacijskimi številkami, ki nam bodo kasneje služile kot mrežni naslovi. To naj naredi program, ki se zažene vzporedno na vseh računalnikih ob vzpostavitvi mreže.

Napišite omenjeni **program** in pri tem uporabite naslednji funkciji in podprogram:

function Poslji(VSmer: integer; Sporocilo): boolean; **external**;
Pošlje sporočilo v želeno smer. Sporočilo definirajte sami. Če računalnik ne more poslati sporočila, ker v dani smeri ni povezave, funkcija vrne false, sicer pa true. Upoštevajte, da se sporočila nikoli ne izgubljajo.

function Sprejmi(**var** IzSmeri: integer; **var** Sporocilo): boolean; **external**;
Če je prispelo novo sporočilo, ga zapiše v spremenljivki IzSmeri in Sporocilo ter vrne true. Če pa sporočila ni bilo, vrne false.

procedure NastaviSvojNaslov(Naslov: integer); **external**;
Ko računalnik dožene, kakšen naslov mu pripada, s tem podprogramom nastavi svoj mrežni naslov.

Pri tem upoštevajte, da smeri pomenijo številke povezav med računalniki, ki so lahko od 1 do 11. Izjema je le prvo sporočilo, ki do naključno izbranega računalnika pride od uporabnika (iz smeri 0) in pomeni ukaz za začetek označevanja. Program napišite tako, da, ko je označevanje na vseh računalnikih končano, eden izmed računalnikov to sporoči uporabniku (pošlje naj kakršnokoli sporočilo v smer 0).

PRVO ZAKLJUČNO TEKMOVANJE V ZNANJU RAČUNALNIŠTVA

1995.Z

Dani sta datoteki `vzorci.txt` in `besede.txt`. V vsaki vrstici obeh datotek se nahaja po ena beseda v velikih črkah. Dolžina besede je od 1 do 12 znakov. R: 247

Napiši program, ki za vsako besedo iz datoteke `vzorci.txt` poišče vse bližnje besede iz datoteke `besede.txt` in jih zapiše v datoteko `rezultat.txt`. Beseda B (iz `besede.txt`) sodi med bližnje besede besedi V (iz `vzorci.txt`), če B vsebuje vse znake iz besede V, vsebuje pa lahko še 2 dodatna znaka.

Primera: Besedi `UNIVERZA` (iz `vzorci.txt`) so bližnje besede `RAZVIDENJU`, `REVANŠIZMU`, `UNIVERZA`, `UNIVERZAH`, `UNIVERZUMA` (iz `besede.txt`). Besedi `MOJSTRANA` je bližnja beseda `ASTRONOMIJA`.

Naloga je napisati program, ki bo čim hitreje iskal bližnje besede. Čas reševanja je 3 ure. V tem času si lahko na področju (direktoriju), ki vam je določen za delo, poleg programov zgradite tudi indekse in druge pomožne datoteke, ki vam pomagajo pri iskanju bližnjih besed. Pri tem lahko vse datoteke na področju zapolnijo največ 5 MB prostora. Po končanem reševanju nalog bomo vsebino področij vseh tekmovalcev prenesli na skupen računalnik, kjer bomo izmerili hitrost izvajanja programov na spremenjeni datoteki `vzorci.txt`, ki bo vsebovala nekaj deset besed v enakem formatu kot testna datoteka. Zmagal bo tekmovalec, katerega program bo najhitreje poiskal vse bližnje besede besedam iz nove datoteke `vzorci.txt`.

Čas bomo merili s programom `mericas.pas`, ki ga poženete iz DOSa s parametrom, ki je ime vašega programa skupaj s podaljškom `exe`. Primer:

Če je datoteki s programom, ki ste ga napisali, ime `isci.pas`, prevedeni pa `isci.exe`, izmerite čas izvajanja iz DOSa z ukazom:

```
c:\>mericas isci.exe
```

Naša inačica programa bo poleg merjenja časa še preverjala, ali vaš program izpiše vse možne besede.

V pomoč vam je lahko naivna izvedba programa za iskanje bližnjih besed, ki se nahaja v datoteki `naivna.pas`. Program, ki ga morate napisati, se mora obnašati enako kot spodnji primer, le vrstni red izpisanih besed je lahko drugačen. Več možnosti za zmago pa boste seveda imeli, če bo vaš program deloval hitreje.

```
program NaivnaResitev;
var
  fVzorci, fBesede, fRezultat: text;
  CelVzorec, Vzorec, CelaBeseda, Beseda: string;
  StZnaka, NajdenZnak: integer;
begin
  Assign(fRezultat, 'rezultat.txt'); Rewrite(fRezultat);
  Assign(fVzorci, 'vzorci.txt'); Reset(fVzorci);
  while not Eof(fVzorci) do begin
    ReadLn(fVzorci, CelVzorec );
    Assign(fBesede, 'besede.txt'); Reset(fBesede);
    while not Eof(fBesede) do begin
      ReadLn(fBesede, CelaBeseda);
      Vzorec := CelVzorec; Beseda := CelaBeseda;
      for StZnaka := 1 to Length(CelVzorec) do begin
        NajdenZnak := Pos(CelVzorec[StZnaka], Beseda);
        if NajdenZnak <> 0 then begin
          Delete(Beseda, NajdenZnak, 1);
          Delete(Vzorec, Pos(CelVzorec[StZnaka], Vzorec), 1);
        end; {if}
      end; {for}
      if (Length(Vzorec) <= 0) and (Length(Beseda) <= 2) then
        WriteLn(fRezultat, CelaBeseda);
    end; {while};
    Close(fBesede);
  end; {while};
  Close(fVzorci); Close(fRezultat);
end. {NaivnaResitev}
```

[Pri nalogah, ki se jih rešuje na računalnikih, vedno obstaja nerodnost, da s prihodom hitrejših procesorjev in večjih količin pomnilnika včasih odpadejo omejitve, zaradi katerih je bila naloga prej težka; marsikatera naloga postane lažja, včasih celo zelo lahka. Mogoče najpomembnejši tovrstni omejitvi na

finalnem tekmovanju leta 1995 sta bili majhna količina pomnilnika (programi so tekli v realnem načinu, kar pomeni, da so imeli za svoje delo na voljo le nekaj sto KB pomnilnika, vsekakor pa ne več kot 640 KB) in pa dejstvo, da niti disk niti operacijski sistem najbrž nista imela kakšne resne količine diskovnega predpomnilnika.

Datoteko `besede.txt` in druge datoteke, povezane s tem tekmovanjem, naj bi se dalo dobiti na `http://rtk.ijs.si/`. Če bralec te datoteke nima pri roki, mu bo pri načrtovanju rešitve mogoče prišel prav vsaj podatek, da je bilo v datoteki `besede.txt` 112539 besed s skupno 883193 črkami (število besed po dolžini od eno- do dvanajstčrkovnih: 29, 387, 1612, 4659, 10202, 14896, 18296, 18848, 16286, 12971, 8844, 5509). Vse besede so sestavljene iz velikih črk `A`, ..., `Z` (tudi `Q`, `W`, `X` in `Y`), `Č`, `Š` in `Ž`.

Zaključno tekmovanje je potekalo dan po glavnem delu tekmovanja iz znanja, torej v nedeljo, 14. maja 1995. Udeležilo se ga je 18 tekmovalcev (najboljši iz 2. in 3. skupine ter nekaj mladih raziskovalcev). Ker na tem zaključnem tekmovanju ni vse teklo čisto tako gladko, kot bi si bilo mogoče želeti, so za osem najboljših udeležencev tega tekmovanja organizirali še drugo tekmovanje, ki je potekalo v soboto, 27. maja 1995; naloga je bila enaka kot pri prvem, le da vzorci niso bili podani v datoteki, ampak je moral program rešiti problem za en sam vzorec, ki ga je dobil v ukazni vrstici (argv v C/C++, ParamStr v Turbo Pascalu). Pri tem drugem tekmovanju ni šlo za klavzurno reševanje naloge, ampak je nalogo vsak reševal doma v dveh tednih med obema tekmovanjema, sámo drugo tekmovanje pa je bilo namenjeno zgolj preizkušanju in merjenju hitrosti nastalih programov. V nadaljevanju sledi opis naloge za drugo tekmovanje. — *Op. ur.*]

Dana je datoteka besed `besede.txt`, ki v vsaki vrstici vsebuje eno besedo dolžine od 1 do 12 znakov. **Napiši program**, ki za posamezno besedo (podano kot podatek iz ukazne vrstice) poišče vse bližnje besede iz datoteke `besede.txt` in jih zapiše v datoteko `rezultat.txt`. Beseda B (iz `besede.txt`) sodi med bližnje besede besedi V (podani iz ukazne vrstice), če B vsebuje vse znake iz besede V, vsebuje pa lahko še 2 dodatna znaka.

Primera: Besedi `UNIVERZA` so bližnje besede `RAZVIDENJU`, `REVANŠIZMU`, `UNIVERZA`, `UNIVERZAH`, `UNIVERZUMA` (iz `besede.txt`). Besedi `MOJSTRANA` je bližnja beseda `ASTRONOMIJA`.

Naloga je napisati program (z imenom `najdi`), ki bo čim hitreje iskal bližnje besede. Dovoljeno je, da si na delovnem področju (direktoriju) poleg programov zgradite tudi indekse in druge pomožne datoteke, ki vam pomagajo pri iskanju bližnjih besed. Pri tem lahko vse datoteke na področju zapolnijo največ 5 MB prostora. Na tekmovanju bomo vsebino področij vseh tekmovalcev prenesli na skupen računalnik, kjer bomo izmerili hitrost izvajanja programov na nekaj deset vzorčnih besedah. Zmagal bo tekmovalec, katerega program bo najhitreje poiskal vse bližnje besede podanim vzorčnim besedam. Program mora biti napisan v Turbo Pascalu 7.0 ali Borland C 3.1 za DOS, z uporabo osnovnega pomnilnika (do 640 KB).

Poleg programa morajo tekmovalci napisati dokumentacijo v angleškem jeziku dolžine med 1000 in 1500 besed. Dokumentacija mora vsebovati opis algoritma za iskanje bližnjih besed in opis programske izvedbe tega algoritma.

Na samem tekmovanju bodo tekmovalci zagovarjali svoj program v pet- do desetminutni predstavitvi v angleškem jeziku. Dovoljena je uporaba grafoskopa. Prosojnice smejo vsebovati le slike in diagrame, ne smejo pa vsebovati besedila, ki bi ga tekmovalec bral.

Čas bomo merili s programom `mericas.pas`, ki ga poženete iz DOSa s parametrom, ki je ime vašega programa skupaj s podaljškom `exe`. Primer: Če je datoteki s programom, ki ste ga napisali, ime `najdi.pas`, prevedeni pa `najdi.exe`, izmerite čas izvajanja iz DOSa z ukazom:

```
c:\>mericas najdi.exe
```

Program `mericas.pas` jemlje vzorčne besede eno po eno iz datoteke `vzorci.txt` in jih daje kot parameter klicu vašega programa. Naša inačica programa bo poleg merjenja časa še preverjala, ali vaš program izpiše vse možne besede.

V pomoč vam je lahko naivna izvedba programa za iskanje bližnjih besed, ki se nahaja v datoteki `naivna.pas`. Program, ki ga morate napisati, se mora obnašati enako kot spodnji primer, le vrstni red izpisanih besed je lahko drugačen. Več možnosti za zmago boste seveda imeli, če bo vaš program deloval hitreje.

```
program NaivnaResitev;
var
   fBesede, fRezultat: text;
   CelVzorec, Vzorec, CelaBeseda, Beseda: string;
   StZnaka, NajdenZnak: integer;
begin
   Assign(fRezultat, 'rezultat.txt'); Rewrite(fRezultat);
   CelVzorec := ParamStr(1);
   Assign(fBesede, 'besede.txt'); Reset(fBesede);
   while not Eof(fBesede) do begin
      ReadLn(fBesede, CelaBeseda);
      Vzorec := CelVzorec; Beseda := CelaBeseda;
      for StZnaka := 1 to Length(CelVzorec) do begin
         NajdenZnak := Pos(CelVzorec[StZnaka], Beseda);
         if NajdenZnak <> 0 then begin
            Delete(Beseda, NajdenZnak, 1);
            Delete(Vzorec, Pos(CelVzorec[StZnaka], Vzorec), 1);
         end; {if}
      end; {for}
      if (Length(Vzorec) <= 0) and (Length(Beseda) <= 2) then begin
         WriteLn(fRezultat, CelaBeseda); WriteLn(CelaBeseda);
      end; {if}
   end; {while};
   Close(fBesede);
   Close(fRezultat);
end. {NaivnaResitev}
```

REŠITVE NALOG ZA PRVO SKUPINO

N: 211

R1995.1.1

Program poskuša zamenjati pojavitve niza `p` v nizu `a` z nizom `r`. V vsaki iteraciji zunanje zanke (zanka **while**)

naj bi išče v nizu a od vključno indeksa j + 1 naprej (zato pri inicializaciji postavimo j na 0, da bomo iskali od začetka niza a).

Zanka **repeat** poišče naslednjo pojavitev podniza p v nizu a (torej prvo pojavitev od vključno indeksa j + 1 naprej). Dokler vidi ujemanje med nizoma a in p, se premika naprej po obeh (povečuje i in k), dokler ne pride do konca niza p (to je pri pogoju k > LP). Če pa se trenutna znaka a[i] in p[k] ne ujemata, poizkusi z naslednjim možnim položajem niza p in začne tam primerjati spet od začetka. Če smo nazadnje primerjali a[i] in p[k], pomeni, da smo začeli pri a[i − k + 1] in p[1]; ko niz p v mislih premaknemo za eno mesto naprej vzdolž niza a, bomo morali torej začeti primerjati pri a[i − k + 2] in p[1].

Notranja zanka **repeat** se konča, če pride k do konca niza p (kar pomeni, da smo našli pojavitev niza p v nizu a) ali pa i do konca niza a (ko je i > LA, kar pomeni, da smo v a-ju odkrili že vse pojavitve p-ja in zato nehamo: v ta namen postavimo rks na 0 (vse dotlej je imel neničelno vrednost), da se bo zunanja zanka prekinila).

Če smo našli pojavitev p-ja v a-ju (k > LP), jo je treba zamenjati z r-jem. Ko indeks i zmanjšamo za LP, kaže spet na začetek te pojavitve p-ja v a-ju. Zdaj je mogoče, da r ni tako dolg kot p in bo treba zato tisti preostanek a-ja, ki ne pripada trenutni pojavitvi p-ja, prestaviti.

Če je r daljši od p-ja, bo treba vsako celico a[k] prestaviti v a[k + (LR − LP)]; da pa pri tem ne bi povozili prejšnje vsebine te celice, bomo počeli to od konca niza proti začetku in k pri tem zmanjševali. Tu program napačno predpostavi, da je niz a dolg LA in za „boljše“ delovanje se izvaja znaka od indeksa LA + 100 navzdol (ob začetku izvajanja programa je dolŽina niza a res LA, toda med izvajanjem programa se lahko dolžina a-ja spreminja in program ne spreminja vrednosti spremenljivke LA). Varno (toda malo bolj potratno) napisana zanka bi prestavila vse znake od konca tabele, to je od celice 255 nazaj.

Če je r krajši od p-ja, ravnamo podobno, le da se tu vsebina a-ja seli na nižje indekse, zato moramo iti po naraščajočih k namesto po padajočih. Nato še prepišemo vsebino r-ja v tabelo a na indekse od i naprej.

Ob zamenjavi trenutne pojavitve p-ja z r-jem postavimo j na i (rks pa tudi, ampak glede tega je čisto vseeno, samo da bo rks neničeln, kar pa zagotovo je, ker je i zagotovo neničeln). Naslednja ponovitev zunanje zanke začne preverjati pojavitve p-ja na indeksih od i + 1 naprej, torej eno mesto za trenutno pojavitvijo.

Program vsebuje napako: ker se iskani niz p pojavlja kot podniz v zamenjavnem nizu r in ker iščemo naslednjo pojavitev p-ja na naslednjem mestu za zadnjo najdeno pojavitvijo p-ja (to je pri i + 1) namesto na koncu zamenjanega (vrinjenega) r-ja), bi zdaj znotraj tega vrinjenega r-ja našel novo pojavitev p-ja, zamenjal še to in tako naprej. Niz a bi se pri tem napihoval v nedogled, če ga ne bi rešila še druga napaka v programu: ko zamenja neko pojavitev p-ja z r-jem, se dolžina niza a poveča za LR − LP, program pa spremenljivke LA ne

spreminja. Zato se ne meni za tiste pojavitve p-ja v a-ju, ki se ne pojavljajo v celoti znotraj prvih LA znakov. Ob koncu izvajanja programa vsebuje a niz

„alibababababababababababababababa in štirideset barbarov.“

(v preostanku tabele a pa so presledki). Ker program (neupravičeno) zaupa vrednosti LA in izpiše le prvih LA znakov niza a, bomo dobili izpis

„alibababababababababababababababa“.

N: 212

R1995.1.2

V spodnjem programu je rešitev naloge pravzaprav le podprogram Nadrejeni, ostalo pa so pomožne deklaracije oz. stvari, za katere je bilo v nalogi rečeno, da lahko predpostavimo, da že obstajajo (npr. funkcija Sef).

Podprogram IzpisiNadrejene hrani v tabeli Nadrejeni podatke o tem, za katere delavce že ve, da so x-ovi šefi. Na začetku označi, da je x šef samemu sebi, nato pa za vsakega že znanega šefa doda v tabelo vse njegove šefe. Če pridemo enkrat skozi celo tabelo Nadrejeni, ne da bi odkrili še kakšnega novega šefa, smo našli vse in lahko končamo (spremenljivka Konec).[39]

Spodnja rešitev je do neke mere neučinkovita, ker vedno znova prečesava celotno tabelo Nadrejeni, čeprav bi bilo v resnici dovolj že, če bi preverila le tiste šefe, ki jih je dodala v prejšnjem prehodu čez to tabelo — lahko bi torej na novo odkrite šefe dodajala v neko pomožno tabelo, ki bi jo potem uporabljala kot vrsto za šefe, ki smo jih že odkrili, a si jih moramo še podrobneje ogledati.

```
const n = ....; { Število oseb. }
type Oseba = 1..n; { Številka osebe. }
function Sef(s, d: Oseba): boolean; external;

procedure IzpisiNadrejene(x: Oseba); { Izpiše vse ljudi, nadrejene delavcu x. }
var Konec: boolean;
    i, j: Oseba;
    Nadrejeni: array [Oseba] of boolean;
begin
  { Inicializacija tabele Nadrejeni. }
  for i := 1 to n do Nadrejeni[i] := false;
  Nadrejeni[x] := true;
  { Poišči vse nadrejene šefe. }
  repeat
    Konec := true;
```

[39] Podprogram IzpisiNadrejene na koncu postavi Nadrejeni[x] na false, ker si pač mislimo, da x ne more biti samemu sebi šef. Vendar bi bilo konec koncev mogoče tudi to — klic Sef(x, x) bi lahko vrnil true ali pa bi šlo vodstvo v ciklih (npr. x in nek y bi bila lahko šefa drug drugemu). V tem primeru bi smeli na koncu postaviti Nadrejeni[x] na false le, če ni x neposredni šef nobenemu od odkritih šefov (niti samemu sebi).

```
      for i := 1 to n do if Nadrejeni[i] then
        for j := 1 to n do
          if Sef(j, i) and not Nadrejeni[j] then
            begin Konec := false; Nadrejeni[j] := true end
  until Konec;
  Nadrejeni[x] := false; { x v resnici ni sam svoj šef. }
  { Izpišimo nadrejene. }
  Write('Nadrejeni(', x, '):');
  for i := 1 to n do if Nadrejeni[i] then Write(' ', i);
  WriteLn;
end; {IzpisiNadrejene}
```

R1995.1.3

Lažje kot unijo dveh pravokotnikov (katerih stranice so vzporedne koordinatnima osema) je določiti njun presek: presek pravokotnikov $\langle(x_a, y_a), (x_b, y_b)\rangle$ in $\langle(x'_a, y'_a), (x'_b, y'_b)\rangle$ je pravokotnik

N: 212

$$\langle(\max\{x_a, x'_a\}, \max\{y_a, y'_a\}), (\min\{x_b, x'_b\}, \min\{y_b, y'_b\})\rangle.$$

Pri tem prvi par koordinat predstavlja spodnje levo, drugi pa zgornje desno oglišče pravokotnika. Paziti moramo še na možnost, da je presek prazen; to se zgodi, če je $\max\{x_a, x'_a\} \geq \min\{x_b, x'_b\}$ ali $\max\{y_a, y'_a\} \geq \min\{y_b, y'_b\}$.

Ploščino unije pravokotnikov lahko zdaj dobimo tako, da seštejemo ploščini obeh pravokotnikov in od vsote odštejemo ploščino preseka (saj smo ga v vsoti šteli dvakrat).

```
program UnijaPravokotnikov(Input, Output);

type Tocka = record x, y: real end;
     Pravokotnik = record a, b: Tocka end;

function Max(a, b: real): real;
  begin if a > b then Max := a else Max := b end;

function Min(a, b: real): real;
  begin if a < b then Min := a else Min := b end;

procedure Zamenjaj(var a, b: real);
var tmp: real;
begin
  tmp := a; a := b; b := tmp;
end; {Zamenjaj}

procedure Uredi(var a, b: Tocka);
begin
  if a.x > b.x then Zamenjaj(a.x, b.x);
  if a.y > b.y then Zamenjaj(a.y, b.y);
end; {Uredi}
```

```
function Presek(var pa, pb: Pravokotnik): real;
var PresekP: Pravokotnik;
begin
  Uredi(pa.a, pa.b);
  Uredi(pb.a, pb.b);
  PresekP.a.x := Max(pa.a.x, pb.a.x);
  PresekP.a.y := Max(pa.a.y, pb.a.y);
  PresekP.b.x := Min(pa.b.x, pb.b.x);
  PresekP.b.y := Min(pa.b.y, pb.b.y);
  Presek := Max(PresekP.b.x − PresekP.a.x, 0) *
            Max(PresekP.b.y − PresekP.a.y, 0);
end; {Presek}

function Unija(var pa, pb: Pravokotnik): real;
begin
  Unija :=   (pa.b.x − pa.a.x) * (pa.b.y − pa.a.y)
           + (pb.b.x − pb.a.x) * (pb.b.y − pb.a.y)
           − Presek(pa, pb);
end; {Unija}

var a, b: Pravokotnik;
begin {UnijaPravokotnikov}
  Write('1. pravokotnik, točka A: '); ReadLn(a.a.x, a.a.y);
  Write('1. pravokotnik, točka B: '); ReadLn(a.b.x, a.b.y);
  Write('2. pravokotnik, točka A: '); ReadLn(b.a.x, b.a.y);
  Write('2. pravokotnik, točka B: '); ReadLn(b.b.x, b.b.y);
  WriteLn('Presek = ', Presek(a, b):0:3);
  WriteLn('Unija  = ', Unija(a, b):0:3);
end. {UnijaPravokotnikov}
```

N: 213

R1995.1.4

Preverimo, če prvi pomožni procesor deluje — to storimo tako, da ga primerjamo z vsemi ostalimi procesorji in če večina pravi, da je pokvarjen (različen od njih), potem je pač res pokvarjen. Vemo namreč, da je večina procesorjev še vedno pri pameti in torej zmožna pravilno ocenjevati druge procesorje. Nato preverimo drugi procesor, pa tretjega in tako naprej. Ko naletimo na delujoč procesor, zaključimo z iskanjem. Nato delujoči procesor primerjamo z vsemi pomožnimi procesorji, da vidimo, kateri so pokvarjeni.

```
program Preverjaj;
const N = 50;
var Pokvarjen: array [1..N] of boolean;
    i, j, Delujocih: integer;
begin
  j := 0;
  repeat
```

```
      j := j + 1;
      { Na tem mestu že vemo, da so vsi procesorji od 1 do j − 1 pokvarjeni
        (sicer bi se tale zanka že končala). O procesorju j zaenkrat predpostavimo,
        da je tudi pokvarjen. Spremenljivka Delujocih šteje v resnici to, koliko
        procesorjev (ne vštevši j-ja samega) meni, da j deluje pravilno. }
      Pokvarjen[j] := true; Delujocih := 0;
      { Poglejmo zdaj, kaj pravijo o j-ju ostali procesorji.
        Tistih od 1 do j − 1 nima smisla spraševati, ker vemo, da so pokvarjeni. }
      for i := j + 1 to N do
        if Enaka(j, i) then Delujocih := Delujocih + 1;

      { Vemo, da je pokvarjenih manj kot pol procesorjev. Ločimo naslednje možnosti:
        N = 2K + 1 (lih), torej vsaj K + 1 dobrih, največ K slabih.
          j pokvarjen → vsaj K + 1 jih bo reklo, da je zanič;
                        torej jih bo največ K − 1 reklo, da je dober.
          j dober     → vsaj K jih bo reklo, da je dober.
        N = 2K (sod), torej vsaj K + 1 dobrih, največ K − 1 slabih.
          j pokvarjen → vsaj K + 1 jih bo reklo, da je zanič;
                        torej jih bo največ K − 2 bo reklo, da je dober.
          j dober     → vsaj K jih bo reklo, da je dober.
        Torej je j dober natanko tedaj, ko je vsaj K = N div 2 drugih procesorjev }
    until Delujocih >= N div 2;                          { reklo, da je dober. }

    { To, da smo prišli do sem, pomeni, da smo ugotovili, da deluje j pravilno.
      Primerjajmo zdaj ostale procesorje z njim. }
    for i := 1 to N do Pokvarjen[j] := not Enaka(j, i);
    { Zdaj imamo za vse procesorje v tabeli Pokvarjen podatke o tem,
      ali delujejo pravilno ali ne. }
end. {Preverjaj}
```

REŠITVE NALOG ZA DRUGO SKUPINO

R1995.2.1

N: 213

Funkcija KajVrnem pove, kateri po velikosti je element x v tabeli t. Program ima drobno napako: če je v tabeli t več elementov z vrednostjo x, se funkcija vrti v neskončni zanki. O tem se lahko prepričamo takole: prej ali slej eden od števcev a in b pride do ene od celic z vrednostjo x. Ko se nato izvede zamenjava, je zdaj pač drugi števec pri celici z vrednostjo x; zato se vsaj ta števec pri naslednji iteraciji zunanje zanke ne bo nič premaknil. Tisti števec, ki se še premika, pa bo prej ali slej naletel na še kakšno drugo celico z vrednostjo x in odtlej se števca ne bosta nikoli več premaknila, program pa bo besno zamenjeval najdena x-a. (Če je v tabeli x en sam, do tega problema ne more priti, saj prej ali slej oba števca prideta do celice z vrednostjo x, ker pa zdaj oba kažeta na isto celico, pogoj a < b v zunanji zanki ni izpolnjen in zanka se konča.) Težavi se izognemo tako, da pogoj t[b] > x spremenimo v t[b] >= x ali pa po vsaki zamenjavi povečamo a

za 1 in zmanjšamo b za 1; eno ali drugo nam zagotovi, da se a in b v vsaki iteraciji zunanje zanke zbližata za vsaj eno mesto, tako da se zanka ne more izvajati v nedogled.

Ta funkcija tudi preuredi elemente tabele: tisti, ki so manjši od x, pridejo v levi del tabele, tisti, ki so večji od x, pa v desni del tabele. Tako funkcijo se običajno uporablja pri algoritmu quicksort za urejanje podatkov: ker so zdaj vsi elementi levega dela manjši ali enaki od vseh v desnem delu, bomo tabelo čisto uredili že s tem, da bomo (z rekurzivnim klicem) uredili levi del posebej in nato še desni del posebej.

N: 214

R1995.2.2

O vsakem bloku, ki se trenutno nahaja v pomožnem pomnilniku, moramo hraniti poleg številke tega bloka in njegove vsebine (struktura Podatki) še čas, kdaj je bil nazadnje uporabljen (da bomo vedeli, kateri najdlje ni bil uporabljen in ga je zato pametno zavreči, če potrebujemo prostor v pomnilniku), in zastavico, ki pove, ali je bil, odkar smo ga prebrali z diska, že kaj spremenjen (da bomo vedeli, ali ga je treba zapisati na disk, preden ga zavržemo iz pomnilnika).

Za čas zadnjega dostopa ni treba, da je to pravi čas; lahko imamo kar nek števec kot globalno spremenljivko in jo ob vsakem dostopu povečamo za 1. Že to je dovolj, da bomo lahko videli, ali je bil nek blok nazadnje uporabljen kasneje kot nek drug blok.

Podprograma ZapisiBlok in PreberiBlok si lahko pomagata s pomožnim podprogramom, ki se spodaj imenuje ZagotoviProstorZaEnBlok. Njegova naloga je poskrbeti, da bo v pomnilniku dovolj prostora za nov blok; če je treba, bo zavrgel nek drug blok iz pomnilnika. Preden zavržemo blok iz pomnilnika, je treba seveda tudi preveriti, če je bil kaj spremenjen, in ga v tem primeru najprej fizično zapisati na disk.

```
procedure ZapisiBlok(StBloka: integer; P: Podatki);
1  Če bloka StBloka še ni v pomnilniku:
2      ZagotoviProstorZaEnBlok;
3      Dodaj v pomnilnik blok StBloka s podatki P;
4  Sicer:
5      Podatki bloka StBloka v pomnilniku := P;
6  Označi, da je bil blok StBloka spremenjen
   in da je bil pravkar opravljen dostop do njega.
```

```
procedure PreberiBlok(StBloka: integer; var P: Podatki);
1  Če bloka StBloka še ni v pomnilniku:
2      ZagotoviProstorZaEnBlok;
3      PreberiBlokZDiska(StBloka, podatki tega bloka);
4      Označi blok StBloka v pomnilniku kot nespremenjenega.
5  P := podatki tega bloka v pomnilniku;
```

6 Označi, da je bil pravkar opravljen dostop do bloka StBloka.

procedure ZagotoviProstorZaEnBlok;
1 Če je v pomnilniku že MaksPPBlokov blokov:
2 Naj bo b tisti blok v pomnilniku, do katerega že najdlje nismo dostopali.
3 Če je bil b že kdaj spremenjen:
4 ZapisiBlokNaDisk(b, njegovi podatki);
5 Zavrzimo b iz pomnilnika.

Ostane nam še vprašanje, kako naj bodo zapisi o posameznih blokih organizirani v pomnilniku. Želimo si, da bi naša podatkovna struktura čim bolj učinkovito podpirala naslednji dve operaciji:

1. preveriti, ali je nek blok že v pomnilniku (in priti do njegovih podatkov);

2. poiskati tisti blok, ki že najdlje ni bil uporabljen (tu moramo upoštevati tudi, saj se bo čas zadnje uporabe blokom pogosto spremenil, pač ob vsaki uporabi).

Bloke bi lahko na primer povezali v dvosmerno povezano verigo (*doubly linked list*), v kateri bi bili urejeni po času zadnjega dostopa. Na začetku seznama bi bil tisti, do katerega smo nazadnje dostopali, tako da bi bil vedno pri roki. Ko nek blok na novo uporabimo, moramo njegovo celico prestaviti na začetek seznama, kar pri dvosmerno povezanem seznamu ni težko. Kako pa bi najenostavneje ugotovili, ali je nek blok že v pomnilniku (in prišli do njemu pripadajoče celice v prej omenjenem dvosmernem seznamu)? Lahko bi preiskali cel seznam, vendar ima lahko ta do MaksPPBlokov elementov, kar najbrž vendarle ni tako malo. Lahko bi imeli tabelo, kjer bi za vsak blok na disku pisalo, ali je (oz. kje je) v pomnilniku; potem bi bilo preverjanje, ali je nek blok v pomnilniku, hitro, vendar bi imela ta tabela MaksBlokov elementov, kaj je najbrž precej preveč (saj je sorazmerno z velikostjo diska). Verjetno je še najbolje pripraviti razpršeno tabelo (*hash table*), kjer je poraba pomnilnika sorazmerna z MaksPPBlokov, čas iskanja pa je približno konstanten (neodvisen od MaksPPBlokov), če se bloki dovolj enakomerno razpršijo.

Našega upravitelja predpomnilnika bi bilo koristno dopolniti tudi s tem, da bi v ozadju občasno (npr. če že kakšno sekundo ni bil izveden dostop do diska) shranil na disk nekaj izmed tistih blokov, ki so trenutno v predpomnilniku in so označeni kot spremenjeni (kar pomeni, da je vsebina tega bloka že stara in neveljavna, trenutno aktualni podatki zanj pa so le v pomnilniku); na primer take, ki se že najdlje niso spremenili (pri takih upamo, da se tudi v bližnji prihodnosti ne bodo, tako da naše zapisovanje trenutne vsebine teh blokov na disk ne bo le nekoristna potrata časa). S tem bi skrbeli za to, da bi ob primeru izpada elektrike ali česa podobnega izgubili čim manj podatkov. Pri tem pa

takšno občasno zapisovanje na disk v ozadju ne bi uporabnika nič motilo niti kako drugače upočasnjevalo dela z računalnikom.

N: 215

R1995.2.3

Za začasno hranjenje znakov, ki smo jih že prebrali z vhodne linije, nismo pa jih še posredovali naprej vsem izhodnim linijam, bomo uporabili vrsto (tabela Vrsta v spodnjem programu). Znake, ki jih prebiramo z vhodne linije, dodajajmo na konec vrste (Rep). Izhodne linije pa imajo vsaka svoj kazalec na prvi znak v vrsti, ki ga še nismo posredovali posamezni izhodni liniji (Glava). Izmenično bomo poskušali prebirati vhodne znake in pisati na izhode; pri branju vhoda se ustavimo, če vhodnih znakov zmanjka ali pa vrsta doseže največjo dovoljeno dolžino (saj naloga pravi, da je količina pomnilnika, ki je na voljo za vrsto, omejena). Pri pisanju na izhode poskusimo pisati na vsako izhodno linijo po vrsti od prvega znaka, ki ga na to linijo še nismo uspešno poslali; ko ne moremo več pošiljati nanjo ali pa smo poslali že vse, pa se lotimo naslednje izhodne linije. Ker je največja dolžina vrste omejena vnaprej, lahko vrsto oblikujemo kot preprost krožni pomnilnik (*ring buffer*) — ko pri branju ali pisanju pridemo do konca vrste, začnemo spet na začetku.

```
program Multicasting(Output);
const
  IzhodM = 20;          { število izhodnih kanalov (linij) }
  VrstaM = 10000;       { največje možno število znakov v čakalni vrsti }
var
  Vrsta: array [1..VrstaM] of char;  { krožni izravnalni pomnilnik }
  Rep: integer;         { kazalec na prvo prazno mesto v tabeli Vrsta }
  Glava:                { za vsak izhodni kanal svoj kazalec na rep vrste }
    array [1..IzhodM] of integer;
  Dolz:                 { za vsak izhodni kanal dolžina vrste (število }
    array [1..IzhodM] of integer;                { čakajočih znakov) }
  MaxDolz: integer;     { dolžina najdaljše vrste: max(Dolz[*]) }
  Prebrano: integer;    { število pravkar prebranih znakov }
  iz: integer;          { številka izhodnega kanala }
  Konec: boolean;

  function Beri(var ch: char): boolean; external;
  function Pisi(iz: integer; ch: char): boolean; external;

begin
  Rep := 1;
  for iz := 1 to IzhodM do begin Glava[iz] := 1; Dolz[iz] := 0 end;
  while true do begin
    { Ugotovi število znakov v spominu (dolžino najdaljše vrste). }
    MaxDolz := Dolz[1];
    for iz := 2 to IzhodM do
      if Dolz[iz] > MaxDolz then MaxDolz := Dolz[iz];
```

```
      { Beri, dokler ne zmanjka znakov ali prostora v pomnilniku. }
      Konec := false; Prebrano := 0;
      while (MaxDolz + Prebrano < VrstaM) and not Konec do
        if not Beri(Vrsta[Rep]) then Konec := true
        else begin Prebrano := Prebrano + 1;
                   Rep := (Rep mod VrstaM) + 1 end;
      { Piši na vsak izhod, dokler se ne zatakne ali ne zmanjka znakov. }
      for iz := 1 to IzhodM do begin
        Dolz[iz] := Dolz[iz] + Prebrano;
        Konec := false;
        while (Dolz[iz] > 0) and not Konec do
          if not Pisi(iz, Vrsta[Glava[iz]]) then Konec := true
          else begin Dolz[iz] := Dolz[iz] - 1;
                     Glava[iz] := (Glava[iz] mod VrstaM) + 1 end;
      end; {for}
    end; {while}
end. {Multicasting}
```

R1995.2.4

N: 215

Računalnik najprej sporoči svoj naslov vsem sosedom. Ob predpostavki, da so vsi računalniki ravnali enako, lahko računalnik sedaj začne sprejemati naslove svojih sosedov, kasneje pa še naslove vseh ostalih računalnikov, ki so povezani v omrežje. Ko sprejme sporočilo, računalnik najprej preveri, če je bilo to prvo sporočilo z danega naslova. To preveri s pomočjo tabele SmeriRacunalnikov, kjer je za vsak naslov shranjena smer, iz katere je prispelo prvo sporočilo. Prvo sporočilo je seveda prišlo po najkrajši poti, zato je tabela SmeriRacunalnikov obenem tudi tabela najkrajših poti. Ko ugotovi, da je sprejel prvo sporočilo z danega naslova, smer sprejema shrani v tabelo, sporočilo pa pošlje še v vse ostale smeri. Na tak način računalnik sprejema in posreduje sporočila, dokler ne izve smeri za vseh devetindevetdeset naslovov.

```
program KamNajPosljem;

  function MojNaslov: integer; external;
  function Poslji(VSmer: integer; Sporocilo: integer): boolean; external;
  function Sprejmi(var IzSmeri: integer; var Sporocilo: integer): boolean; external;

const
  SteviloRacunalnikov = 100;
  SteviloSmeri = 5;

var
  VSmer, IzSmeri, SprejetiNaslov: integer;
  SmeriRacunalnikov: array [1..SteviloRacunalnikov] of integer;
  Stevec: integer;
  SteviloPoznanihRacunalnikov: integer;
```

```
begin
  SteviloPoznanihRacunalnikov := 1;
  for Stevec := 1 to SteviloRacunalnikov do
    SmeriRacunalnikov[Stevec] := 0;
  for VSmer := 1 to SteviloSmeri do Poslji(VSmer, MojNaslov);
  while SteviloPoznanihRacunalnikov < SteviloRacunalnikov do begin
    repeat until Sprejmi(IzSmeri, SprejetiNaslov);
    if (SmeriRacunalnikov[SprejetiNaslov] = 0) and
              (SprejetiNaslov <> MojNaslov) then begin
      SteviloPoznanihRacunalnikov := SteviloPoznanihRacunalnikov + 1;
      SmeriRacunalnikov[SprejetiNaslov] := IzSmeri;
      for VSmer := 1 to SteviloSmeri do
        if VSmer <> IzSmeri then Poslji(VSmer, SprejetiNaslov);
    end; {if}
  end; {while}
end. {KamNajPosljem}
```

REŠITVE NALOG ZA TRETJO SKUPINO

N: 216

R1995.3.1

Najzanimivejši del je podprogram RKSearch.[40] V osnovi gre za iskanje enega niza v drugem v času, ki je sorazmeren vsoti dolžin obeh nizov (namesto produktu dolžin, kar velja za naivni postopek iskanja podniza v nizu).

Naiven postopek za iskanje niza p (dolžine LP) v nizu a (dolžine LA) bi pregledal vse podnize a-ja, dolge po LP znakov (torej a[1..LP], a[2..LP + 1], ..., a[LA − LP + 1..LA]), in vsakega primerjal z nizom p. Če imamo smolo in se p vedno čisto ujema z opazovanim podnizom, bo vsaka taka primerjava izvedla LP primerjav posameznih znakov. Časovna zahtevnost celotnega postopka je zato približno sorazmerna s produktom LA · LP.

Recimo pa, da bi znali vsakemu nizu prirediti nek celoštevilski „indeks“. Enak niz dobi vedno enak indeks, različni nizi pa po možnosti različne indekse (čeprav se ne da povsem izogniti temu, da včasih različni nizi dobijo isti indeks). Potem lahko za začetek primerjamo indeks niza p z indeksom posameznega podniza a[i..i + LP − 1]; če se indeksa razlikujeta, vemo, da je p različen od a[i..i + LP − 1]. Če pa sta indeksa enaka, je prav mogoče, da sta tudi niza enaka, povsem nujno pa to ni, zato ju moramo primerjati še na tradicionalen način, znak po znak (resna slabost programa iz naše naloge je, da tega ne dela; srečo ima, da pri nizih, s katerimi ima pri tej nalogi opravka, to ne pripelje do težav, pri kakšnih drugih nizih pa bi bile lahko težave).

Lepo pri tem postopku je, da lahko indekse definiramo tako, da je mogoče

[40] Imenuje se po Rabinu in Karpu, ki sta prva predlagala takšen postopek za iskanje podnizov v nizih. Glej npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 34.2 v prvi izdaji, 32.2 v drugi.

indeks niza a[i + 1..i + LP] izračunati zelo poceni, če že poznamo indeks niza a[i..i + LP − 1].

Izberimo si neki naravni števili d in q. Niz $s = s_1 s_2 \ldots s_n$ si predstavljajmo kot velikansko celo število, zapisano v d-iškem sestavu; posamezni znaki $s_1, \ldots, s_n$ so pri tem njegove „števke“. Niz s torej predstavlja število

$$d^{n-1}s_1 + d^{n-2}s_2 + \ldots + d^2 s_{n-2} + d\, s_{n-1} + s_n.$$

Za njegov indeks pa vzemimo vrednost

$$(d^{n-1}s_1 + d^{n-2}s_2 + \ldots + d^2 s_{n-2} + d\, s_{n-1} + s_n) \bmod q.$$

Indeksi so torej vedno cela števila iz množice $\{0, 1, \ldots, q-1\}$. Za q je koristno vzeti kakšno praštevilo; tako naredi tudi naš program.

Označimo indeks podniza a[i..i + LP − 1] z x; indeks naslednjega podniza, a[i + 1..i + LP], pa z x'. Iz gornje formule sledi, da je

$$x = (d^{LP-1}a[i] + d^{LP-2}a[i+1] + \ldots + da[i+LP-2] + a[i+LP-1]) \bmod q$$

in

$$x' = (d^{LP-1}a[i+1] + d^{LP-2}a[i+2] + \ldots + da[i+LP-1] + a[i+LP]) \bmod q.$$

Koristna lastnost operacije mod (ostanek po deljenju) je, da jo lahko opravimo kadarkoli, pa vseeno dobimo enak odgovor. Drugače povedano, če izračunamo ostanek deljenja s q po vsaki aritmetični operaciji (tako bomo imeli vedno opraviti samo z majhnimi števili), dobimo enak rezultat, kot če bi opravili vse aritmetične operacije in šele potem izračunali ostanek deljenja s q.

Če primerjamo izraza za x in x' in upoštevamo lastnosti operacije mod, vidimo:

$$x' = ((x - d^{LP-1}a[i]) \cdot d + a[i+LP]) \bmod q.$$

Zaradi lastnosti operacije mod tudi sledi, da lahko v tem izrazu namesto d^{LP-1} vzamemo $d^{LP-1} \bmod q$ (to vrednost si naš program pripravi v spremenljivki dM). Tako vidimo, da lahko x' izračunamo iz x le s peščico računskih operacij, ne glede na to, kako velik je LP.

Podprogram Replace preprosto zamenja najdeni niz z nizom r: najprej premakne rep niza za potrebno število znakov levo ali desno in potem vpiše v niz nove vrednosti.

Program izpiše vrednost niza po vsaki zamenjavi in popravi mesto zadnjega iskanja tako, da se to začne po koncu zadnje zamenjave. Zadnja izpisana vrstica je torej: „`alibabababa in stirideset babarbabarov.`“

Poleg že omenjenega dejstva, da se zadovolji z ujemanjem indeksov in ne gre preverjat še ujemanja nizov, je v programu še nekaj drugih zanikrnosti. V podprogramu Replace imamo zanko:

```
for k := LA + 100 downto j + LP do a[k] := a[k − (LR − LP)];
```

Namen te zanke je premakniti „rep“ niza a: a[j + LP..LA] → a[j + LR..LA + LR − LP]. Nekaj napak v njej: (1) Če je LA + LR − LP (kar bo dolžina niza a po zamenjavi) večje od dolžine tabele (v našem primeru 255 znakov), bi morali javiti napako. (2) Če je začetna vrednost k = LA + 100 večja od 255, celica a[k] sploh ne obstaja. (3) Če pa bi bila razlika LR − LP večja od 100, ta zanka ne bi premaknila celega „repa“, ampak le sto znakov. Problema (2) in (3) rešimo tako, da zanko začnemo pri k = LA + LR − LP. (4) Zanka gre do k = j + LP namesto k = j + LR. Tako si v najboljšem primeru nakopava nekaj nepotrebnega dela (ker bo tisto, kar vpiše v a[j + LP..j + LR − 1], takoj povozila zanka v naslednji vrstici), če pa imamo opravka z neko pojavitvijo p-ja bolj na začetku niza a, je j majhen in celica a[k − (LR − LP)] je neveljavna.

Neroden je tudi stavek

```
j := j + Length(r) − Length(p);
```

v glavnem bloku programa. Ob vrnitvi iz podprograma RKSearch nam j pove (če je neničeln), da je RKSearch našel pojavitev niza p na mestih a[j..j + LP − 1]. Po zamenjavi se ta podniz zamenja z nizom r, ki pokrije mesta a[j..j + LR − 1]. Preiskovanje niza a se torej spodobi nadaljevati od a[j + LR] naprej. Podprogram RKSearch, ko ga pokličemo, primerja p najprej z a[j + 1..j + LP], torej bi morali mi zdaj pred naslednjim klicem RKSearch postaviti

```
j := j + Length(r) − 1;
```

Različica iz besedila naloge, torej tista z Length(p) namesto 1, bi na primer iz a = yxx, p = yx, r = xxxy dobila niz xxxxxxy namesto pričakovanega xxxyx.

Še ena majhna slabost našega programa je, da se kliče RKSearch po enkrat za vsako zamenjavo in pri tem vsakič znova tudi izračuna indeks niza p. Če bi bilo v nizu a veliko pojavitev p-ja, bi bilo lahko to zelo potratno; takrat bi res morali izračunati indeks p-ja samo enkrat in si ga potem zapomniti v kakšni spremenljivki.

N: 217

R1995.3.2

Preprosta rešitev problema bi bila lahko takšna: za vsako mesto na zaslonu poglejmo, katere števke bi utegnilo predstavljati (torej: pri katerih števkah bi bili res prižgani vsi tisti segmenti, ki so prižgani na zaslonu). Potem preglejmo (npr. z rekurzijo) vse kombinacije števk, ki imajo prižgane vse segmente, vidne na zaslonu; pri primeru iz besedila naloge bi na primer za prvo mesto dobili števki 3 in 8, za drugo pa 0, 2, 3, 7, 8 in 9, torej bi morali pregledati 12 kombinacij: 30, 32, 33, 37, 38, 39, 80, 82, 83, 87, 88 in 89. Za vsako kombinacijo števk preverimo, če je njihova vsota deljiva z 10; če je tako, je ta kombinacija ena od kandidatk za pravo rešitev naloge.

Naloga zahteva, naj poiščemo kombinacijo z najmanj popravljenimi segmenti, torej z najmanjšo razliko med kombinacijo in stanjem na zaslonu; ker pa pridejo v poštev tako ali tako le kombinacije, pri katerih gorijo vsi segmenti, ki so prižgani tudi na zaslonu, je zahteva po najmanj popravljenih segmentih enakovredna zahtevi po najmanjšem skupnem številu prižganih segmentov. Če torej trenutna kombinacija ustreza ostalim zahtevam in ima poleg tega še manj prižganih segmentov kot najboljša doslej znana kombinacija, si jo zapomnimo kot novo kandidatko za najboljšo rešitev.

Ko z rekurzijo postopoma sestavljamo kombinacije števk, je zelo koristno že sproti, po postavitvi vsake števke, preveriti, če ni skupno število prižganih segmentov slučajno že preseglo števila prižganih segmentov pri najboljši doslej znani rešitvi. Če se namreč zgodi kaj takega, nima smisla postavljati še preostalih števk, saj tako dobljena rešitev gotovo ne bo boljša od najboljše doslej znane. Tako prihranimo veliko časa, ki bi ga drugače po nepotrebnem potratili za pregledovanje neobetavnih kombinacij. Takšnemu pristopu k pregledovanju prostora možnih kombinacij zato pogosto pravijo „razveji in omeji“ (*branch and bound*) — ob vsakem rekurzivnem klicu se iskanje razveji, ko poskušamo na trenutno mesto postavljati različne možne števke; obenem pa se poskušamo s pogoji, kot je v našem primeru tale s številom segmentov, čim bolj omejiti, da nam ne bi bilo treba v celoti pregledati neobetavnih delov prostora.

```
program PokvarjeniZaslon(Output);
type Stevka = 0..9;
     Segment = 1..7;
     Segmenti = set of Segment;
const OpisiStevk: array [Stevka] of Segmenti = (
          [1, 2, 3, 5, 6, 7], [3, 6], [1, 3, 4, 5, 7], [1, 3, 4, 6, 7],
          [2, 3, 4, 6], [1, 2, 4, 6, 7], [1, 2, 4, 5, 6, 7],
          [1, 3, 6], [1, 2, 3, 4, 5, 6, 7], [1, 2, 3, 4, 6, 7] );
       SegmentovPriStevki: array [Stevka] of integer= (6, 2, 5, 5, 4, 5, 6, 3, 7, 6);
       MaxN = 10;
type ZaslonT = array [1..MaxN] of Segmenti;

procedure PoisciVrednosti(var Zaslon: ZaslonT; N: integer);
var
  { Trenutna in najboljša doslej znana kombinacija števk. }
  Trenutna, Najboljsa: array [1..MaxN] of integer;
  { Za vsako mesto na zaslonu (od 1 do N) imamo seznam števk,
    ki imajo prižgane vse tam vidne segmente. To so Mozne[i, 1..StMoznih[i]]. }
  StMoznih: array [1..MaxN] of integer;
  Mozne: array [1..MaxN, 1..10] of Stevka;
  MinStSegmentov, StSegmentov: integer;

  procedure Rekurzija(Mesto: integer);
  var i, j, Vsota, StaroStSegmentov: integer;
  begin
```

```
      StaroStSegmentov := StSegmentov;
      for j := 1 to StMoznih[Mesto] do begin
        Trenutna[Mesto] := Mozne[Mesto, j];
        StSegmentov := StaroStSegmentov + SegmentovPriStevki[Trenutna[Mesto]];
        if StSegmentov >= MinStSegmentov then
          continue; { Brezupno, segmentov je že zdaj preveč. }
        if Mesto < N then begin Rekurzija(Mesto + 1); continue end;

        { Preverimo, če ima trenutna rešitev vsoto, deljivo z 10. }
        Vsota := 0;
        for i := 1 to N do Vsota := Vsota + Trenutna[i];
        if Vsota mod 10 <> 0 then continue;

        { To je najboljša doslej znana rešitev — zapomnimo si jo. }
        MinStSegmentov := StSegmentov;
        for i := 1 to N do Najboljsa[i] := Trenutna[i];
      end; {for j}
      StSegmentov := StaroStSegmentov;
    end; {Rekurzija}


var i: integer; s: Stevka;
begin {PoisciVrednosti}
    { Za vsako mesto na zaslonu poglejmo, katere števke bi utegnile
      povzročiti tako stanje segmentov, kot ga vidimo na zaslonu. }
    for i := 1 to N do begin
      StMoznih[i] := 0;
      for s := 0 to 9 do if Zaslon[i] - OpisiStevk[s] = [] then
        begin StMoznih[i] := StMoznih[i] + 1; Mozne[i, StMoznih[i]] := s end;
      if StMoznih[i] = 0 then exit; { Do takega stanja sploh ne more priti! }
    end; {for i}

    MinStSegmentov := 7 * N + 1; StSegmentov := 0;
    Rekurzija(1);

    if MinStSegmentov > 7 * N then WriteLn('Ni rešitev!')
    else begin
      Write('Najboljša rešitev:');
      for i := 1 to N do Write(' ', Najboljsa[i]);
      WriteLn;
    end; {if}
end; {PoisciVrednosti}

const
    Primer1: ZaslonT = ([1, 3, 4, 6, 7], [1, 3], [], [], [], [], [], [], [], []);
    Primer2: ZaslonT = ([1, 3, 4, 6, 7], [1, 3], [2, 4], [1, 7], [1, 2], [4], [6, 7],
                        [1, 7], [1, 2, 3, 4], []);
begin {PokvarjeniZaslon}
    PoisciVrednosti(Primer1, 2);
```

```
  PoisciVrednosti(Primer2, 9);
end. {PokvarjeniZaslon}
```

Če naš zaslon nima veliko števk (torej: če N ni prevelik) in če na posameznem mestu ni možnih prav veliko števk, bo ta rešitev čisto dobra. Pri večjih zaslonih pa bi lahko število kombinacij, ki bi jih moral naš rekurzivni postopek pregledati, toliko naraslo, da bi se morali začeti ozirati za učinkovitejšo rešitvijo.

Zato si oglejmo še, kako bi se lahko naloge lotili z mehanizmi, ki se uporabljajo v logičnem programiranju z omejitvami (Constraint Logic Programming — CLP).[41] Problem bomo opisali z množico spremenljivk; vsaka od njih ima neko zalogo možnih vrednosti, poleg tega pa obstajajo tudi omejitve, ki zahtevajo, da morajo biti vrednosti več različnih spremenljivk v določeni zvezi. Omejitve nam lahko pomagajo, da nekatere vrednosti spremenljivk že vnaprej prepoznamo kot nemogoče; tako lahko upamo, da bomo pri izbiranju konkretnih vrednosti posameznih spremenljivk preizkusili čim manj neobetavnih možnosti.[42]

Na primer, recimo, da imamo spremenljivke x, y in z z zalogo vrednosti $\{1, 2, \ldots, 10\}$, poleg tega pa imamo omejitve $z < 7$, $x+y = z$ in $x \geq 2$. Na podlagi omejitve $z < 7$ lahko zalogo vrednosti spremenljivke z takoj zmanjšamo na $\{1, 2, \ldots, 6\}$. Zaradi omejitve $x + y = z$ potem vemo, da bo vsota vsaj 2, ker sta x in y oba velika vsaj 1; zato lahko zalogo vrednosti spremenljivke z zmanjšamo na $\{2, \ldots, 6\}$; obenem pa vemo, da niti x niti y ne smeta biti večja od 5 (ker je drugi seštevanec vsaj 1 in bi bila vsota sicer zanesljivo večja od 6, kar pa z ne sme biti), tako da lahko njuni zalogi vrednosti zmanjšamo na $\{1, \ldots, 5\}$. Nato lahko zaradi omejitve $x \geq 2$ zmanjšamo zalogo vrednosti spremenljivke x na $\{2, \ldots, 5\}$. Omejitev $x + y = z$ nam zdaj pove, da mora biti z vsaj 3 (ker je x vsaj 2 in y vsaj 1), tako da lahko z-jevo zalogo vrednosti zmanjšamo na $\{3, \ldots, 6\}$; pa tudi za y zdaj vemo, da sme biti največ 4, ker je x vsaj 2 in bi drugače prekoračili največjo dovoljeno vrednost z-ja, namreč 6: tako lahko y-ovo zalogo vrednosti zmanjšamo na $\{1, \ldots, 4\}$. Če nimamo nobenih drugih omejitev, zalog vrednosti ne bomo mogli nič bolj zmanjšati; končno stanje je torej: $x \in \{2, \ldots, 5\}$, $y \in \{1, \ldots, 4\}$, $z \in \{3, \ldots, 6\}$.

Tako okleščene zaloge vrednosti so dobro izhodišče za preizkušanje konkretnih vrednosti spremenljivk. Spremenljivki x lahko poskusimo eno za drugo prirediti neko konkretno vrednost iz doslej dobljene zaloge vrednosti (torej iz $\{2, \ldots, 5\}$). To si lahko predstavljamo kot začasno dodatno omejitev oblike

[41] Za več o CLP gl. npr. Ivan Bratko, *Prolog Programming for Artificial Intelligence*, 3. izd. (2001), 14. pogl.; Thom Frühwirth et al., *Constraint Logic Programming: An Informal Introduction*, Tech. Rept. ECRC-93-5, `http://www.clps.de/html/reports.html`; Manuel Carro et al., *An Introductory Course on Constraint Logic Programming*, dec. 1998, `http://clip.dia.fi.upm.es/~vocal/public_info/index.html`.

[42] S tem postopkom usklajevanja omejitev se podrobneje ukvarja tudi naloga 1996.3.3 (str. 265, rešitev na str. 286).

$x = \langle$neka konstanta$\rangle$. Ta dodatna omejitev nam omogoča še zmanjšati zalogi vrednosti ostalih spremenljivk. Nato poskusimo izbrati neko konkretno vrednost spremenljivke y, spet oklestiti zaloge vrednosti, nato pa enako storimo še za z. Če uspemo vsem prirediti konkretne vrednosti, ne da bi se ob usklajevanju omejitev zaloga vrednosti kakšne spremenljivke izpraznila, pomeni, da izbrane vrednosti spremenljivk res ustrezajo vsem omejitvam. V našem gornjem primeru bi lahko na primer začeli s tem, da bi izbrali $x = 3$, kar bi nam omogočilo zmanjšati zalogo vrednosti z-ja na $\{4, \ldots, 6\}$ (ker je y vsaj 1) in nato zalogo y-a na $\{1, 2, 3\}$ (ker je z največ 6). Nato bi izbrali nek konkreten y, na primer $y = 2$, omejitev $x + y = z$ bi nam zmanjšala zalogo vrednosti z-ja na $\{5\}$, po izboru vrednosti $z = 5$ pa vse omejitve še vedno veljajo, kar pomeni, da smo našli eno od možnih rešitev: $x = 3$, $y = 2$ in $z = 5$ ustrezajo vsem omejitvam z začetka primera. Izbiranje vrednosti spremenljivk bi izvajali rekurzivno, tako da se lahko ob vrnitvi iz rekurzivnega klica posvetimo še drugim vrednostim trenutne spremenljivke. Tako bi na primer poskusili še $y = 1$ in $y = 3$, vse to še vedno pri $x = 3$, ko pa bi končali s tem, bi se lotili še drugih vrednosti spremenljivke x in tam spet preizkušali razne možne y in z. Glavno pa je, da po vsakem izboru neke vrednosti spet pregledamo omejitve, v katerih tista spremenljivka nastopa, da vidimo, če lahko zaradi tega izbora vrednosti kaj zmanjšamo zaloge vrednosti preostalih spremenljivk.

Pri našem problemu bi bilo koristno imeti po eno spremenljivko za vsako števko opazovanega zaslona; recimo jim $c_1, \ldots, c_n$. Njena začetna zaloga vrednosti bi bile kar vse tiste števke (od 0 do 9), ki vsebujejo vse segmente, ki so vidni na tem delu zaslona. Zdaj bi potrebovali omejitev, da mora biti vsota deljiva z 10; da pa posamezne omejitve ne bodo prezapletene, uvedimo raje posebno spremenljivko v in omejitvi $v = c_1 + \cdots + c_n$ ter $v \bmod 10 = 0$. Preostane nam še zahteva, da mora biti razlika v številu segmentov med c_i in tem, kar se res vidi na zaslonu, čim manjša; v ta namen uvedimo spremenljivke r_i ter omejitve $r_i = \mathit{Razlika}(c_i, \mathit{Zaslon}[i])$, kar nam bo predstavljalo zahtevo, da je r_i razlika v številu vidnih segmentov med vrednostjo c_i in tem, kar se res vidi na i-ti števki zaslona. Končno je tu še spremenljivka r z omejitvama $r = r_1 + \cdots + r_n$ in $r < R$, pri čemer je R konstanta, ki predstavlja skupno razliko pri najboljši doslej najdeni rešitvi. Vsakič, ko najdemo neko novo rešitev, ki ustreza vsem dosedanjim omejitvam, bomo R zmanjšali in program s tem prisilili, da bo v bodoče odkrival le še boljše rešitve (če jih je kaj).

Program lahko zdaj deluje tako, da najprej uskladi vse omejitve, kolikor se to le da. Vsakič, ko se zaloga vrednosti neke spremenljivke zmanjša, si je treba ponovno ogledati omejitve, v katerih ta spremenljivka nastopa, ker nam lahko te zdaj omogočijo zmanjšati zaloge vrednosti še kakšnih drugih spremenljivk. Zato vzdržujemo med usklajevanjem množico spremenljivk, ki se jim je zaloga vrednosti spremenila in bo treba njihove omejitve ponovno pregledati; ko se ta množica izprazni, vemo, da smo z usklajevanjem zmanjšali zaloge vrednosti,

kolikor se je le dalo.

V nadaljevanju izvajamo rekurzivne klice, ki skušajo po vrsti izbrati neke konkretne vrednosti za spremenljivke $c_1, \ldots, c_n$, po vsakem pa spet poženemo usklajevanje, da bi čim bolj zmanjšali zaloge vrednosti. Če se zaloga vrednosti kakšne spremenljivke pri tem izprazni, rekurzija ne bo mogla nadaljevati in vemo, da z doslej izbranimi vrednostmi pač ni mogoče dobiti nobene rešitve.

Spodnji program je napisan v pythonu, ker je tako lahko krajši, kot bi bil v pascalu (pa še tako je dovolj dolg). Nekaj opomb glede pythonove sintakse: metode z imenom **__init__** so pravzaprav konstruktorji, izrazi v oglatih oklepajih so seznami (pravzaprav dinamične tabele), izraz oblike [f(a) **for** a **in** L **if** p(a)] pa sestavi seznam, v katerem je vrednost f(a) za vsak tak element a seznama L, ki ustreza pogoju p(a).

```
StSegmentov = 7
OpisiStevk = [   [1, 2, 3, 5, 6, 7], [3, 6], [1, 3, 4, 5, 7], [1, 3, 4, 6, 7],
                 [2, 3, 4, 6], [1, 2, 4, 6, 7], [1, 2, 4, 5, 6, 7],
                 [1, 3, 6], [1, 2, 3, 4, 5, 6, 7], [1, 2, 3, 4, 6, 7]   ]
StStevk = len(OpisiStevk)

def Podmnozica(pod, mnozica):
  for i in pod:
    if not i in mnozica: return False
  return True

def RazlikaMnozic(odstej, odMnozice):
  return [i for i in odMnozice if not i in odstej]

class Spremenljivka:
  # Atributi tega razreda: zaloga — seznam možnih vrednosti;
  # zaloge — sklad prejšnjih seznamov „zaloga“ (za backtracking);
  # omejitve — seznam omejitev, v katerih nastopa ta spremenljivka;
  # min, max — najmanjša in največja vrednost iz seznama „zaloga“.
  vse = []    # statičen seznam vseh spremenljivk v sistemu
  def __init__(self, zaloga = []):
    self.omejitve = []; self.NovaZaloga(zaloga[:], [])
    Spremenljivka.vse.append(self); self.zaloge = []
  def NovaZaloga(self, novaZaloga, spremenjene):
    self.zaloga = novaZaloga
    if self.zaloga == []: self.min = 9999; self.max = -9999
    else: self.min = min(self.zaloga); self.max = max(self.zaloga)
    if not self in spremenjene: spremenjene.append(self)

def PushZaloge():
  for s in Spremenljivka.vse: s.zaloge.append(s.zaloga[:])
def PopZaloge():
  for s in Spremenljivka.vse: s.NovaZaloga(s.zaloge.pop(), [])
```

```
class Omejitev: pass

class Manjsa(Omejitev):
  # Atributa: s — neka spremenljivka; meja — neko celo število.
  # To predstavlja omejitev „s < meja“.
  def __init__(self, spremenljivka, meja):
    self.s = spremenljivka; self.meja = meja
    self.s.omejitve.append(self)
  def uskladi(self, spremenjene):
    if self.s.max >= self.meja:
      self.s.NovaZaloga([i for i in self.s.zaloga if i < self.meja], spremenjene)

class Vsota(Omejitev):
  # Atributa: v — neka spremenljivka; s — seznam spremenljivk.
  # Predstavlja omejitev „v == s[0] + s[1] + ... + s[len(s) - 1]“.
  def __init__(self, vsota, sestevanci):
    self.v = vsota; self.sestevanci = sestevanci;
    self.v.omejitve.append(self)
    for s in self.sestevanci: s.omejitve.append(self)
  def uskladi(self, spremenjene):
    while True:
      minVsota = 0; maxVsota = 0; spremembe = False
      # Seštevanci lahko vplivajo na vsoto.
      for s in self.sestevanci: minVsota += s.min; maxVsota += s.max;
      if len(self.v.zaloga) > 0 and (minVsota > maxVsota
                                     or (self.v.min < minVsota or self.v.max > maxVsota)):
        self.v.NovaZaloga([i for i in self.v.zaloga
                                     if minVsota <= i <= maxVsota], spremenjene)
        spremembe = True
      # Vsota lahko vpliva na seštevance.
      for s in self.sestevanci:
        maxS = self.v.max - (minVsota - s.min)
        minS = self.v.min - (maxVsota - s.max)
        if len(s.zaloga) > 0 and (len(self.v.zaloga) == 0
                                                  or (s.min < minS or maxS < s.max)):
          s.NovaZaloga([i for i in s.zaloga if minS <= i <= maxS], spremenjene)
          spremembe = True
      if not spremembe: return

class Ostanek(Omejitev):
  # Atributi: s — neka spremenljivka; d, o — celi števili.
  # Predstavlja omejitev „s % d == o“.
  def __init__(self, spremenljivka, delitelj, ostanek):
    self.s = spremenljivka; self.d = delitelj; self.o = ostanek
    self.s.omejitve.append(self)
  def uskladi(self, spremenjene):
    novaZaloga = [i for i in self.s.zaloga if i % self.d == self.o]
```

```
    if len(novaZaloga) != len(self.s.zaloga):
      self.s.NovaZaloga(novaZaloga, spremenjene)

class Razlika(Omejitev):
  # Atributi: r, c — dve spremenljivki; vidni — seznam segmentov.
  # Predstavlja omejitev „r == moč mnozice(OpisiStevk[c] − vidni)“.
  def __init__(self, razlika, stevka, vidniSegmenti):
    self.r = razlika; self.c = stevka; self.vidni = vidniSegmenti
    self.r.omejitve.append(self); self.c.omejitve.append(self)
  def uskladi(self, spremenjene):
    while True:
      spremembe = False
      # Števka lahko vpliva na razlike.
      mozneRazlike = [len(RazlikaMnozic(self.vidni, OpisiStevk[i]))
                                                    for i in self.c.zaloga]
      novaZaloga = [i for i in self.r.zaloga if i in mozneRazlike]
      if len(novaZaloga) != len(self.r.zaloga):
        self.r.NovaZaloga(novaZaloga, spremenjene); spremembe = True
      # Razlike lahko vplivajo na števko.
      mozneStevke = [i for i in self.c.zaloga
                       if len(RazlikaMnozic(self.vidni, OpisiStevk[i])) in self.r.zaloga]
      if len(mozneStevke) != len(self.c.zaloga):
        self.c.NovaZaloga(mozneStevke, spremenjene); spremembe = True
      if not spremembe: return

def UskladiOmejitve(omejitve):
  while len(omejitve) > 0:
    spremenjene = []   # Spremenljivke, ki se jim je zaloga vrednosti spremenila.
    for o in omejitve: o.uskladi(spremenjene)
    # V nadaljevanju bo treba uskladiti vse omejitve,
    # ki vsebujejo kakšno spremenljivko iz seznama „spremenjene“.
    omejitve = []
    for s in spremenjene:
      for o in s.omejitve:
        if not o in omejitve: omejitve.append(o)

def PrirediVrednost(i):
  global n, spStevke, spRazlike, spVsota, spVsotaRazlik, omejitve

  if i == n:
    print "Rezultat: %s, vsota = %s, vsota razlik = %s" % (
      [c.zaloga[0] for c in spStevke], spVsota.zaloga, spVsotaRazlik.zaloga)
    assert len(spVsotaRazlik.zaloga) == 1
    # Našli smo novo najboljšo rešitev. Ta zdaj postane omejitev
    # pri nadaljnjem iskanju. Zadnja omejitev v seznamu „omejitve“
    # je ravno omejitev „vsota razlik < doslej najboljša znana vsota razlik“.
    omejitve[len(omejitve) − 1].meja = spVsotaRazlik.zaloga[0]
```

```
      return

    zaloga = spStevke[i].zaloga[:]
    for j in zaloga:
      PushZaloge()
      spStevke[i].NovaZaloga([j], []) # Izberimo naslednjo možno vrednost j-te števke.
      UskladiOmejitve(spStevke[i].omejitve)
      PrirediVrednost(i + 1)
      PopZaloge()    # backtrack
      # Če smo medtem našli kakšno novo boljšo rešitev, lahko zaloge vrednosti
      UskladiOmejitve([omejitve[-1]])                    # še malo poklestimo.

def BruteForce(i, stevke, vsota, razlika, vidniSegmenti):
    global bfBest, bfBestR
    if i == n:
      if vsota % 10 == 0 and razlika < bfBestR:
        print "BruteForce: %s (vsota = %d, razlika = %d)" % (stevke,
                                                                vsota, razlika)
        bfBest = stevke; bfBestR = razlika
    else:
      for j in spStevke[i].zaloga:
        novaRazlika = razlika + len(RazlikaMnozic(vidniSegmenti[i], OpisiStevk[j]))
        if novaRazlika < bfBestR:
          BruteForce(i + 1, stevke + [j], vsota + j, novaRazlika, vidniSegmenti)

def PoisciVrednosti(vidniSegmenti):
    global n, spStevke, spRazlike, spVsota, spVsotaRazlik, omejitve
    import time

    n = len(vidniSegmenti)
    # Pripravimo primerke razreda Spremenljivka.
    spStevke = []      # spStevke[i] predstavlja števko na i-tem mestu na zaslonu,
    spRazlike = []     # spRazlike[i] predstavlja stevilo segmentov, po katerih se to
    for i in range(n):                # mesto zaslona razlikuje od števke spStevke[i].
      # Začetna zaloga vrednosti vsake števke je množica
      # tistih števk, ki vsebujejo vse segmente, ki so vidni na tem mestu.
      spStevke.append(Spremenljivka([j for j in range(StStevk)
                                       if Podmnozica(vidniSegmenti[i], OpisiStevk[j])]))
      spRazlike.append(Spremenljivka(range(0, StSegmentov + 1)))
    spVsota = Spremenljivka(range(0, n * (StStevk - 1) + 1))          # vsota števk
    spVsotaRazlik = Spremenljivka(range(0, n * StSegmentov + 1))     # vsota razlik

    # Za primerjavo poženimo še branch-and-bound.
    global bfBestR; bfBestR = StSegmentov * n + 1; t = time.clock()
    BruteForce(0, [], 0, 0, vidniSegmenti)
    print "Brute-force je tekel %.3f s." % (time.clock() - t)

    t = time.clock()
```

```
  omejitve = []
  omejitve.append(Vsota(spVsota, spStevke))      # spVsota je vsota števk
  omejitve.append(Ostanek(spVsota, 10, 0))       # spVsota % 10 == 0
  for i in range(n):
    omejitve.append(Razlika(spRazlike[i], spStevke[i], vidniSegmenti[i]))   # razlike
  omejitve.append(Vsota(spVsotaRazlik, spRazlike))   # vsota razlik
  # Naslednja omejitev je zdajle trivialna, kasneje pa bomo njeno mejo zmanjševali,
  # ko bomo odkrili kakšno rešitev. Tako bo ta omejitev vedno zahtevala,
  # da mora biti naslednja rešitev boljša od doslej najboljše znane.
  omejitve.append(Manjsa(spVsotaRazlik, StSegmentov * n))
  UskladiOmejitve(omejitve)
  PrirediVrednost(0)   # poženimo izbiranje vrednosti spremenljivk spStevke[0..n-1]
  print "CLP je tekel %.3f s." % (time.clock() - t)

PoisciVrednosti([[1, 3, 4, 6, 7], [1, 3]]) # Primer iz besedila naloge.
# Še en večji primer:
PoisciVrednosti([[1, 3, 4, 6, 7], [1, 3], [2, 4], [1, 7], [1, 2], [4], [6, 7], [1, 7], [1, 2, 3, 4]])
```

R1995.3.3

Najprej poiščimo vse ljudi, ki so skupni šefi vsem delavcem iz dane množice S; nato pa preverjajmo, če ni kakšen od njih šef kakšnega drugega (v tem primeru vemo, da tisti drugi ne more biti najbližji skupni šef). Kar ostane, so najbližji skupni šefi.

N: 218

```
const n = ....; { Število oseb. }
type Oseba = 1..n; { Številka osebe. }
     MnozicaOseb = array [Oseba] of boolean;
function Sef(s, d: Oseba): boolean; external;

{ Funkcija vrne vrednost true, če je s posredni ali neposredni šef osebe d.
  Pri tem si pomaga s funkcijo Sef(s, d), ki pove, če je s neposredni šef osebe d.
  Pri reševanju naloge predpostavljamo, da je ta funkcija dana. }
function JeSef(s, d: Oseba): boolean;
var i: Oseba;
begin
  if Sef(s, d) then
    JeSef := true
  else begin
    JeSef := false;
    for i := 1 to n do
      if Sef(s, i) then if JeSef(i, y) then
        begin JeSef := true; exit end;
  end; {if}
end; {JeSef}

{ Funkcija vrne vrednost true, če je delavec x šef vseh delavcev iz
  množice S. V nasprotnem primeru funkcija vrne vrednost false. }
function SefVseh(x: Oseba; S: MnozicaOseb): boolean;
```

```
var i: Oseba;
begin
  SefVseh := true;
  for i := 1 to n do
    if S[i] then
      if not JeSef(x, i) then
        begin SefVseh := false; exit end;
end; {SefVseh}

{ Podprogram poišče vse skupne šefe delavcev iz množice S. }
procedure VsiSkupniSefi(var S: MnozicaOseb);
var vss: MnozicaOseb;
    i: Oseba;
begin
  for i := 1 to n do
    if not S[i] then vss[i] := SefVseh(i, S)
    else vss[i] := false;
end; {VsiSkupniSefi}

{ Podprogram izloči iz mnozice šefov vse, ki niso najbližji
  glede na relacijo JeSef. }
procedure NajblizjiSefi(var S: MnozicaOseb);
var i, j: Oseba;
begin
  for i := 1 to n do
    for j := 1 to n do
      if S[i] and S[j] and (i <> j) then
        if JeSef(j, i) then S[j] := false;
end; {NajblizjiSefi}

var S: MnozicaOseb; i: Oseba;
begin {Sefi}
  for i := 1 to n do S[i] := false;
  S[123] := true; S[456] := true;
  VsiSkupniSefi(S);
  NajblizjiSefi(S);
end. {Sefi}
```

Tip MnozicaOseb bi bil lahko tudi **set of** Oseba, vendar bi bile s tem v praksi lahko težave. Mnogi prevajalniki ne dovolijo množic nad tipi z več kot nekim določenim številom možnih vrednosti, torej pri velikih n tipa **set of** Oseba najbrž ne bi mogli uporabiti. Poleg tega ni standardnega načina za dodajanje in brisanje posameznih elementov množice (nekateri prevajalniki imajo v ta namen podprograma Include in Exclude ali kaj podobnega); zato bi morali uporabiti konstrukt oblike S := S + [x], ki pa je lahko neučinkovit (če niso bili pisci prevajalnika posebej pozorni na to možnost) — če zares skonstruira množico [x] v neki pomožni spremenljivki in nato računa unijo, bo poraba časa za to

operacijo sorazmerna z n, ne pa konstantna, kar bi pri dodajanju posameznega elementa sicer pričakovali.

Gornji program bi se dalo še izboljšati. Podprogram JeSef (ki sicer ni del naloge, ampak naloga pravi, da lahko predpostavimo, da že obstaja) lahko izvede ogromno rekurzivnih klicev za ene in iste ljudi, če so šefovski odnosi neugodno strukturirani. Na primer: a je šef b-ju in c-ju, tadva d-ju, slednji e-ju in f-ju, tadva pa g-ju; za povrhu naj bo še a šef y-u in slednji z-ju. Če zdaj kličemo JeSef(a, z), se bodo najprej izvedli rekurzivni klici za (b, z), (d, z), (e, z), (g, z), (f, z), (g, z), (c, z), (e, z), (g, z), (f, z), (g, z), šele nato pa (y, z), ki bi ugotovil, da je y (in zato tudi a) res nadrejen z-ju. Če bi imel tudi g podrejene delavce in slednji nekega skupnega podrejenega človeka, itd., bi se število klicev še povečevalo (eksponentno hitro glede na globino teh nadrejenosti). Če bi funkcija Sef dopuščala ciklične odnose (npr. x je šef y-u in obratno), pa bi se lahko celo zaciklali. Najmanj, kar bi lahko naredili, bi bila globalna tabela, v katero bi si JeSef zapisoval, katere pare delavcev je že obdelal in ali je v tistem paru res prvi šef drugega ali ne.

Še bolje bi verjetno bilo, če bi namesto podprograma JeSef(x, y) dobili podprogram, ki nam vrne vse nadrejene določenega človeka (ta bi se namesto na relacijo Sef(x, y), kakršno uporablja zdaj, verjetno opiral na neko funkcijo, ki bi za danega delavca vrnila seznam vseh njegovih neposrednih šefov). Podprogram VsiSkupniSefi bi potem le izračunal presek teh množic nadrejenih po vseh delavcih iz skupine S. Za učinkovito računanje presekov (če nimamo le 12 delavcev, ampak recimo na tisoče) bi bilo koristno, če bi bile te množice nadrejenih predstavljene z urejenimi seznami ali pa z razpršenimi tabelami. Taka predstavitev množic bi tudi omogočila, da bi šli zanki v podprogramu NajbližjiSefi le po članih množice šefov S, ne pa po vseh osebah, kar bi bil verjetno velik prihranek.

Lahko bi gledali tudi v obratno smer: za vsakega človeka bi pripravili množico vseh podrejenih in preverili, če vsebuje vse delavce iz skupine S. Tako bi ugotovili, ali je ta človek skupni šef vsem iz skupine S. Vendar pa je treba v tem primeru to ponoviti za vse ljudi (v prejšnjem odstavku pa le za vse iz S), poleg tega pa ima v tipični (drevesasti) organizaciji vsakdo le malo nadrejenih, podrejenih pa imajo nekateri ljudje zelo veliko.

R1995.3.4

N: 218

Pri pisanju programa smo si pomagali z metodo pošiljanja žetona. Računalniki si preko omrežja pošiljajo natanko en žeton, na katerem je zapisan tekoči naslov. Prvi računalnik (tisti, ki je sporočilo sprejel iz smeri nič) na žeton zapiše številko ena. Žeton potem pošlje svojemu sosedu. Ko mu sosed vrne žeton, ga pošlje naslednjemu sosedu in tako naprej. Nazadnje vrne žeton pošiljatelju.

Tudi sosedje ravnajo podobno. Vsak sosed poveča naslov na žetonu za ena in vrne žeton pošiljatelju šele potem, ko je žeton že obhodil ves okoliš. Včasih

se lahko zgodi, da se žeton vrne do pošiljatelja po krožni poti; v tem primeru ga pošiljatelj nespremenjenega vrne v smer, od koder je prispelo. Tako so na koncu označena vsa vozlišča, žeton pa je vsako povezavo prepotoval natanko dvakrat; v vsako smer enkrat.

```
program KdoSem;

  function Poslji(VSmer: integer; Sporocilo: integer); external;
  function Sprejmi(var IzSmeri: integer; var Sporocilo: integer): boolean; external;
  procedure NastaviSvojNaslov(Naslov: integer); external;

const SteviloSmeri = 11;

var
  PrvotnaSmer, Smer, StevecSmeri, Naslov: integer;
  PoznaneSmeri: array [1..SteviloSmeri] of boolean;
begin
  for StevecSmeri := 1 to SteviloSmeri do
    PoznaneSmeri[StevecSmeri] := false;
  repeat until Sprejmi(PrvotnaSmer, Naslov);
  if PrvotnaSmer = 0 then Naslov := 1
  else begin
    Naslov := Naslov + 1;
    PoznaneSmeri[PrvotnaSmer] := true;
  end; {if}
  NastaviSvojNaslov(Naslov);
  for StevecSmeri := 1 to SteviloSmeri do begin
    if not ZnanaSmer[StevecSmeri] then begin
      PoznaneSmeri[StevecSmeri] := true;
      if Poslji(StevecSmeri, Naslov) then
        repeat
          repeat until Sprejmi(Smer, Naslov);
          if Smer <> StevecSmeri then begin
            PoznaneSmeri[Smer] := true;
            Poslji(Smer, Naslov);
          end; {if}
        until Smer = StevecSmeri;
    end; {if}
  end; {for}
  Poslji(PrvotnaSmer, Naslov);
end. {KdoSem}
```

REŠITEV NALOGE PRVEGA ZAKLJUČNEGA TEKMOVANJA IZ ZNANJA RAČUNALNIŠTVA

R1995.Z

N: 219

Naš program naj bi se pognal za vsak vzorec posebej; torej je koristno, če z diska ne nalagamo celotnega slovarja, ker bomo morali to početi pri vsakem vzorcu znova in bomo s tem le zapravljali čas (sploh pa celoten slovar najbrž tako ali tako ne bi šel v pomnilnik, saj imajo besede iz datoteke `besede.txt` skupno 883 193 črk, naš program pa bo načeloma tekel v realnem načinu in lahko izkoristi največ 640 KB pomnilnika). Delo z diskom je v primerjavi z delom s podatki v pomnilniku precej počasno, zato naj naš program z diska pri vsakem vzorcu prenese čim manj podatkov. Zavedati se moramo tudi tega, da so pri delu z diskom naključni dostopi veliko dražji od zaporednih, zato je lahko neka rešitev, ki prebere več podatkov, boljša od neke take, ki jih prebere manj, vendar so ti bolj razsuti po datoteki in je potrebno zato veliko čakati, da se glava znajde na pravem mestu.

Besede bomo poskušali na disku organizirati tako, da pri iskanju bližnjih besed ne bo treba prebrati vseh, ampak le tiste, ki imajo res nekaj možnosti, da bi se izkazale kot bližnje. Tiste, ki jih bo pri delu s posameznim vzorcem vendarle treba prebrati, pa naj bodo čim bolj skupaj, ne pa razmetane naokoli po disku, tako da jih bo čim ceneje prebrati. Preprost kriterij, s katerim si lahko pomagamo, je dolžina: če je naš vzorec dolg n_0 črk, ima vsaka bližnja beseda vsaj n_0 in največ n_0+2 črk, torej bo za iskanje bližnjih besed dovolj, če preberemo le besede take dolžine. Še en koristen kriterij je število različnih črk v besedi: če ima naš vzorec m_0 različnih črk, jih imajo bližnje besede vsaj m_0 in največ m_0+2. Oba kriterija lahko tudi združimo in vidimo, da so ugodne le besede z naslednjimi lastnostmi (prvo število v paru pove dolžino besede, drugo pa število različnih črk v njej): (n_0, m_0), (n_0+1, m_0), (n_0+2, m_0), (n_0+1, m_0+1), (n_0+2, m_0+1) in (n_0+2, m_0+2). Ostale tri možnosti, (n_0, m_0+1), (n_0, m_0+2) in (n_0+1, m_0+2), odpadejo; na primer, če je beseda le za en znak daljša od našega vzorca, pa je vendarle bližnja, mora vsebovati vse črke vzorca in ima torej lahko za povrhu največ eno tako črko, ki je vzorec nima; zato ima lahko m_0 ali m_0+1, ne pa m_0+2 različnih črk.

Lahko bi torej besede v indeksni datoteki uredili tako, da pridejo za vsak par (n, m) skupaj podatki o besedah dolžine n z m različnimi črkami. Tej skupini besed recimo $W(n, m)$. Naš program bi izračunal dolžino n_0 in število različnih črk m_0 dobljenega vzorca in se potem posvetil ustreznim šestim skupinam besed, vsaki posebej (mogoče še manj kot šestim: če je na primer $n_0 = 12$, besed s 13 in 14 črkami pač ni; poleg tega pa za nekatere pare (n, m) sploh ni nobene take besede, na primer nobene z dvanajstimi enakimi črkami; v resnici je le 58 množic $W(n, m)$ nepraznih). Ko preberemo besede v pomnilnik, bi pač za vsako posebej preverili, ali je bližnja našemu vzorcu ali ne; tu se nam

z učinkovitostjo ni treba toliko ukvarjati, saj bo program veliko večino časa najbrž tako ali tako porabil za branje z diska.

Vendarle pa je po svoje škoda, da bomo brali z diska vse te besede, saj se bo večinoma izkazalo, da je med njimi le peščica bližnjih. Če bi na primer kot vzorce zapovrstjo vzeli vse besede iz danega slovarja in pri vsaki pogledali, koliko besed moramo prebrati pri zgoraj opisanem postopku, koliko pa je zares bližnjih vzorcu, bi videli, da bi naš program prebral v povprečju pri vsakem vzorcu okoli 22 837,56 besed, bližnjih pa je povprečno le 12,37.[43] Dodatna nerodnost je tudi ta, da jih bomo morali brati v več kosih, kajti za nekatere (n, m) je skupna dolžina vseh besed iz $W(n, m)$ daljša od 2^{16} in jih torej naš 16-bitni program ne bo morel prebrati v enem kosu (najdaljša je $W(8, 6)$, ki ima 8 668 besed in torej skupno 69 344 črk).

Ena možna izboljšava je ta, da bi besede predstavili v kakšni bolj jedrnati obliki, ki bi nam omogočila za čim več nebližnjih besed ugotoviti, da niso bližnje, tako da se nam z njimi ne bi bilo treba več ukvarjati. Lahko bi na primer za vsako besedo izračunali „bitno karto“ — 32-bitno število, v katerem vsak bit pove, ali je neka črka v tej besedi prisotna ali ne (potrebujemo torej le 29 bitov, trije pa bi pač ostali neizkoriščeni). Beseda vsekakor ne more biti bližnja, če v njej manjka kakšna črka, ki jo vzorec ima, take primere pa lahko preprosto ugotovimo s primerjanjem bitnih kart: če ima vzorec bitno karto `p`, beseda pa `w`, mora veljati (`p` **and** `w`) = `p`, sicer beseda gotovo ni bližnja. Ta pogoj pa seveda še ni zadosten, saj na primer ne preverja tega, ali se vsaka črka vzorca pojavlja v besedi vsaj tolikokrat kot v vzorcu.

Količino podatkov, ki jih bo treba brati, lahko še zmanjšamo. Bitne karte za posamezno skupino $W(n, m)$ (saj bomo vedno brali celo skupino naenkrat) so čisto navadna cela števila; lahko jih uredimo in si zapomnimo pri vsakem le razliko med prejšnjim številom in tem. Lepo pri tem je, da so te razlike pogosto najbrž razmeroma majhne; lahko bi recimo razlike, manjše od 128, shranili kot en byte, ostale pa kot štiri. Pri tem moramo paziti le še na to, da jih bomo lahko tudi nedvoumno prebrali (na primer tako: pri razlikah, večjih od 128, premaknimo bite od bita 7 naprej za eno mesto proti levi in bit 7 prižgimo; ko bomo to število zapisali na disk, bo imel tako prvi byte, ki hrani najnižjih osem bitov, na najvišjem bitu enico, tako da ga bo pri branju lahko ločiti od byta, ki sam po sebi predstavlja razliko, manjšo od 128; pri tistem pomiku v levo smo sicer izgubili najvišji bit, ampak saj vemo, da so najvišji trije biti pri nas tako ali tako neizkoriščeni). Izkaže se, da ta preprosta oblika

[43] Ko primerjamo različne algoritme in razmišljamo o tem, kateri je boljši, je seveda malce nerodno to, da ne vemo, na kakšni množici vzorcev bodo naš program na koncu preizkušali. Tu smo vzeli povprečja kar po množici vseh besed v slovarju, torej kot da bi za vzorce vzeli vse te besede; pri tem se zanašamo na upanje, da bo imela množica vzorcev, ki jih bodo na koncu res uporabili, v povprečju vsaj podobne lastnosti. Na primerno izbrani množici vzorcev pa bi znala biti razmerja med hitrostmi različnih algoritmov tudi precej drugačna. Več o tem gl. D. LaLoudouana, M. B. Tarare: *Data set selection*, NIPS 2002.

kompresije bitnih kart skrajša zapis za približno 36 % in za toliko se zmanjša tudi količina podatkov, ki jih moramo pri branju bitnih kart prenašati z diska.

Naš program naj bi torej najprej prebral bitne karte za ustrezni $W(n, m)$, jih primerjal z bitno karto vzorca in tako dobil seznam besed-kandidatk, ki bi utegnile biti bližnje vzorcu. Nato mora prebrati vse te kandidatke in za vsako dokončno preveriti, ali je bližnja ali ni. Tu imamo sicer načeloma pri vsaki kandidatki po en naključen dostop do diska, vendar se izkaže, da gre to branje vendarle presenetljivo hitro, če beremo ves čas po naraščajočih naslovih. Zato je koristno besede $W(n, m)$ zapisati v enakem vrstnem redu, v kakršnem smo uredili njihove bitne karte. Tako bo že iz položaja bitne karte znotraj zaporedja bitnih kart jasno tudi, na katero besedo se ta bitna karta nanaša. V nasprotnem primeru bi morali ob vsaki bitni karti hraniti še indeks besede, ki ji ta karta pripada, to pa je spet potrata prostora in bi nam povečalo količino podatkov, ki jih moramo ob poizvedbi prebrati.

Z zgoraj opisano kompresijo podatkov smo pridobili še nekaj: izkaže se, da zdaj bitne karte za (n, m), $(n, m + 1)$ in $(n, m + 2)$ tudi v najslabšem primeru ne zasedejo vse skupaj več kot 45 530 bytov prostora. Torej se nam, če imamo skupine bitnih kart na disku razporejene po naraščajočih n in pri vsakem n še po naraščajočih m, ne bo treba ukvarjati z vsakim od šestih parov (n, m) posebej. Lahko bomo prebrali najprej bitne karte za (n_0, m_0); nato z enim samim branjem še vse tiste za $(n_0 + 1, m_0)$ in $(n_0 + 1, m_0 + 1)$; in nato spet z enim samim branjem še vse tiste za $(n_0 + 2, m_0)$, $(n_0 + 2, m_0 + 1)$ in $(n_0 + 2, m_0 + 2)$. Tako si prihranimo še nekaj naključnih dostopov do diska.

Opisana rešitev z bitnimi kartami je že zelo dobra, vendar pa se da najti še boljše. Označimo z $W(n, m, a)$ množico vseh besed iz $W(n, m)$, ki vsebujejo tudi vsaj en primerek črke a. Če vsebuje tudi naš vzorec črko a, je jasno, da so možne bližnje besede tiste iz $W(n, m, a)$, ne pa tudi tiste iz množice $W(n, m) \setminus W(n, m, a)$. Če bi torej podatke o besedah na disku uredili po trojicah (n, m, a), bi bilo dovolj, če bi pri vsakem (n, m) delali le z besedami $W(n, m, a)$ za eno od črk a iz našega vzorca. (Seveda bi bilo pametno vzeti tisto črko, pri kateri bi morali z diska prebrati najmanj podatkov. To je še zlasti lepo, ker ima vzorec skoraj zagotovo vsaj kakšno razmeroma redko črko, pri kateri bo množica $W(n, m, a)$ prijetno majhna. Če smo prej kot povprečje prek poizvedb z vsemi besedami iz slovarja videli, da bi se morali ukvarjati pri vsaki poizvedbi s povprečno 22 837,54 besedami, pade zdaj to število na 3 382,85.)

Nerodno pa je to, da zdaj spada skoraj vsaka beseda v več množic $W(n, m, a)$, medtem ko je prej pripadala samo eni $W(n, m)$. Če hočemo hraniti podatke o vseh $W(n, m, a)$, bo tu skoraj vsaka beseda predstavljena na več mestih in količina podatkov na disku se nam zato precej napihne. Za branje to ni slabo, saj jih bomo morali brati manj kot prej; paziti pa moramo, da ne presežemo omejitve na 5 MB.

Če bi recimo hoteli hraniti celotne sezname besed za vse $W(n,m,a)$, tako kot smo jih prej za vse $W(n,m)$, bi videli, da so vsi skupaj zdaj dolgi 6 267 552 črk namesto dosedanjih 883 193; mi pa moramo v tistih 5 MB stlačiti še vse bitne karte! Lahko bi se odločili hraniti le sezname $W(n,m)$ kot doslej, bitne karte pa bi hranili za vsako $W(n,m,a)$ posebej. Vendar pa bi morali zdaj ob vsaki bitni karti hraniti še indeks, ki pove, katera po vrsti je v seznamu $W(n,m)$ beseda, na katero se nanaša ta bitna karta. Ker nima nobena množica $W(n,m)$ več kot 8 668 besed, bi tu zadostovala 16-bitna števila. Bitne karte bi nam tako zasedle 3 576 150 bytov prostora, tako da ga ostane dovolj še za sezname besed $W(n,m)$ (883 193 bytov) in še za knjigovodstvo. (Lahko se tudi pri indeksih odločimo za kompresijo — če se indeks pri neki bitni karti razlikuje za manj kot 128 od tistega pri prejšnji, ga shranimo le kot en byte. Tu se zanašamo na to, da bo tudi seznam $W(n,m)$, enako kot seznami bitnih kart, urejen po naraščajoči vrednosti bitne karte, zato indeksi, upajmo, ne bodo delali prevelikih skokov. Vendar se izkaže, da je ta prihranek majhen; poraba prostora za bitne karte se tako zmanjša na 3 177 443 bytov, podobno pa tudi količina podatkov, ki jih moramo v povprečju prebrati pri iskanju bližnjih besed.)

S tem, ko beremo bitne karte za manj besed, smo precej pridobili, s tem, da vsebujejo te bitne karte zdaj tudi indeks v zaporedje $W(n,m)$, pa smo nekaj izgubili, vendar je končni učinek še vedno zelo ugoden: v povprečju (čez poizvedbe z vsemi besedami iz slovarja) moramo za branje bitnih kart prebrati z diska okoli 75 % podatkov manj kot prej (pri vsaki poizvedbi povprečno 15 022 bytov namesto 58 188).

Nekaj smo izgubili tudi s tem, ker moramo pri vsakem (n,m) izbrati nek a in imamo potem en naključni dostop do diska za ta $W(n,m,a)$. Druga možnost bi bila ta, da bi bitne karte na disku zložili po naraščajočem n in nato pri vsakem n najprej po a, nato pa za vsak (n,a) še po naraščajočem m. Če nas v nekem trenutku zanimajo besede iz (n,m), $(n,m+1)$ in $(n,m+2)$, bi pogledali, kateri a ima najmanjšo vsoto $|W(n,m,a)|+|W(n,m+1,a)|+|W(n,m+2,a)|$ in potem zanj v enem zamahu prebrali bitne karte za vse tri skupine. (Poraba prostora na disku se pri tem ni nič spremenila, saj smo le drugače razmestili skupine z bitnimi kartami.) Res je sicer, da moramo brati zdaj nekaj več podatkov kot prej, ko smo si lahko za vsak m posebej izbrali najugodnejši a; vendar pa je črka, ki je redka pri enem m, najbrž kolikor toliko redka tudi pri kakšnem drugem; namesto prihranka 75 %, ki smo ga ugotovili na koncu prejšnjega odstavka, bi imeli zdaj malo manjši prihranek 65 % (povprečno bi morali za branje bitnih kart prebrati po 20 222 bytov pri vsakem vzorcu), kar še vedno ni slabo; in poleg tega imamo zdaj za branje bitnih kart le po en naključni dostop za vsak n, ne pa več po enega za vsak par (n,m).

Vendar pa smo si, odkrito povedano, vse to delo z bitnimi kartami nakopali le zato, da bi morali z diska prebrati čim manj podatkov. (Varčevanje s

procesorjevim časom je precej nepomembno, ker naš program levji delež časa čaka na disk.) Najprej prebiramo bitne karte, izluščimo kup besed-kandidatk in nato vsako od njih preberemo še posebej, da vidimo, če je res bližnja. Mar ne bi bilo bolje, če bi prebrali že takoj kar besede same? Poskusimo torej raje zmanjšati količino potrebnega branja z bolj zgoščenim zapisom besed, ne pa z bitnimi kartami.

Za začetek lahko opazimo, da je zelo potratno, če namenimo vsaki črki cel byte, saj imamo le 29 možnih črk (črke slovenske abecede in še q, w, x in y) in je torej vsaki dovolj že 5 bitov, ne pa 8. Druga koristna zamisel je, da besede uredimo po abecedi (pravzaprav smo jih dobili tako urejene že v slovarju) in pri vsaki napišemo, v koliko prvih črkah se ujema s prejšnjo, nato pa zapišemo le črke od prvega neujemanja naprej (za število skupnih črk bi zadostovali štirje biti, saj imajo besede največ 12 črk, vendar bomo zaradi preprostosti tudi tu uporabili pet bitov). Ker so urejene po abecedi, se dve sosednji gotovo pogosto začneta na nekaj skupnih črk. Izkaže se, da nam vsaka od teh dveh izboljšav posebej prihrani okoli 30 % prostora, obe skupaj pa slabih 60 %. Če bi recimo hoteli po starem prebirati vse besede iz vseh šestih možnih $W(n, m)$, bi zneslo to povprečno po 188 098 bytov pri vsakem vzorcu, zdaj pa le še 77 740.

Še bolj koristno pa je to, da lahko zdaj shranimo v indeksno datoteko vse $W(n, m, a)$; namesto prej omenjenih 6 267 552 zasedejo le še 2 808 460 bytov. Če torej pri vsakem (n, m), ki nas zanima, izberemo za a tisto črko vzorca, ki bo zahtevala branje najmanj podatkov, in potem preberemo $W(n, m, a)$, bomo pri povprečni poizvedbi za branje teh besed porabili skupno le 12 907 bytov. Če pa pri vsakem n izberemo le en a, ki bo dal najkrajšo skupno dolžino zaporedij $W(n, m, a)$, $W(n, m + 1, a)$ in mogoče še $W(n, m + 2, a)$ (kolikor jih pač potrebujemo), bomo morali pri povprečni poizvedbi prebrati 17 175 bytov (imamo pa zato manj naključnih dostopov, namreč po enega za vsak n namesto po enega za vsak (n, m)). Vidimo lahko, da je zdaj količina podatkov, ki jih moramo prebrati, že manjša kot tista pri branju bitnih kart, pri čemer smo morali tam kasneje prebirati še besede kandidatke, česar zdaj ni več.

Razmislimo malo še o knjigovodskih podatkih. Pri rešitvi, do katere smo prišli na koncu, zadostuje, če na začetku indeksne datoteke zapišemo, koliko 5-bitnih znakov potrebuje posamezno zaporedje $W(n, m, a)$. To bi bila lahko tabela $12 \times 12 \times 29$ šestnajstbitnih števil. To pomeni 8 352 bytov, ki jih moramo prebrati na začetku vsake poizvedbe; mogoče bi lahko še kaj prihranili, če bi jih hranili kako bolj zgoščeno, ker je veliko seznamov praznih in so v tej tabeli zato tam ničle (na primer pri $m > n$, pa tudi pri kakšnih redkejših črkah). Rešitev z bitnimi kartami bi zahtevala podatke o velikosti množic $W(n, m, a)$ in še množic $W(n, m)$, saj imamo besede shranjene le v slednjih. Če ne delamo z množicami $W(n, m, a)$, je dovolj že tabela z velikostmi množic $W(n, m)$.

Za preverjanje, ali je neka beseda res bližnja danemu vzorcu, lahko uporabimo naslednji pogoj: beseda mora pripadati eni od šestih skupin $W(n, m)$

(torej za $n_0 \le n \le n_0 + 2$ in $m_0 \le m \le m_0 + (n - n_0)$) in vsake črke, ki se pojavlja v vzorcu, mora vsebovati vsaj toliko izvodov kot vzorec, vendar največ dva izvoda več kot vzorec. Ta pogoj je potreben: če je neka beseda bližnja, gotovo spada v eno od tistih šestih skupin $W(n, m)$ (o tem smo razmislili že zgoraj) in gotovo tudi ne vsebuje nobene črke manjkrat kot vzorec, niti ne v treh ali več izvodih več kot vzorec, saj vse to sledi že takoj iz definicije bližnjih besed. Naš pogoj pa je tudi zadosten: če neka beseda ustreza temu pogoju, očitno vsebuje vse črke vzorca v vsaj toliko izvodih kot vzorec, obenem pa že iz omejitve na dolžino ($n \le n_0 + 2$) sledi, da ne more vsebovati več kot dveh dodatnih črk, torej je gotovo bližnja vzorcu.

Za konec lahko opozorimo še na en pomemben vir dostopov do diska, o katerem doslej nismo razmišljali. To je izvršljiva (`.exe`) datoteka našega programa. Ker se požene naš program za vsak vzorec posebej, računalnik pa nima diskovnega predpomnilnika, mora pred obdelavo vsakega vzorca ponovno prebrati to datoteko; torej, če je ta predolga, je lahko to že kar pomemben strošek, saj naš program vendarle ne bo bral tako veliko podatkov (recimo 20 KB za besede in še prej 10 KB za knjigovodske podatke). Po naših opažanjih se Turbo Pascal tu odreže precej bolje kot Turbo C, saj nam je slednji vztrajno pripravljal izvršljive datoteke, dolge čez 40 KB, Turbo Pascal pa je ostal pri dobrih 10 KB (oz. 6 KB, če izrežemo dele programa za tvorbo indeksa).

```
program Isci;

{ Skupne deklaracije }

const
  MaxDolz = 12;
  StCrk = 29;
  Crke: string = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ^[@';
  ImeIndeksa = 'index.dat';

type
  Beseda = string[MaxDolz];
  Glava = array [1..MaxDolz, 1..MaxDolz, 1..StCrk] of integer;
  SeznamCrk = array [1..MaxDolz] of 1..StCrk;

{ Vrne število različnih črk v besedi W. V sc vrne seznam teh črk. }
function RazlicneCrke(const W: Beseda; var sc: SeznamCrk): integer;
var i, j, M: integer;
begin
  M := 0;
  for i := 1 to Length(W) do begin
    j := 1; while (j < i) and (W[j] <> W[i]) do j := j + 1;
    if j = i then begin M := M + 1; sc[M] := Pos(W[i], Crke) end;
  end; {for}
  RazlicneCrke := M;
```

```
end; {RazlicneCrke}

{ Priprava indeksa }

type
  TextBuf = array [0..4095] of byte;
  BufText = record { za hitrejše delo s tekstovnimi datotekami }
    T: text; Buf: ↑TextBuf;
  end;

procedure Odpri(var ob: BufText; const ImeDat: string; Novo: boolean);
begin
  Assign(ob.T, ImeDat); New(ob.Buf);
  SetTextBuf(ob.T, ob.Buf↑);
  if Novo then Rewrite(ob.T) else Reset(ob.T);
end; {Odpri}

procedure Zapri(var ob: BufText);
  begin Close(ob.T); Dispose(ob.Buf) end;

function ToStr(N: integer): string;
  var S: string; begin Str(N, S); ToStr := S end;

{ Za vsak (n, m) pripravi po eno datoteko z vsemi besedami iz W(n, m). }
procedure RazdeliWnm;
var T: BufText; W: Beseda; N, M: integer; sc: SeznamCrk;
    Tnm: array [1..MaxDolz] of BufText;
begin
  for N := 1 to MaxDolz do begin
    for M := 1 to N do
      Odpri(Tnm[M], 'Wnm' + ToStr(N) + '-' + ToStr(M) + '.txt', true);
    Odpri(T, 'besede.txt', false);
    while not Eof(T.T) do begin
      ReadLn(T.T, W);
      if Length(W) = N then
        WriteLn(Tnm[RazlicneCrke(W, sc)].T, W);
    end; {while}
    Zapri(T); for M := 1 to N do Zapri(Tnm[M]);
  end; {for N}
end; {RazdeliWnm}

{ Pripravi po eno datoteko za vsak (n, a); v njej so besede iz W(n, m, a) za vse m,
  urejene po naraščajočih m. V G[n, m, a] vpiše število besed v množici W(n, m, a).
  Datoteke s seznami W(n, m) na koncu pobriše. }
procedure PripraviWna(N: integer; var G: Glava);
var M, A: integer; T, Ta: BufText; W: Beseda;
begin
  for M := 1 to N do begin
```

```
      for A := 1 to StCrk do begin
        G[N, M, A] := 0;
        Odpri(Ta, 'Wna' + ToStr(N) + '-' + ToStr(A) + '.txt', M = 1);
        if M > 1 then Append(Ta.T);
        Odpri(T, 'Wnm' + ToStr(N) + '-' + ToStr(M) + '.txt', false);
        while not Eof(T.T) do begin
          ReadLn(T.T, W);
          if Pos(Crke[A], W) > 0 then
            begin WriteLn(Ta.T, W); G[N, M, A] := G[N, M, A] + 1 end;
        end; {for A}
        Erase(T.T);
      end; {for M}
end; {PripraviWna}

{ Doda vsebino vseh W(n, m, a) v indeksno datoteko. Vsaka od teh skupin je
  komprimirana posebej. V G[N, M, A] vpiše število 5-bitnih znakov v
  komprimiranem zapisu seznama W(n, m, a). Pomožne datoteke na koncu pobriše. }
procedure DodajWna(var F: file; N, A: integer; var G: Glava);
const BufLen = 4095;
var Buf: array [0..BufLen - 1] of byte; BufPos, Bit, D: integer;

    procedure Zapisi; { Zapiše vsebino bloka Buf na disk. }
    begin
      if (Bit > 0) and (BufPos < BufLen) then BufPos := BufPos + 1;
      BlockWrite(F, Buf, BufPos); BufPos := 0;
    end; {Zapisi}

    procedure Dodaj5Bitov(X: integer);
    begin
      Buf[BufPos] := (Buf[BufPos] or (X shl Bit)) and 255;
      Bit := Bit + 5; D := D + 1;
      if Bit >= 8 then begin { Nekaj bitov zapišimo v naslednji byte. }
        Bit := Bit - 8; BufPos := BufPos + 1;
        if BufPos = BufLen then Zapisi;
        Buf[BufPos] := X shr (5 - Bit);
      end; {if}
    end; {Dodaj5Bitov}

var M, i, j: integer; T: BufText; W, Wp: Beseda;
begin {DodajWna}
    Odpri(T, 'Wna' + ToStr(N) + '-' + ToStr(A) + '.txt', false);
    for M := 1 to N do begin
      BufPos := 0; Bit := 0; D := 0; Buf[BufPos] := 0; W := '';
      for i := 1 to G[N, M, A] do begin
        Wp := W; ReadLn(T.T, W);
```

```
        j := 1; if i > 1 then while (j < N) and (Wp[j] = W[j]) do j := j + 1;
        Dodaj5Bitov(j − 1); { Skupni začetek s prejšnjo besedo. }
        { Shrani še preostanek besede. }
        while j <= N do begin Dodaj5Bitov(Pos(W[j], Crke)); j := j + 1 end;
      end; {for i}
      Zapisi; G[N, M, A] := D;
    end; {for M}
    Zapri(T); Erase(T.T);
end; {DodajWna}

{ Na podlagi datoteke besede.txt pripravi indeksno datoteko,
  ki jo bomo uporabljali pri odgovarjanju na poizvedbe. }
procedure PripraviIndeks;
var N, A: integer; G: Glava; F: file;
begin
    RazdeliWnm;
    for N := 1 to MaxDolz do PripraviWna(N, G);
    Assign(F, ImeIndeksa); Rewrite(F, 1);
    BlockWrite(F, G, SizeOf(G));
    for N := 1 to MaxDolz do for A := 1 to StCrk do DodajWna(F, N, A, G);
    Seek(F, 0); BlockWrite(F, G, SizeOf(G));
    Close(F);
end; {PripraviIndeks}

{ Odgovarjanje na poizvedbe }

type
    Pojavitve = array [1..StCrk] of integer;
    Buffer = array [0..31807] of byte;
    Odmiki = array [1..MaxDolz, 1..MaxDolz, 1..StCrk] of longint;

{ Koliko bytov potrebujemo za zapis N 5-bitnih simbolov? }
function PakDolz(N: longint): integer;
    begin PakDolz := (N * 5 + 7) div 8 end;

{ Za vsako trojico (n, m, a) izračuna, na katerem odmiku v
  datoteki se zacnejo podatki za W(n, m, a). }
procedure IzracunajOdmike(const G: Glava; var O: Odmiki);
var N, M, A: integer; OO: longint;
begin
    OO := SizeOf(G);
    for N := 1 to MaxDolz do for A := 1 to StCrk do for M := 1 to N do
      begin O[N, M, A] := OO; OO := OO + PakDolz(G[N, M, A]) end;
end; {IzracunajOdmike}

procedure PoisciBliznje(const P: Beseda);
var Buf: ↑Buffer; BufPos, Bit: integer; { Blok z besedami v komprimirani obliki. }
    Znakov: integer; { Število še neprebranih 5-bitnih znakov v bloku Buf. }
```

```
  { Vrne naslednjih 5 bitov iz bloka Buf. Poveča Bit in BufPos, zmanjša Znakov. }
  function Beri5Bitov: integer;
  var X: integer;
  begin
    X := (Buf↑[BufPos] shr Bit) and 31; Bit := Bit + 5;
    if Bit >= 8 then begin { Nekaj bitov vzemimo še iz naslednjega byta. }
      Bit := Bit − 8; BufPos := BufPos + 1;
      X := (X or (Buf↑[BufPos] shl (5 − Bit))) and 31;
    end; {if}
    Beri5Bitov := X; Znakov := Znakov − 1;
  end; {Beri5Bitov}

var
  F: file; G: ↑Glava; O: ↑Odmiki;
  M: integer; PojP: Pojavitve; CrkeP: SeznamCrk; { Lastnosti vzorca P. }

  { Pregleda besede iz množice W(NN, MM, A), ki se morajo nahajati
    v bloku Buf od indeksa BufPos naprej. Če je LeCePrva = true,
    se vsaka beseda izpiše le, če ne vsebuje nobene črke, manjše od A.
    To se uporablja, če je vzorec prazen niz in hočemo v resnici
    izpisati vse besede iz W(NN, MM), vsako natanko enkrat. }
  procedure PreglejWnma(NN, MM, A: integer; LeCePrva: boolean);
  var i: integer; PojW: Pojavitve; Bliznja: boolean; CrkeW: SeznamCrk; W: Beseda;
  begin
    Znakov := G↑[NN, MM, A]; Bit := 0;
    while Znakov > 0 do begin
      i := Beri5Bitov + 1; { Skupni začetek s prejšnjo besedo. }
      Bliznja := true;
      while i <= NN do begin { Preberimo preostanek besede. }
        CrkeW[i] := Beri5Bitov;
        if LeCePrva and (CrkeW[i] < A) then Bliznja := false;
        i := i + 1;
      end; {while}
      if not Bliznja then continue; { Vsebuje črko, manjšo od A. }

      { Preverimo, če je res bližnja. }
      for i := 1 to M do PojW[CrkeP[i]] := PojP[CrkeP[i]];
      for i := 1 to NN do PojW[CrkeW[i]] := PojW[CrkeW[i]] − 1;
      i := 1; while (i <= M) and Bliznja do
        if (PojW[CrkeP[i]] < −2) or (PojW[CrkeP[i]] > 0)
        then Bliznja := false else i := i + 1;

      if Bliznja then begin { Če je bližnja, jo izpišimo. }
        W[0] := Chr(NN);
        for i := 1 to NN do W[i] := Crke[CrkeW[i]];
        WriteLn(W);
      end; {if Bliznja}
    end; {while}
```

```
    if Bit > 0 then BufPos := BufPos + 1;
  end; {PreglejWnma}

var N, NN, A, i, j, KA: integer; D, KD: longint;
begin
  Assign(F, ImeIndeksa); Reset(F, 1); New(G); New(O);
  BlockRead(F, G↑, SizeOf(G↑)); IzracunajOdmike(G↑, O↑);
  N := Length(P); M := RazlicneCrke(P, CrkeP); New(Buf);

  if N = 0 then begin { Prazen vzorec obravnavajmo posebej. }
    { Izpisati bomo morali vse besede dolžine 1 in 2. }
    D := 0; for A := 1 to StCrk do D := D + PakDolz(G↑[1, 1, A]) +
      PakDolz(G↑[2, 1, A]) + PakDolz(G↑[2, 2, A]);
    Seek(F, O↑[1, 1, 1]); BlockRead(F, Buf↑, D); BufPos := 0;
    for j := 1 to 2 do for A := 1 to StCrk do for i := 1 to j do
      PreglejWnma(j, i, A, True);
    exit;
  end; {if prazen vzorec}

  for A := 1 to StCrk do PojP[A] := 0;
  for i := 1 to N do PojP[Pos(P[i], Crke)] := PojP[Pos(P[i], Crke)] + 1;
  for j := 0 to 2 do if N + j <= MaxDolz then begin
    NN := N + j;
    { Zdaj se bomo ukvarjali z besedami iz W(NN, MM) za NN = N + j,
      M ≤ MM ≤ M + j. Naj bo A tista črka, pri kateri bomo morali
      prebrati najmanj podatkov. }
    for i := 1 to M do begin
      KA := CrkeP[i]; KD := PakDolz(G↑[NN, M, KA]);
      if j > 0 then KD := KD + PakDolz(G↑[NN, M + 1, KA]);
      if j > 1 then KD := KD + PakDolz(G↑[NN, M + 2, KA]);
      if (i = 1) or (KD < D) then begin A := KA; D := KD end
    end; {for i}
    if D <= 0 then continue;
    Seek(F, O↑[NN, M, A]); BlockRead(F, Buf↑, D); BufPos := 0;
    for i := 0 to j do PreglejWnma(NN, M + i, A, false);
  end; {for, if}
  Dispose(Buf); Close(F); Dispose(G); Dispose(O);
end; {PoisciBliznje}

begin {Isci}
  if ParamCount < 1 then PripraviIndeks
  else if Length(ParamStr(1)) <= MaxDolz then PoisciBliznje(ParamStr(1));
end. {Isci}
```

Program smo preizkusili na množici skupno 56 vzorcev, ki so jih uporabljali tudi na tekmovanju leta 1995 (ali pri testiranju ali pa kot primer za pomoč tekmovalcem); le računalnik je imel malo hitrejši procesor (P166). Zgoraj prikazani program potrebuje povprečno okoli 6,9 s, da odgovori na vse poizvedbe.

Različica, ki zloži množice $W(n, m, a)$ po n, nato po m in šele nato po a (in lahko zato za vsak (n, m) izbere drug a ter zato prebere z diska malo manj podatkov, vendar pa ima zato po en dostop do diska za vsak (n, m) in ne le po enega za vsak n), porabi na isti množici vzorcev povprečno 8,1 s. Malo preprostejša rešitev, ki prebere komprimirane bitne karte za vse besede iz primernih $W(n, m)$, se ob njih odloči, katere besede bi bilo res treba pregledati, in nato prebere te (vsako z enim naključnim dostopom; celoten slovar ima v nekomprimirani obliki) ter izpiše zadetke, porabi 9,5 s. Naivna rešitev iz besedila naloge pa porabi 133 s časa. Izkaže se tudi, da ima na čas izvajanja teh rešitev zelo velik vpliv fragmentiranost diska. Da bi se pri merjenju časa izvajanja izognili vplivu pisanja rezultatov na disk, kar sicer zahteva naša naloga, so vsi ti programi pisali rezultate kar na zaslon.

20. državno tekmovanje v znanju računalništva (1996)

NALOGE ZA PRVO SKUPINO

1996.1.1

Nekega dne je programer vstal z napačno nogo in napisal naslednji program. **Kaj izpiše program** in kako bi ga popravil, da bi to izpisal hitreje? R: 271

```
program SlepaKuraZrnoNajde;
const
  MaxT = 5;
var
  t: array [1..MaxT] of integer;
  i, j: integer;
begin
  for i := 1 to MaxT do t[i] := MaxT − i + 1;
  repeat
    i := Random(MaxT − 1) + 1;    { 1 ≤ i ≤ MaxT − 1 }
    j := t[i]; t[i] := t[i + 1]; t[i + 1] := j;
    i := 1;
    while (i < MaxT − 1) and (t[i] <= t[i+1]) do i := i + 1;
  until t[i] <= t[i + 1];
  for i := 1 to MaxT do WriteLn(t[i]);
end. {SlepaKuraZrnoNajde}
```

1996.1.2

Opiši postopek, ki preveri, če sta dve ogrlici enaki. Ogrlica je sestavljena iz N kroglic, ki so nanizane ena za drugo. Ogrlico predstavimo s tabelo znakov, kjer znaki predstavljajo barve kroglic. R: 272

```
type Ogrlica = array [1..N] of char;
```

Ker so ogrlice krožne, moramo tako obravnavati tudi tabelo. Primeri ogrlic:

`'1234567890'`	je enaka	`'7890123456'`
`'1234567890'`	ni enaka	`'1234567809'`

Opišite postopek, ki na vhodu dobi dve ogrlici, na izhodu pa vrne vrednost `true`, če sta ogrlici enaki, in vrednost `false`, če nista.

1996.1.3

Računalnikar Janez Novak je na očetovem računalniku pisal domačo nalogo. Oče je nalogo našel in vanjo napisal svoje pripombe. K sreči je oče svoje pripombe dal med zlomljene oklepaje („<“ in „>“). Ker je naloge precej, se je Janez odločil, da očetovih komentarjev ne bo brisal ročno (peš), temveč bo za brisanje napisal program. R: 273

Napiši program, ki bo iz poljubnega besedila pobrisal vse, kar se nahaja med znakoma „<“ in „>“, in besedilo spet izpisal.

Primer: Stavek

```
Danes je lepo, sončno vreme <kaj pa še,
danes dežuje> in ptički pojejo.
```

naj prevede v

```
Danes je lepo, sončno vreme  in ptički pojejo.
```

(Pozor: med `vreme` in `in` sta dva presledka.)

R: 274

1996.1.4

Hekerji so vsepovsod naokoli. Vdirajo v zasebnost, berejo in spreminjajo zaupna sporočila, zato je potrebno sporočila zakodirati. Ker želimo zagotoviti kar največjo varnost, smo se odločili, da bomo podatke kodirali s ključem, ki bo primerno velik. Ker pa računalniku ne moremo zaupati, da bo s funkcijo naključja (Random) izbral dovolj dober ključ, smo se odločili, da bomo naključni ključ vnesli sami. Pri tem nam pomaga ugotovitev, da je čas med dvema pritiskoma tipk pri tipkanju zelo naključen.

Napiši algoritem za vnos N-bitnega ključa, ki ga predstavimo z zaporedjem časov med dvema sosednjima pritiskoma na tipko. Na razpolago imaš naslednje funkcije (privzemi, da je N konstanta):

function Timer: integer; — Vrne trenutni čas v 1/1000 (tisočinkah) sekunde.

function KeyPressed: boolean; — Vrne true, če je uporabnik pritisnil kakšno tipko, sicer pa false.

function ReadKey: char; — Prebere pritisnjeno tipko. Če ni pritisnjena nobena tipka, funkcija počaka do naslednjega pritiska.

NALOGE ZA DRUGO SKUPINO

R: 275

1996.2.1

Zgodovinarji so v biltenu *20. državnega tekmovanja v znanju računalništva za srednješolce* iz leta 1996 med nalogami iz prve skupine odkrili naslednji program:

```
program Najden;

function xx(x: integer): integer;
const b = 10;
var y: integer;
begin
  y := 0;
  while x > 0 do begin
```

```
    y := y * b + x mod b;
    x := (x − y mod b) div b;
  end;
  xx := y;
end;

var x: integer;
begin
  for x := 1 to 1000 do
    if xx(x + xx(x)) = xx(x) + x then WriteLn(x);
end.
```

Zaradi skrajne neresnosti komisije se je besedilo naloge izgubilo, tako da zdaj nesrečni zgodovinarji ne vedo, **kakšen je** pravzaprav **problem**, ki so ga tekmovalci reševali. Jim lahko pomagaš?

1996.2.2

R: 276

Pri podjetju Telekom bi med mestoma Ljubljana in Amsterdam radi položili optični kabel za prenos podatkov. Ker se svetlobni signal v optičnem kablu hitro izgubi, ga je potrebno ojačati na vsake štiri kilometre, lahko pa ga ojačamo tudi pogosteje. Optični ojačevalniki so zelo dragi, zato so se odločili, da bodo kabel položili tako, da bodo ojačevalniki nameščeni na razdalji natanko štiri kilometre.

Pri tem pa so naleteli na majhen problem. Kanal, po katerem bo potekal optični kabel, prečka nekatera kratka območja, kot so na primer reke, ceste, mostovi in predori, na katerih se ojačevalnikov ne da namestiti. Podatke o ovirah so vnesli v računalnik, s programom Postavi pa bodo izbrali lokacijo prvega ojačevalnika na razdalji, manjši od štirih kilometrov, in to tako, da noben izmed preostalih ojačevalnikov ne bo pristal na oviri. Lahko se zgodi, da ojačevalnikov ni mogoče postaviti na zahtevani način. V tem primeru naj program to sporoči.

Napiši program Postavi. Pri tem si pomagaj s podatki, zbranimi v naslednjih tabelah:

- **var** Zacetek: **array** [1..StOvir] **of** integer;
  Razdalja med začetkom kabla in začetkom ovire (v metrih).
- **var** Konec: **array** [1..StOvir] **of** integer;
  Razdalja med začetkom kabla in koncem ovire (v metrih).

StOvir je torej število vseh ovir. Ovira z začetkom 100 in koncem 200 pomeni, da ojačevalnika ne smemo postaviti $100, 101, \ldots, 199$ m od začetka kabla, lahko pa ga postavimo 200 m od začetka kabla. Predpostaviš lahko, da je Zacetek[i] $<$ Konec[i] $<$ Zacetek[i] $+ 4000$.

R: 278

1996.2.3

Gospa Bogataj je navdušena zbirateljica bisernih ogrlic. Za obisk računalniškega tekmovanja si je izbrala pisano ogrlico z rdečimi, modrimi in brezbarvnimi biseri.

Zjutraj jo je na poti opazil ropar Dolgoprst, sicer star znanec policije, ki se mu zdi stara ženica lahek plen. Odločil se je, da bo ogrlico pretrgal, potem pa z vsakega konca ogrlice snel nekaj biserov. Ogrlico bo pretrgal tako, da mu bodo v vsaki roki ostali samo biseri ene barve in brezbarvni (modri in brezbarvni ali rdeči in brezbarvni).

Napišite program, ki bo Dolgoprstu pomagal pretrgati ogrlico tako, da mu bo v vsaki roki ostalo čimveč biserov. Ogrlica je podana kot niz znakov (`m`, `b`, `r`). Pri tem pazite na to, da se desni konec niza krožno nadaljuje od začetka naprej in obratno.

Primer ogrlice s 45 biseri:

```
  5×   2× 3×   4×     6×      5×    3×    5×   2× 4×     6×
mmmmm bb rrr bbbb rrrrrr bbbbb rrr mmmmm rr bbbb mmmmmm
         ↑
```

Če ogrlico pretrga na mestu, označenem s ↑ (med sedmim in osmim biserom, torej za `bb` in pred `rrr`), mu ostane v levi roki $2\mathtt{b} + 5\mathtt{m} + 6\mathtt{m} + 4\mathtt{b} = 17$ biserov, v desni pa $3\mathtt{r} + 4\mathtt{b} + 6\mathtt{r} + 5\mathtt{b} + 3\mathtt{r} = 21$ biserov. Ukradenih biserov je torej 38, kar je pri tej ogrlici tudi najboljši možni izkupiček.

R: 280

1996.2.4

Na računalniku, priključenem na Internet, opažamo povečano število poskusov vdorov. Na voljo imamo program, s katerim lahko posameznim računalnikom ali pa celi podmreži (delu omrežja) omejimo dostop do našega računalnika. Radi bi napisali še program, ki bi sproti popravljal spisek računalnikov in mrež, katerim dostop ni dovoljen.

Vsak računalnik v omrežju ima svoj naslov, ki je sestavljen iz dveh številk: prva določa podmrežo, druga pa računalnik v okviru podmreže. Odločili smo se za naslednji postopek omejevanja dostopa:

- vsakič, ko pride do poskusa vdora, si zapomnimo naslov računalnika in čas poskusa;
- če je poskus iz iste podmreže kot nek drug poskus, ki je že v tabeli, si zapomnimo kar celo podmrežo (naslov podmreže ponazorimo tako, da številko računalnika v naslovu nadomestimo z 0 — številke računalnikov v mreži se sicer začnejo z 1);
- po nekem določenem času od zadnjega poskusa vdora z nekega računalnika ali podmreže, ki ga označimo s TTL (*time-to-live* ali življenjska doba omejitve), dostop s tega računalnika oz. podmreže spet omogočimo.

Naslove in čase shranjujemo v urejen seznam. Pri upravljanju s seznamom si pomagamo z naslednjimi deklaracijami in podprogrami:

const TTL = 1800;

type Naslov = **array** [1..2] **of** integer;

function Cas: integer; — Vrne trenutni čas v sekundah.

procedure Dodaj(N: Naslov; t: integer); — V tabelo doda naslov N in čas t.

procedure Brisi(N: Naslov); — Iz tabele izbriše naslov N.

procedure Obnovi(N: Naslov; t: integer); — V tabeli spremeni čas t, ki je zapisan ob naslovu N.

function Poisci(N: Naslov): integer; — Če v tabeli obstaja naslov N, vrne pripadajoči čas, sicer vrne 0.

function Naslednji(**var** N: Naslov; **var** t: integer): boolean; —
V tabeli poišče naslednji naslov po vrsti (prvega večjega od naslova N). Funkcija vrne true, če tak naslov obstaja, sicer vrne false.

Napiši podprogram Vsiljivec, ki ga bo sistem poklical vsakič, ko bo opazil poskus vdora. Naslov vdiralca dobi podprogram Vsiljivec kot parameter. Napiši tudi podprogram Sprosti, ki ga bo sistem občasno izvedel in z njim iz tabele odstranil zastarele zapise.

NALOGE ZA TRETJO SKUPINO

1996.3.1

Dva računalnika si prek omrežja pošiljata in izpisujeta tekstovna sporočila, ki jih vtipkavajo uporabniki. Na obeh računalnikih se sočasno izvaja naslednji program:

R: 281

```
var SprejemVkljucen: boolean;

procedure PosljiSporocilo(S: string);
var Odg: string;
begin
  Poslji('PošiljalBi');
  repeat Sprejmi(Odg) until Odg <> '';
  if Odg = 'RajeNe!' then WriteLn('Kolega ima izključen sprejem.')
  else if Odg = 'KarDaj!' then Poslji(S);
end; {PosljiSporocilo}

procedure SprejmiSporocilo;
var S: string;
begin
  if SprejemVkljucen then begin
    Poslji('KarDaj!');
```

```
    repeat Sprejmi(S) until S <> '';
    WriteLn(S);
  end
  else Poslji('RajeNe!')
end; {SprejmiSporocilo}

begin
  SprejemVkljucen := true;
  repeat
    Sprejmi(S);
    if S = 'PošiljalBi' then SprejmiSporocilo;
    if KeyPressed then begin
      WriteLn('Vtipkaj sporočilo: ');
      ReadLn(S);
      if S = '*' then SprejemVkljucen := not SprejemVkljucen
      else if S <> '' then PosljiSporocilo(s)
    end;
  until false;
end.
```

Zunanji podprogram Poslji(**var** S: string) pošlje sporočilo drugemu računalniku. Sprejeta sporočila se shranjujejo v predpomnilnik. Prebiramo jih s podprogramom Sprejmi(**var** S: string). Če je predpomnilnik prazen, Sprejmi vrne prazen niz. Funkcija KeyPressed: boolean vrne true, ko uporabnik pritisne kako tipko.

Razloži, kako deluje gornji program! V katerem primeru program ne deluje (skiciraj časovni potek „črnega scenarija“)? Kako bi ga popravil?

R: 284

1996.3.2

Firma Passive Fools® je kupila superračunalnik in uporabnikom prodaja njegovo uporabo. Vsak uporabniški program Prog(Data) ima neke vhodne podatke Data in izpiše svoje rezultate v posebno datoteko.

Superračunalnik ima omejeno kapaciteto diskov, zato je pomembno, da je izpis vsakega uporabniškega programa končen, ker bi sicer izpisi enega programa zasedli celoten disk. Sistemski programer Lisjak B.™ je zato napisal program KoncenIzpis(Prog, Data), ki za poljuben program Prog in podatke Data ugotovi, če je izpis programa Prog pri vhodnih podatkih Data končen. Če je izpis programa Prog pri vhodnih podatkih Data končen, vrne vrednost true, sicer pa false (program Prog se pri podatkih Data zacikla, pri tem pa še vedno izpisuje svoje rezultate).

Uporabniki superračunalnika so praviloma strokovnjaki le na svojih področjih; pogosto se zgodi, da se njihovi programi zaciklajo ali pa sploh ničesar ne izpišejo. Taki programi zgolj kradejo računalnikov čas in jih ni smiselno izvajati.

Pomagaj sistemcu Lisjaku B.™ **napisati program**, ki zna za poljuben program Prog in podatke Data ugotoviti (dve nalogi):

1. ali program Prog pri vhodnih podatkih Data sploh kaj izpiše; in

2. ali se program Prog pri vhodnih podatkih Data ustavi ali zacikla.

Pri tem lahko tvoji programi kličejo poljubne programe, ki jih napišeš sam, pa tudi program KoncenIzpis(Prog, Data). Uporabljajo lahko vgrajeno proceduro Execute(Prog, Data), ki izvede program Prog na podatkih Data.

Primer uporabe programa KoncenIzpis: spodaj podani program Test ugotovi, če je prebrano število praštevilo. *Opomba:* predpostavimo, da je Random(n) nek generator psevdonaključnih celih števil, ki prej ali slej poljubno mnogokrat vrne vsako celo število med 1 in n.

```
program Delitelji(St);
begin
  while true do
    if St mod Random(St − 2) + 1 = 0 then
      WriteLn('Število ', St, ' ima delitelja med 2 in ', St − 1);
end. {Delitelji}

program Test;
begin
  ReadLn(Stevilo);
  if KoncenIzpis(Delitelji, Stevilo) then
    WriteLn(Stevilo, ' je praštevilo!')
  else
    WriteLn(Stevilo, ' ni praštevilo!');
end. {Test}
```

1996.3.3

Napiši algoritem za usklajevanje dane skupine omejitev (relacij). V omejitvah nastopajo celoštevilske spremenljivke in na začetku poznamo za vsako spremenljivko neko začetno zalogo možnih vrednosti. Za posamezno omejitev, v kateri nastopajo recimo spremenljivke $X_1, \ldots, X_n$, pravimo, da je usklajena (konsistentna), če za vsak $i = 1, \ldots, n$ in za vsako možno vrednost x_i iz trenutne zaloge vrednosti spremenljivke X_i obstaja nek tak nabor vrednosti $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ iz trenutnih zalog vrednosti spremenljivk $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n$, da skupaj z vrednostjo $X_i = x_i$ ustrezajo opazovani omejitvi. Tvoj algoritem naj (če je to potrebno) zaloge vrednosti vseh spremenljivk zmanjša, tako da bodo vse omejitve v sistemu konsistentne. (Lahko se tudi izkaže, da to sploh ni mogoče.)

R: 286

Dva primera:

$X = Y + Z$, $Y = Z + 1$, $Z > 10$

Recimo, da ima na začetku vsaka spremenljivka kot zalogo vrednosti vsa pozitivna cela števila. Po usklajevanju omejitev ostanejo zaloge vrednosti

$X \in \{23, 24, \ldots\}$, $Y \in \{12, 13, \ldots\}$, $Z \in \{11, 12, \ldots\}$.[44]

$X = 3$, $X < Y$, $Z = 3$, $Y < Z$

Tega sistema omejitev ni mogoče uskladiti — zaloga vrednosti ene izmed spremenljivk se izprazni (pa kakršne koli so že bile prvotne zaloge vrednosti).

Pri pisanju algoritma si pomagajte z naslednjimi podprogrami in funkcijami:

- PreberiOmejitev — Funkcija vrne prebrano omejitev.
- UskladiOmejitev(Omejitev) — Podprogram uskladi zalogo vrednosti spremenljivkam, ki nastopajo v omejitvi Omejitev. To pomeni, da zaloge vrednosti teh spremenljivk, če je potrebno, zmanjša tako, da omejitev postane usklajena (po definiciji usklajenosti, ki smo jo navedli na začetku naloge).

  Primer 1: Če ima spremenljivka X zalogo vrednosti $X \in \{1, \ldots, 10\}$, ima po usklajevanju omejitve $X = 3$ zalogo vrednosti $X \in \{3\}$.

  Primer 2: Pred usklajevanjem imata spremenljivki X in Y zalogi vrednosti $X \in \{5, \ldots, 15\}$ in $Y \in \{1, \ldots, 10\}$. Po usklajevanju omejitve $X < Y$ sta zalogi vrednosti $X \in \{5, \ldots, 9\}$ in $Y \in \{6, \ldots, 10\}$.
- ŠteviloSpr(Omejitev) — Funkcija vrne število spremenljivk v omejitvi Omejitev.
- ImeSpr(Omejitev, Spr) — Funkcija vrne ime spremenljivke z indeksom Spr v omejitvi Omejitev.
- ZmanjšanaZalogaVrednosti(Omejitev, Spr) — Funkcija vrne true, če je bila zaloga vrednosti spremenljivke z indeksom Spr v omejitvi Omejitev zmanjšana (po uporabi podprograma UskladiOmejitev), sicer vrne false.
- PraznaZalogaVrednosti(Omejitev, Spr) — Funkcija vrne true, če ima spremenljivka z indeksom Spr v omejitvi Omejitev prazno zalogo vrednosti (po uporabi podprograma UskladiOmejitev), sicer vrne false.

[44] Tukaj tudi vidimo, da samo z usklajevanjem na nivoju posameznih omejitev (s čimer se ukvarja ta naloga) še ne pridemo nujno tudi do konkretnih rešitev celotnega sistema omejitev. V ta namen bi morali dodajati nove omejitve; če na primer dodamo omejitev $X = 23$, bi se zalogi vrednosti Y in Z zmanjšali na 12 oz. 13, kar predstavlja eno izmed konkretnih rešitev prvotnega sistema. Po drugi strani pa bi, če bi dodali omejitev $X = 30$, bi se zalogi vrednosti Y in Z zmanjšali na $Y \in \{12, \ldots, 19\}$ in $Z \in \{11, \ldots, 18\}$; za konkretno rešitev bi bila torej potrebna še ena omejitev oblike $Y = y$ ali pa $Z = z$.

Izkaže se celo, da dejstvo, da se je usklajevanje na nivoju posamičnih omejitev uspešno končalo, še ne pomeni nujno, da je sistem omejitev kot celota sploh rešljiv. Na primer: pri sistemu $X = Y$, $X = Z$, $Y \neq Z$ bi, če bi postavili na začetku vse zaloge vrednosti na $\{1, 2, \ldots\}$, ugotovili, da je vsaka omejitev sama zase že usklajena, kljub temu pa seveda nobena konkretna trojica števil (X, Y, Z) ne more ustrezati vsem trem omejitvam.

1996.3.4

R: 287

Pogosto uporabljena metoda za tajen prenos sporočil je šifriranje sporočila s tajnim ključem (npr. z nekim dovolj dolgim skrivnim geslom ali naključnim številom). Pri tem sta algoritma za šifriranje in dešifriranje sicer lahko javno znana in tudi šifrirano sporočilo je dostopno nepovabljenim očem. Kljub temu pa je dešifriranje sporočila brez poznavanja ključa praktično nemogoče, saj bi preizkušanje vseh možnih ključev trajalo predolgo.

Odločili smo se, da bo ključ dolg 64 bitov.

Predpostavimo, da lahko vlomilec pri ugibanju ključa (npr. pri poskušanju dešifriranja prestreženega besedila z vsemi možnimi ključi) vedno ugotovi, ali je bil njegov poskus uspešen ali ne.

Ker nimamo na razpolago generatorja 64-bitnih naključnih števil, smo si skušali pomagati z generatorjem 16-bitnih psevdonaključnih števil:

```
procedure Random(var r: integer); external;
```

pri čemer je r neka spremenljivka, ki hrani 16-bitno število. Podprogram Random ob vsakokratnem klicu iz trenutnega števila r enolično izračuna novo 16-bitno število in ga shrani nazaj v r.

(*Opomba:* tako se obnašajo praktično vsi generatorji naključnih števil, ki jih najdemo v sistemskih knjižnicah ali vgrajene v programske jezike, le da je tam r običajno neka globalna spremenljivka in uporabnik podprograma nima neposrednega opravka z njo, pač pa bi bil Random v tem primeru funkcija, ki bi vrnila novo vrednost te globalne spremenljivke. Zaradi lažje formulacije naloge smo sintakso klica podprograma tu malo prilagodili.)

Naš algoritem za šifriranje je brez odvečnih podrobnosti približno takšen:

```
var
  j: integer;
  h, m: integer;            { trenutni čas: ura (0..23), minuta (0..59) }
  r: integer;               { naključno 16-bitno število }
  r1, r2, r3, r4: integer;  { 16-bitni deli končnega 64-bitnega ključa }
begin
  Time(h, m);               { v spremenljivkah h in m vrne trenuten čas }
  r := h * 60 + m;          { izračunamo začetno „naključno“ število }
  for j := 1 to 10000 do Random(r);    { malo pomešamo }
  Random(r); r1 := r;       { prvih 16 bitov ključa }
  Random(r); r2 := r;       { drugih... }
  Random(r); r3 := r;
  Random(r); r4 := r;

  Preberi(Besedilo);
  Sifriraj(Besedilo, r1, r2, r3, r4, SifriranoBesedilo);
  Poslji(SifriranoBesedilo);
end.
```

Vlomilec ima na razpolago računalnik z enakimi podprogrami za šifriranje in dešifriranje. Pozna tudi program, s katerim smo zašifrirali sporočilo. Ne pozna pa ključa (64-bitnega števila), s katerim je bilo prestreženo sporočilo šifrirano.

Vlomilčev računalnik zmore preizkusiti 1000 ključev na sekundo. Predpostavimo lahko, da je dešifriranje najzamudnejša operacija in da lahko ostale dele izvajanja vlomilčevega programa zanemarimo.

Izračunaj in utemelji grobo oceno, v kolikšnem času lahko vlomilec pričakuje, da bo uganil pravi ključ in dešifriral prestreženo sporočilo.

Predlagaj, kako bi lahko izboljšal zgornji program.

DRUGO ZAKLJUČNO TEKMOVANJE V ZNANJU RAČUNALNIŠTVA

R: 288

1996.Z

Tombola je igra, v kateri nastopa n (1000, 10000, 100000) kartic, na katerih je po 15 različnih števil med 1 in 100. V bobnu imamo 100 oštevilčenih kroglic (od 1 do 100). Zaporedoma iz bobna vlečemo kroglice, ki jih ne vračamo. Po vsaki izvlečeni kroglici poiščemo kartice, ki vsebujejo sama taka števila, ki so že bila izžrebana (t. j. vseh 15 števil na kartici je že bilo izžrebanih oz. nobeno od števil na kartici ni več v bobnu).

Napiši program Tombola, ki bo v čim krajšem času po vsaki izvlečeni kroglici izpisal število na novo zapolnjenih kartic (zadnja izvlečena kroglica je pokrila zadnje nepokrito polje na kartici).

V datoteki so zapisane kartice, ki so v igri. V vsaki vrstici datoteke je 15 različnih celih števil med 1 in 100, ki predstavljajo številke na eni kartici.

V rešitvi uporabi naslednje podprograme:

- **procedure** Pripravi — Podprogram, katerega klic mora biti prvi izvedljivi stavek v glavnem programu rešitve.
- **procedure** Pospravi — Podprogram, katerega klic mora biti zadnji izvedljivi stavek v glavnem programu rešitve.
- **function** Zrebaj(**var** St: integer): boolean — Funkcija izvleče naslednjo kroglico iz bobna in vrne njeno številko v spremenljivki St — vrednost funkcije je v tem primeru true. Ko v bobnu ni več kroglic, funkcija vrne false in vrednost spremenljivke St ni definirana.
- **procedure** Zapisi(Zadetkov: integer) — Podprogram objavi (zapiše) število zadetkov. Po vsaki izvlečeni kroglici (po klicu funkcije Zrebaj) naj program izračuna število na novo izpolnjenih kartic in pokliče Zapisi.

Čas izvajanja rešitve se začne meriti ob prvem klicu funkcije Zrebaj in neha meriti ob klicu podprograma Pospravi. Čas pred prvo izvlečeno kroglico se ne upošteva, mora pa biti krajši od minute.

Program bomo pognali trikrat:

1. `tombola 1000` — Podatki o 1000 karticah so v datoteki `k1000.txt`.

2. `tombola 10000` — Podatki o 10000 karticah so v datoteki `k10000.txt`.

3. `tombola 100000` — Podatki o 100000 karticah so v datoteki `k100000.txt`.

Ime datoteke z izvršljivim programom mora biti `tombola.exe`. Ime datoteke z izvornim programom mora biti `tombola.pas`. Iz datotek `k*.txt` si lahko pripravite drugače organizirane datoteke, ki se morajo skupaj z izvorno in izvršljivo kodo nahajati na področju `c:\finale`. Skupna velikost datotek ne sme presegati 20 MB.

V datoteki `tombola.txt` opiši svojo rešitev, ki naj obsega od 5 do 50 vrstic besedila. To besedilo je predloga za triminutni ustni zagovor pred komisijo.

Vsak tekmovalec ima na voljo disketo. Ob morebitnih nesrečah je tekmovalec sam odgovoren za svoje rešitve.

Tekmovanje bo trajalo 4 ure. Ob koncu tekmovanja mora biti delujoča rešitev v prej zahtevani obliki. Izpis rešitve mora biti identičen priloženi vzorčni rešitvi. Dodatni popravki, prevajanja, dodajanja, brisanja, preimenovanja, ... ne bodo dovoljena. Tekmovalci brez delujoče rešitve bodo diskvalificirani. Rešitve, ki bodo npr. spreminjale sistemsko uro ali se drugače grdo obnašale, bodo diskvalificirane.

Po končanem tekmovanju bo komisija prenesla vsebino področja `c:\finale` na računalnik komisije (120 MHz, 32 MB RAM, Win95), kjer bo javno preverila pravilnost in izmerila hitrost izvajanja programa. Na računalniku bo deloval diskovni predpomnilnik. Komisija bo upoštevala tudi vsebino izvorne kode in opis rešitve s triminutno predstavitvijo. Najboljše rešitve bo po tekmovanju komisija še enkrat natančno pregledala.

Naivna rešitev naloge je v datoteki `naivna.pas`. Potrudite se napisati rešitev, ki bo delovala hitreje.

```
program Tombola;
uses Zirija;
var
  fKarte: text;
  Izzrebane, Karta: set of 1..100;
  St, NovaSt: integer;
  Zadetkov: longint;
begin
  Pripravi;
  Izzrebane := [];
  if      ParamStr(1) = '1000'   then Assign(fKarte, 'k1000.txt')
  else if ParamStr(1) = '10000'  then Assign(fKarte, 'k10000.txt')
  else if ParamStr(1) = '100000' then Assign(fKarte, 'k100000.txt')
  else    Halt;
  while Zrebaj(NovaSt) do begin
```

```
      Izzrebane := Izzrebane + [NovaSt]; Zadetkov := 0;
      Reset(fKarte);
      while not Eof(fKarte) do begin
        Karta := [];
        while not Eoln(fKarte) do begin
          Read(fKarte, St); Karta := Karta + [St];
        end; {while}
        ReadLn(fKarte);
        if (NovaSt in Karta) and (Karta - Izzrebane = []) then
          Zadetkov := Zadetkov + 1;
      end; {while}
      Zapisi(Zadetkov);
    end; {while}
    Close(fKarte);
    Pospravi;
end. {Tombola}
```

Pomožni podprogrami se nahajajo v datoteki `zirija.pas`.

```
unit Zirija;

interface

  var fZreb, fRezultat: text;

  procedure Pripravi;
  procedure Pospravi;
  function Zrebaj(var St: integer): boolean;
  procedure Zapisi(Zadetkov: integer);

implementation

  uses Dos;
  var ZacetniCas, KoncniCas: longint;

  procedure Pripravi;
  begin
    Assign(fRezultat, 'rezultat.txt'); Rewrite(fRezultat);
    Assign(fZreb, 'zreb.txt'); Reset(fZreb);
    ZacetniCas := -1;
  end; {Pripravi}

  procedure Pospravi;
  var H, M, S, S100: word;
  begin
    GetTime(H, M, S, S100);
    KoncniCas := longint(H) * 360000 + longint(M) * 6000 +
                 longint(S) * 100 + longint(S100);
    WriteLn(fRezultat, 'Trajanje: ',
```

```
      (KoncniCas − ZacetniCas) / 100:0:1, ' sekund');
    Close(fZreb);
    Close(fRezultat);
  end; {Pospravi}

  function Zrebaj(var St: integer): boolean;
  var H, M, S, S100: word;
  begin
    if ZacetniCas < 0 then begin
      GetTime(H, M, S, S100);
      ZacetniCas := longint(H) * 360000 + longint(M) * 6000 +
                    longint(S) * 100 + longint(S100);
    end; {if}
    if Eof(fZreb) then
      Zrebaj := false
    else begin
      ReadLn(fZreb, St); Write(fRezultat, St);
      Zrebaj := true;
    end; {if}
  end; {Zrebaj}

  procedure Zapisi(Zadetkov: integer);
  begin
    WriteLn(fRezultat, ' ', Zadetkov);
  end; {Zapisi}

end. {Zirija}
```

[Datoteke s podatki o karticah, ki so jih uporabljali na tekmovanju leta 1996, se dobijo na naslovu `http://rtk.ijs.si/`. Opozorimo še na to, da so na tem tekmovanju programi že lahko tekli v zaščitenem načinu (DPMI), kar pomeni, da so lahko izkoriščali do 16 MB pomnilnika, le delo z bloki, večjimi od 64 KB, je bilo malo bolj nerodno. — *Op. ur.*]

REŠITVE NALOG ZA PRVO SKUPINO

R1996.1.1

Program izpiše urejene vrednosti tabele t, kar v našem primeru pomeni števila od 1 do MaxT. N: 259

V prvem stavku napolnimo tabelo t z vrednostmi od MaxT do 1. Zanka **repeat** opravlja nalogo t.i. naključnega urejanja podatkov (angl. *shuffle-sort*[45]).

[45] Izraz shuffle-sort se res uporablja za algoritem, ki naključno premeša podatke, preveri, če so urejeni, in to ponavlja, dokler ne dobi urejenega zaporedja; vendar pa se uporablja ta izraz tudi za nek algoritem za urejanje podatkov z več paralelno delujočimi enotami (Harold S. Stone: *Parallel processing with the perfect shuffle*, IEEE Trans. on Computers, C-20(2):153–61, Feb. 1971; Jai Menon: *A study of sort algorithms for multiprocessor database machines*, VLDB 1986, pp. 197–206).

V vsaki ponovitvi te zanke naključno zamenjamo dva sosednja elementa tabele t, nato pa v zanki **while** preverimo, če so podatki urejeni v naraščajočem vrstnem redu (ta zanka pravzaprav to preveri le za vse elemente razen zadnjega in če je tu vse v redu, bo pogoj **until** preveril še, če je zadnji element vsaj tolikšen kot predzadnji). Ko so enkrat vsi podatki urejeni, se izvajanje zanke **repeat** konča in vrednosti tabele izpišemo.

Naloga sprašuje še, kaj bi lahko naredili, da bi program to, kar sicer izpiše že zdaj, izpisal hitreje. Po vsem, kar smo videli, lahko odgovorimo nekako takole:

```
const MaxT = 5;
var i: integer;
begin
  for i := 1 to MaxT do WriteLn(i);
end.
```

N: 259

R1996.1.2

Ker sta ogrlici krožni, ne vemo vnaprej, katera kroglica druge ogrlice ustreza npr. prvi kroglici prve ogrlice (tudi če sta ogrlici zares enaki). Zato moramo poskusiti vse možne zamike (zunanja zanka v spodnjem programu); pri vsakem zamiku i pa moramo drugo ogrlico pred primerjavo v mislih krožno zamakniti za i mest. Spodnji program šteje kroglice od 0 do N − 1 namesto od 1 do N, ker je tako lažje uporabiti operator **mod**. Če se pri nekem zamiku ogrlici v celoti ujemata, lahko takoj končamo (spremenljivka Ok).

```
program AliStaOgrliciEnaki;
const
  N = 10;
var
  Ogrlica1, Ogrlica2: array [0..N − 1] of char;
  i, j: integer;
  Ok: boolean;
begin
  Ogrlica1 := '1234567890';
  Ogrlica2 := '7890123456';
  i := 0;
  repeat
    j := 0; Ok := true;
    while (j < N) and Ok do begin
      Ok := Ogrlica1[j] = Ogrlica2[(i + j) mod N]; j := j + 1;
    end; {while};
    i := i + 1;
  until Ok or (i = N);
  if Ok then
    WriteLn('Ogrlici sta enaki.')
```

```
  else
    WriteLn('Ogrlici nista enaki.');
end. {AliStaOgrliciEnaki}
```

Nalogo bi lahko rešili tudi tako, da bi enega od nizov podvojili (Ogrlica1 := Ogrlica1 + Ogrlica1; pravzaprav bi bilo dovolj že Ogrlica1 := Ogrlica1 + Copy(Ogrlica1, 1, Length(Ogrlica1) − 1)) in nato na poljuben način preverili, če se drugi (Ogrlica2) pojavlja v njem kot podniz.

R1996.1.3

Ko beremo vhodne podatke, hranimo v spremenljivki Oklepaj podatek o tem, ali se trenutno nahajamo znotraj oklepajev <...> ali ne. Če se ne, prebrane vhodne znake tudi sproti izpisujemo. Ko naletimo na oklepaj ali zaklepaj, pa vrednost spremenljivke Oklepaj primerno popravimo. Da nam bo lažje, smo predpostavili, da oklepaji in zaklepaji niso gnezdeni (npr. <...<...>...>); če bi bili gnezdeni, bi morala biti spremenljivka Oklepaj tipa integer in bi štela, koliko oklepajev < je trenutno odprtih; ob vsakem znaku < bi jo povečali za ena, ob vsakem > pa zmanjšali za ena. Ostale znake bi izpisovali le, če bi bila Oklepaj = 0.

N: 259

```
program OckaNehaj(Input, Output);
var ch: char;
    Oklepaj: boolean;
begin
  Oklepaj := false;
  while not Eof(Input) do begin
    while not Eoln(Input) do begin
      Read(Input, ch);
      if ch = '<' then Oklepaj := true
      else if ch = '>' then Oklepaj := false
      else if not Oklepaj then Write(Output, ch);
    end; {while};
    ReadLn(Input);
    if not Oklepaj then WriteLn(Output);
  end; {while}
end. {OckaNehaj}
```

S primernim jezikom, kot je na primer `awk`, je lahko rešitev še veliko krajša:

```
BEGIN { RS = "<[^>]*>" }
{ printf "%s", $0 }
```

Program v `awk`u je sestavljen iz pravil (*rules*), vsako pravilo pa iz vzorca (*pattern*) in akcije (*action*). Vzorec določa, kdaj se pravilo izvede, akcija pa, kaj se takrat zgodi. Tu imamo dve pravili; pri prvem je vzorec BEGIN, kar pomeni, naj se to pravilo izvede na samem začetku. `awk` razdeli svoj vhodni tok na zapise (*records*) in pri tem kot ločilo uporabi niz ali regularni izraz RS (privzeta

vrednost RS je znak za konec vrstice). Prvo pravilo postavi RS na regularni izraz, ki se ujema z vsakim nizom oblike <...> (med oklepajema je lahko poljuben niz znakov, le nobenega > ne sme vsebovati — brez te dodatne zahteve bi se ta izraz ujel z vsem besedilom od prvega < do zadnjega > v celi datoteki!). Zapisi, ki tako nastanejo, so torej ravno koščki besedila med komentarji; če vse izpišemo, smo dobili ravno celotno besedilo brez komentarjev. To naredimo z drugim pravilom; to vzorca sploh nima, zato se izvede po enkrat pri vsakem zapisu. Vsebino vzorca dobimo v nizu $0, izpišemo pa ga s stavkom printf, ki deluje podobno kot istoimenska funkcija v C/C++.

N: 260

R1996.1.4

Naredimo podprogram, ki bo meril čas med pritisnjenimi tipkami, jemal spodnji bit teh časov in te bite vtikal posamezno v ključ. Pazimo na to, če se časi oziroma tipke ponavljajo (autorepeat, avtomatski keyboard stufferji). Torej, če je med dvema zaporednima pritiskoma minilo premalo časa, se za drugega raje ne zmenimo; podobno, če je med enim in drugim pritiskom minilo skoraj čisto enako časa kot med drugim in tretjim, pa gre ves čas za isto tipko, si mislimo, da uporabnik najbrž tišči tipko pritisnjeno in se za te pritiske raje ne zmenimo. Ko je ključ poln, podprogram konča delo.

```
const
  MinMed = 20;      { Minimalni sprejemljivi čas med dvema pritiskoma
                      (drugače drugi pritisk ignoriramo). }
  Natancnost = 2;   { Koliko niha čas med dvema zaporednima „pritiskoma“,
                      če uporabnik v resnici tišči tisto tipko ves čas pritisnjeno. }
  Dolzina = ...;    { Dolžina ključa v bitih. }
type
  KljucT = array [0..(Dolzina + 7) div 8 - 1] of byte;

procedure PripraviKljuc(var Kljuc: KljucT);
var Bit, Cas, PrejCas, Med, PrejMed: integer;
    Tipka, PrejTipka: char;
begin
  WriteLn('Tipkajte...');
  Bit := 0; PrejTipka := ReadKey; PrejCas := Timer; PrejMed := 0;
  while Bit < Dolzina do begin
    Tipka := ReadKey; Cas := Timer; Med := Cas - PrejCas;
    if (Med > MinMed) and ((Tipka <> PrejTipka) or
                           (Abs(Med - PrejMed) > Natancnost)) then begin
      if (Bit mod 8) = 0 then Kljuc[Bit div 8] := 0;
      Kljuc[Bit div 8] := Kljuc[Bit div 8] or ((Med and 1) shl (Bit mod 8));
      PrejMed := Med; PrejTipka := Tipka; Bit := Bit + 1;
      Write('.'); { Obvestimo uporabnika. }
    end; {if}
    PrejCas := Cas;
```

```
  end; {while}
  WriteLn; WriteLn('Ključ uspešno prebran.');
end; {PripraviKljuc}
```

Ena od slabosti tega pristopa je, če na primer Timer od nekod dobi čas v stotinkah sekunde, pa ga potem pomnoži z 10, da bi dobil tisočinke: potem bo Med vedno sod in naš ključ bo sestavljen iz samih ničel. Malo robustnejša rešitev bi bila, če bi gledali za tri zaporedne pritiske na tipko, ali je minilo več časa med prvim in drugim ali med drugim in tretjim (v prvem primeru bi dodali svojemu naključnemu zaporedju recimo bit 0, v drugem pa bit 1). To različico uporabljajo na primer na `http://www.fourmilab.ch/hotbits/`, kjer namesto pritiskov na tipke spremljajo razpad neke radioaktivne snovi.

Glej tudi 4. nalogo za tretjo skupino (str. 267).

REŠITVE NALOG ZA DRUGO SKUPINO

R1996.2.1

N: 260

Besedo ali stavek, ki se nazaj bere enako kot naprej, imenujemo *palindrom*. Podobno so *številski palindromi* števila, katerih zapis v desetiškem sistemu je z desne proti levi enak kot z leve proti desni (taki števili sta na primer 48584 ali 232).

Podprogram xx(x) vrne število, ki ga dobimo, če x zapišemo v desetiškem sistemu in ga preberemo v nasprotni smeri. Pri b = 10 je namreč x **mod** b ravno najbolj desna števka v desetiškem zapisu števila x, tako da vrstica y := y * b + x **mod** b v bistvu pripiše številu y na desni še x-ovo skrajno desno števko. Po tem prirejanju imata tako y kot x isto skrajno desno števko, zato x − y **mod** b v naslednji vrstici pravzaprav postavi skrajno desno števko števila x na 0, deljenje z b v isti vrstici pa skrajno desno x-ovo števko še pobriše. (Ker operator **div** tako ali tako zaokroži rezultat deljenja navzdol, bi lahko vzeli tudi preprosto x := x **div** b).

Glavni blok programa izpiše vsa cela števila x od 1 do 1000, za katera velja: če temu x prištejemo število (recimo mu $\hat{x}$), ki ga dobimo, če x preberemo z desne proti levi, je vsota neko palindromno število. Izkaže se, da ima to lastnost 291 števil. Primer takega števila je recimo 65, saj je $65 + 56 = 121$, torej palindrom.

Do tega, kateri so ugodni x, lahko pridemo tudi z razmislekom. Od števil $x < 10$ so ugodna 1, 2, 3 in 4, ostala pa ne. Oglejmo si zdaj dvomestna števila: $x = 10a + b$, torej $\hat{x} = 10b + a$. Kdaj je vsota $S := x + \hat{x} = 10(a + b) + (a + b)$ palindrom? No, če je $a + b < 10$, sta obe števki S-ja enaki, tako da je v redu. Temu pogoju ustreza 45 števil: 10..18, 20..27, 30..36, 40..45, 50..54, 60..63, 70..72, 80..81 in 90. Sicer pa je $a + b = 10 + c$ za nek $c \in 0..8$ in $S = 100 + 10(1 + c) + c$, kar je lahko palindrom le, če je $c = 1$, torej če se a in b seštejeta v 11. Ugodni x so torej 29, 38, 47, 56, 65, 74, 83 in 92.

Oglejmo si še tromestna števila: $x = 100a + 10b + c$, $S = 100(a + c) + 10(2b) + (a + c)$. Ločili bomo primere glede na to, ali pride pri seštevanju do prenosa naprej. (*i*) Ena možnost je na primer ta, da je $a+c < 10$ in $b < 5$; hitro vidimo, da je takih x-ov $45 \cdot 5 = 225$. (*ii*) Če $a+c < 10$ in $b \geq 5$, bo $2b = 10+d$ za nek d in $S = 100(a + c + 1) + 10d + (a + c)$, kar ne more biti palindrom, če je res tromestno število; štirimestno pa je v primeru, ko je $a + c = 9$, S pa je oblike $1000 + 10d + 9$, kar tudi ne more biti palindrom. (*iii*) Če $a + c = 10 + d$ in $b < 5$, bo $1 + 2b < 10$ in S bo oblike $1000 + 100d + 10(1 + 2b) + d$. To je torej palindrom natanko tedaj, ko je $d = 1$ in $1 + 2b = d = 1$, torej $b = 0$, za a in c pa je potem 8 načinov, da se seštejeta v 11 $(2 + 9, 3 + 8, \ldots, 9 + 2)$. (*iv*) Če $a + c = 10 + d$ (za nek $d \in 0..8$) in $2b = 10 + e$ (za nek $e \in \{0, 2, 4, 6, 8\}$), ima S obliko $1000 + 100(1 + d) + 10(1 + e) + d$, torej je palindrom le, če je $e = d = 1$, kar pa ni mogoče, saj bi $e = 1$ pomenilo $2b = 11$.

Skupaj smo torej našteli 4 enomestne, $45 + 8$ dvomestnih in $45 \cdot 5 + 8$ tromestnih števil z iskano lastnostjo; to lastnost pa ima tudi število 1000, kar nam da vsega skupaj 291 števil.

N: 261

R1996.2.2

Naj bo x položaj prvega ojačevalnika, na intervalu od z do k pa naj bo neka ovira. Kateri x so zaradi te ovire neugodni? Tisti, pri katerih pademo nekoč v interval med z in k, če začnemo pri x in prištevamo 4000. Naj bo $z = 4000q + z'$ za nek $0 \leq z' < 4000$ in $k = 4000r + k'$ za nek $0 \leq k' < 4000$. (Pri tem vemo, da je $r = q$ ali pa $r = q + 1$, ker bi bila ovira drugače zanesljivo dolga vsaj 4000 m, kar bi bilo brezupno.) Očitno velja, da na ojačevalnike $x + 4000s$ za $s < q$ in $s > r$ ta ovira ne more vplivati. Če je $r = q$, bo ovira mogla vplivati le na ojačevalnik $x + 4000q$, ki torej ne sme ležati na intervalu od z do k. Tako imamo pogoja, da ovira ni v nadlego: veljati mora $x + 4000q < z$ ali pa $x + 4000q \geq k$, kar pa je isto kot $x < z'$ oz. $x \geq k'$. Druga možnost, $r = q + 1$, pomeni, da lahko vpliva ta ovira na ojačevalnika $x + 4000q$ in $x + 4000r$; no, zanesljivo vedno velja $x + 4000q < k$ in $x + 4000r > z$, tako da bo moral prvi od omenjenih dveh ojačevalnikov ležati pred oviro, drugi pa za njo. To nam da pogoja $x + 4000q < z$ in $x + 4000r \leq k$ oziroma $x < z'$ in $x \geq k'$. (Primere, ko je $r = q + 1$, lahko enostavno ločimo od tistih, ko je $r = q$, takole: če je $r = q$, sledi iz $z < k$ tudi $z' < k'$; če pa je $r = q + 1$, mora biti $k' \leq z'$, saj bi bila sicer ovira daljša od 4000 m.)

Spodnji program torej lahko ravna takole: za vsako oviro $[z, k]$ izračuna $z' = z \bmod 4000$ in $k' = k \bmod 4000$. Vsaka ovira, pri kateri je $k' \leq z'$, nam dá pogoj, da mora biti $x \geq k'$, torej lahko za začetni x vzamemo največjo k' po vseh teh ovirah. Od tu naprej postopoma povečujemo x, če najdemo kakšno oviro, ki ji trenutni x ne ustreza; vsakič ga povečamo kar najmanj, vendar toliko, da ga tista ovira ne omejuje več: tako vemo, da ne bomo nobenega spregledali (v vsakem trenutku vemo, da bi kateri koli x, manjši od trenutnega, gotovo imel težave pri vsaj eni oviri). Torej: če najdemo oviro, za katero je

$z' \le x < k'$, moramo postaviti x na k', če pa najdemo táko, za katero je $k' \le z' \le x$, smo v težavah, saj vemo, da bi moral biti pri taki oviri x manjši od z', obenem pa očitno noben manjši x ni ustrezal vsem dosedanjim oviram.

```
program Postavi;
var
  Zacetek: array [1..StOvir] of integer;
    { razdalja med začetkom kabla in začetkom ovire (v metrih) }
  Konec: array [1..StOvir] of integer;
    { razdalja med začetkom kabla in koncem ovire (v metrih) }
  Ovira: integer; { števec ovir }
  Kje: longint;   { možna lokacija prvega ojačevalnika}
begin
  Kje := 0;
  for Ovira := 1 to StOvir do begin
    Zacetek[Ovira] := Zacetek[Ovira] mod 4000;
    Konec[Ovira] := Konec[Ovira] mod 4000;
    if (Zacetek[Ovira] >= Konec[Ovira]) and (Konec[Ovira] > Kje) then
      Kje := Konec[Ovira];
  end; {for}
  Ovira := 1;
  repeat
    if Zacetek[Ovira] <= Kje then begin
      if Konec[Ovira] <= Zacetek[Ovira] then begin
        Kje := 4000; Ovira := StOvir
      end else if Konec[Ovira] > Kje then begin
        Kje := Konec[Ovira]; Ovira := 0
      end; {if}
    end; {if}
    Ovira := Ovira + 1;
  until Ovira > StOvir;
  if Kje < 4000 then
    WriteLn('Prvi ojačevalnik naj bo na razdalji ', Kje, ' metrov.')
  else WriteLn('Problem je nerešljiv.');
end. {Postavi}
```

Ta program bi se dalo še izboljšati, da ne bi vsakič, ko popravi trenutni položaj prvega ojačevalnika (spremenljivka Kje), šel pregledovat vseh ovir znova od začetka. Zadostovalo bi že, če bi ovire pred izvajanjem glavne zanke **repeat** uredili po naraščajoči k' (pri tistih s $k' \le z'$ pa bi si mislili, da imajo $k' = 4000$). Tako bi vedeli, da, ko se Kje recimo pri i-ti oviri poveča, postane enak njenemu k' in s tem $\ge$ od k' vseh prejšnjih ovir, torej mu one zdaj prav gotovo niso v napoto in jih ni treba ponovno pregledovati. Zanka **for** bi ostala nespremenjena, v zanki **repeat** pa bi se znebili stavka Ovira := 0.

N: 262

R1996.2.3

Med dvema biseroma iste barve (ali dvema brezbarvnima) ogrlice nima smisla prestriči; lahko prestrižemo eno mesto bolj levo ali desno in še vedno dobimo vse, kar bi dobili tudi prej.

Prav tako nima smisla prestriči med barvnim in brezbarvnim, če je prvi barvni za tistim brezbarvnim iste barve. Na primer: `rbbrm` nima smisla prestriči med prvim in drugim, saj bomo tako pobrali le `rbbr`; če bi prestrigli desno od drugega `r`, bi vse to še vedno dobili, za povrhu pa tudi `m`. Zato se bomo v nadaljevanju tega razmisleka pretvarjali, da so brezbarvni biseri, ki jih na obeh straneh obdajata bisera iste barve, v resnici tudi sami take barve.

Če bi radi prestrigli med rdečim in modrim, pa je vmes še nekaj brezbarvnih, je vseeno, pri katerem prerežemo.

Zato je dovolj, če se pri razmišljanju o tem, kje bi prestrigli, omejimo na primer na takšna mesta: takoj za biserom ene barve, ki mu sledi 0 ali več brezbarvnih in nato vsaj en biser druge barve. Okolico takšnega mesta si lahko predstavljamo takole (barvni biseri so podčrtani):

...m	bb...b	rr...r	bb...b	mm...m	bb...b	rr...r	bb...b	mm...m	bb...b	r...
	↑	↑	↑	↑	↑	↑	↑	↑	↑	
	i	j	i_1	j_1	i_2	j_2	i_3	j_3	i_4	

Če prestrižemo takoj levo od bisera j_1, bomo lahko obdržali vse bisere od j do $i_3 - 1$. Naslednje mesto, o katerem bi razmišljali, je takoj levo od bisera j_2; če bi prestrigli tam, bi obdržali vse od j_1 do $i_4 - 1$, pri čemer je i_4 indeks prvega rdečega bisera desno od i_3. Tako nadaljujemo po ogrlici, dokler ne pridemo naokoli: če je bil j_1 prvi biser, pri katerem smo poskušali prestriči, bomo končali po tistem, ko bomo razmislili še o striženju levo od j.

Podprogram Nasl v spodnji rešitvi poišče, če mu damo indeks nekega bisera i, indeks prvega naslednjega bisera nasprotne barve (i_1) in še indeks (j) prvega brezbarvnega bisera v strnjenem zaporedju brezbarvnih biserov tik pred i_1.

Podprogram KjeRezati začne tako, da poišče prvi barvasti biser (h), vendar pa ta ni nujno na začetku zaporedja biserov svoje barve (lahko je to biser v s[0], pa so še na drugem koncu tabele biseri iste barve). Zato si nato pomagamo s podprogramom Nasl, da poiščemo primeren i. V nadaljevanju se lahko zgodi, da je $i_1 = i$, kar pomeni, da biserov druge barve sploh ni (in bomo ne glede na mesto reza dobili celo ogrlico), ali pa, da je $i_2 = i$, kar pomeni ogrlico oblike `rrr bbb mmm bbb`, ki jo bomo tudi lahko dobili celo, če le ne prerežemo med biseroma iste barve. Drugače pa poiščemo še j_2 in i_3 in že imamo vse, kar potrebujemo, da ocenimo možnost reza pri j_1. Nato se v zanki pomikamo naprej, dokler ne pridemo spet naokoli: zadnje mesto reza, ki ga moramo oceniti, je prvotna vrednost j (ki si jo podprogram zapomni kot j_0).

Pri delu z indeksi moramo ves čas upoštevati, da je ogrlica krožna. Zaradi lažjega naslavljanja štejemo indekse v nizu od 0 do $n - 1$ namesto od 1 do n. Za indeksom i pride $(i+1) \bmod n$, pred njim pa $(i-1) \bmod n$, vendar slednje raje pišimo kot $(i + n - 1) \bmod n$, da ne bi imel mod kakšnih težav zaradi

negativnega operanda.[46] Če vrne KjeRezati vrednost i, to pomeni rezanje med biseroma s[i − 1] in s[i] (če vrne 0, pa seveda rezanje med s[n − 1] in s[0].

```
const MaxDolzina = ...;
type Niz = array [0..MaxDolzina − 1] of char;

function KjeRezati(s: Niz; n: integer): integer;

  procedure Nasl(i: integer; var j, i1: integer);
  begin
    i1 := (i + 1) mod n;
    while i1 <> i do
      if (s[i1] <> 'b') and (s[i1] <> s[i]) then break
      else i1 := (i1 + 1) mod n;
    j := (i1 + n − 1) mod n;
    while j <> i do
      if s[j] <> 'b' then break
      else j := (j + n − 1) mod n;
    j := (j + 1) mod n;
  end; {Nasl}

var h, h1, i, j, j0, i1, j1, i2, j2, i3, b, bi, c: integer;
begin
  h := 0;
  while h <> n do     { poiščimo prvi barvni biser }
    if s[h] <> 'b' then break
    else h := h + 1;
  if h = n then       { vse brezbarvno }
    begin KjeRezati := 0; exit end;
  Nasl(h, h1, i); Nasl(i, j, i1);
  if i1 = i then      { nista prisotni in modra in rdeča }
    begin KjeRezati := i; exit end;
  Nasl(i1, j1, i2);
  if i2 = i then      { en kos modre in en kos rdeče }
    begin KjeRezati := i; exit end;
  b := 0; bi := 0; j0 := j;
  repeat
    Nasl(i2, j2, i3);
    c := i3 − j;      { izplen, če prerežemo pred j1 }
    if c <= 0 then c := c + n;
    if c > b then begin b := c; bi := j1 end;
    i := i1; j := j1; i1 := i2; j1 := j2; i2 := i3;
  until j = j0;
```

[46] V resnici je operator a **mod** n v standardnem pascalu definiran tako, da vedno vrača vrednosti z intervala $0, \ldots, n-1$, vendar se nekateri prevajalniki tega ne držijo in za a **mod** n raje vzamejo vrednost a − (a **div** n) * n, ki je pri negativnem a lahko negativna. Za več o različnih definicijah operacij div in mod glej rešitev naloge 1997.2.1, str. 309.

```
  KjeRezati := bi;      { položaj najboljšega izplena }
end;
```

N: 262

R1996.2.4

Podprogram Vsiljivec(N) lahko najprej preveri, če že obstaja omejitev za celo N-jevo podmrežo; če obstaja, je treba samo osvežiti njen čas in že lahko končamo. Drugače pa preveri, če že obstaja omejitev za kak drug (od N-ja različen) naslov iz N-jeve podmreže (pri tem omejitve, ki so že prestare, ignorira oz. jih kar sproti pobriše iz seznama). Če najdemo kakšno tako omejitev (zastavica Nasel), moramo uvesti omejitev za celo N-jevo podmrežo, dotedanje omejitve za posamezne naslove iz N-jeve podmreže pa lahko pobrišemo. Če pa ni bilo nobene take omejitve, je mogoče v seznamu vsaj omejitev za sam naslov N; če je, obnovimo njen čas, drugače pa jo dodajmo kot novo omejitev.

Podprogram Sprosti je še preprostejši, saj mora le prečesati vse naslove v seznamu (začne lahko pri $\langle 0, 0 \rangle$) in pobrisati tiste, ki so že prestari.

```
procedure Vsiljivec(N: Naslov);
var M: Naslov; t, tM: integer; Nasel: boolean;
begin
  M[1] := N[1]; M[2] := 0; t := Cas;
  { Preverimo, če že obstaja omejitev za N-jevo podmrežo. }
  if Poisci(M) > 0 then begin Obnovi(M, t); exit end;    {⋆}
  { Preverimo, če obstaja omejitev za kak drug naslov iz te podmreže. }
  Nasel := false;
  while Naslednji(M, tM) and not Nasel do begin
    if M[1] <> N[1] then break;
    if (M[2] <> N[2]) then
        if t − tM < TTL then Nasel := true
                        else Brisi(M);
  end; {while}
  if Nasel then begin
    { Zamenjajmo omejitve za naslove te podmreže z omejitvijo za celo podmrežo. }
    M[1] := N[1]; M[2] := 0;
    while Naslednji(M, tM) do
      if M[1] <> N[1] then break else Brisi(M);
    M[1] := N[1]; M[2] := 0; Dodaj(M, t);
  end else if Poisci(N) > 0 then Obnovi(N, t)
  else Dodaj(N, t);
end; {Vsiljivec};

procedure Sprosti;
var N: Naslov; t, tN: integer;
begin
  N[1] := 0; N[2] := 0; t := Cas;
  while Naslednji(N, tN) do
    if t − tN > TTL then Brisi(N);
```

end; {*Sprosti*}

Zgornji podprogram Vsiljivec ima še eno morebitno slabost: na začetku preveri, če že obstaja omejitev za podmrežo, in če obstaja, jo v vsakem primeru obnovi. Toda ta omejitev je mogoče že prestara (starejša od TTL sekund) — natančneje, prestara je lahko za največ toliko, kolikor časa mine med dvema zaporednima klicema podprograma Sposti. Podprogram Vsiljivec bi na osnovi take zastarele omejitve spet prepovedal dostop celi podmreži namesto le naslovu N. Bolj pošteno bi bilo v takem primeru omejitev za podmrežo pobrisati in uvesti le omejitev za naslov N. Ker podprogrami, ki jih imamo na voljo za delo z omejitvami, ne omogočajo bolj elegantnega načina za ugotavljanje starosti omejitve, si bomo morali pomagati s podprogramom Naslednji. Vrstico {⋆} podprograma Vsiljivec bi morali zamenjati z nečim takšnim:

```
M[1] := N[1] − 1; M[2] := 9999;
if Naslednji(M, tM) then if (M[1] = N[1]) and (M[2] = 0) then begin
  if t − tM < TTL then Obnovi(M, t)
  else begin Brisi(M); Dodaj(N, t) end;
  exit;
end; {if}
M[1] := N[1]; M[2] := 0;
```

REŠITVE NALOG ZA TRETJO SKUPINO

R1996.3.1

Komunikacija poteka v dveh fazah — prva faza je vzpostavljanje zveze in druga pošiljanje sporočila. N: 263

Zveza se vzpostavi tako, da računalnik, ki želi oddati sporočilo, pošlje drugemu računalniku sporočilo „`PošiljalBi`“. Drugi lahko na to odgovori s „`KarDaj!`“ (zveza je vzpostavljena, sporočilo se pošlje in izpiše na drugem računalniku) ali z „`RajeNe!`“ (drugi uporabnik ne želi sprejemati sporočil; računalnik, ki oddaja sporočilo, o tem obvesti svojega uporabnika).

Črni scenarij je naslednji:

računalnik A	računalnik B
uporabnik začne tipkati sporočilo	
	uporabnik začne tipkati sporočilo
uporabnik konča tipkanje	
računalnik pošlje „`PošiljalBi`“	
računalnik čaka v stavku Sprejmi(s)	
	uporabnik konča tipkanje
	računalnik pošlje „`PošiljalBi`“
	računalnik čaka v stavku Sprejmi(s)
računalnik sprejme „`PošiljalBi`“	
	računalnik sprejme „`PošiljalBi`“

Nobeden od računalnikov ne pošlje sporočila, oba se iz podprograma PosljiSporocilo vrneta v glavni program, uporabnik pa pri tem ni obveščen, da njegovo sporočilo ni bilo poslano.

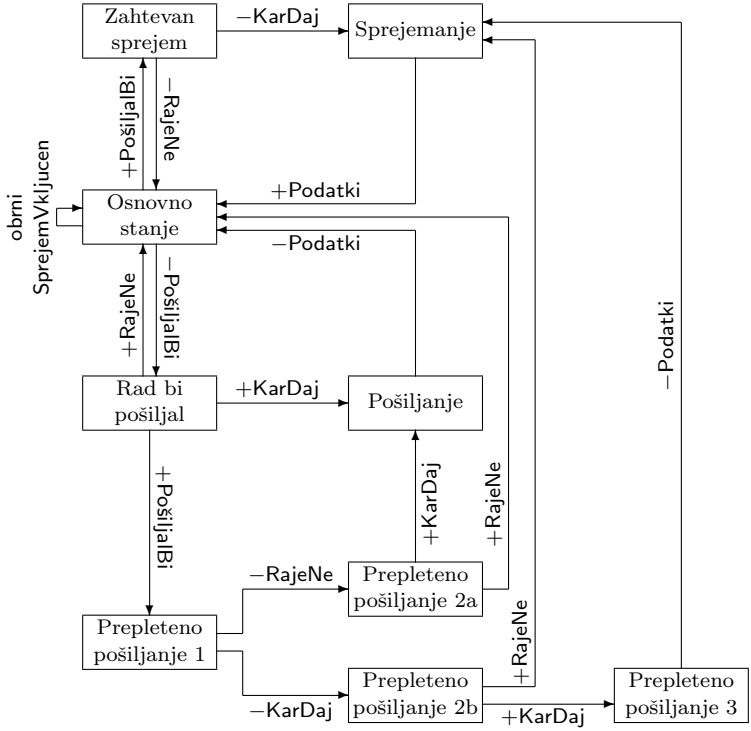
Izkaže se, da je to pravzaprav edina tovrstna resna težava našega protokola. Izognemo se ji lahko tako, da dovolimo „prepleteno prejemanje“:

```
procedure PosljiSporocilo(S: string);
var Odg: string;
begin
  Poslji('PošiljalBi');
  repeat Sprejmi(Odg) until Odg <> '';
  if Odg = 'RajeNe!' then WriteLn('Kolega ima izključen sprejem.')
  else if Odg = 'KarDaj!' then Poslji(S)
  else if Odg = 'PošiljalBi' then begin
    { Prepleteno prejemanje. }
    { Najprej mu povejmo, kaj si mislimo o njegovi zahtevi. }
    if SprejemVkljucen then Poslji('KarDaj!')
    else Poslji('RajeNe!');
    { Nato poglejmo, kaj si misli on o naši. }
    repeat Sprejmi(Odg) until Odg <> '';
    { Naj pošljemo svoje podatke? }
    if Odg = 'KarDaj!' then Poslji(S);
    { Naj čakamo na njegove? }
    if SprejemVkljucen then begin
      repeat Sprejmi(S) until S <> '';
      WriteLn(S);
    end; {if}
  end; {if}
end; {PosljiSporocilo}
```

Drugih delov programa ni treba spreminjati. Ta rešitev še vedno predpostavlja, da se paketi ne izgubljajo (lahko pa se zakasnijo) in da se njihov vrstni red ne spreminja (prejemnik jih dobi v takem vrstnem redu, v kakršnem jih je pošiljatelj oddal). Izkaže pa se, da prejemnika pri tem protokolu nikoli ne bodo čakala več kot tri sporočila hkrati (npr. če je drugi računalnik pri prepletenem prejemanju zavrnil našo željo po prenosu sporočila, mi pa smo njegovo sprejeli, se lahko zgodi, da se nam v vrsti naberejo njegova sporočila `RajeNe`, uporabnikovi podatki in še (ker je z njegovega vidika prepleteno prejemanje končano in si lahko zaželi pošiljati naslednje sporočilo) `PošiljalBi`; dlje kot to pa ne more iti, ker pošiljatelj zdaj čaka na naš odgovor na ta `PošiljalBi`, dobil pa ga ne bo prej, preden ne bomo mi pobrali vseh treh sporočil iz vrste).

O tem, da je dobljeni protokol zdaj res brez napak, se lahko prepričamo tudi z avtomatskim testiranjem protokola.[47] Program lahko predstavimo z grafom;

[47] Gl. npr. Tone Vidmar, *Računalniška omrežja in storitve*, 1997, razd. 4.3.



Graf stanj, s katerim lahko opišemo naš popravljeni protokol. Prehodi, ki jih sproži prevzem sporočila x, so označeni s „+x“; prehodi, pri katerih odpošljemo sporočilo x, pa so označeni z „−x“. Sporočilo Podatki predstavlja podatke, ki jih je vtipkal uporabnik in jih hoče poslati na drugi računalnik. Če sta iz nekega stanja narisani puščici tako za −RajeNe kot za −KarDaj, to pomeni, da se pri SprejemVkljucen = true izvede vedno prehod −KarDaj, sicer pa −RajeNe.

---

točkam pravimo stanja, povezave pa so prehodi med njimi (glej sliko). Ob vsakem prehodu lahko računalnik pošlje ali sprejme eno sporočilo ter spreminja vrednosti svojih spremenljivk. Stanje celotnega sistema je zdaj sestavljeno iz tega, v katerem stanju grafa se nahaja prvi in v katerem drugi računalnik, kakšne so vrednosti njunih spremenljivk (v našem primeru je pomembna le spremenljivka SprejemVkljucen) in kakšna sporočila so v njunih vrstah (za vsak računalnik imamo medpomnilnik oz. vrsto sporočil, ki mu jih je drugi računalnik že poslal, on jih pa še ni prevzel). Začetno stanje sistema poznamo (programa na začetku izvajanja, prazne vrste ipd.), nato pa lahko za vsako stanje ugotovimo, katera stanja bi mu utegnila slediti (odvisno od tega, kateri računalnik izvede prehod in kakšen prehod naredi). Če omejimo dolžino vrst, je možnih stanj sistema le končno mnogo, zato lahko s takšnim siste-

matičnim preiskovanjem prej ali slej pridemo do vseh; če pri tem ne opazimo nobenih težav, je z našim protokolom vse v redu. Pri razmeroma preprostem protokolu, kot je naš, možnih stanj sistema običajno sploh ni tako zelo veliko (npr. okoli sto), razen če ni s protokolom kaj narobe in se potem nakopiči ogromno „čudnih“ oz. problematičnih stanj.

N: 264

R1996.3.2

Podprogram KoncenIzpis nam pomaga ločiti med primeri, ko ima nek program končen izpis, in primeri, ko ima neskončen izpis. Če pa bi mi v resnici radi ločili med tem, da nek program Prog sploh ničesar ne izpiše, in tem, da ima nek izpis (ki je lahko končen ali neskončen), bomo morali Prog oviti v nek nov program (recimo mu Prog2) in to tako, da bo imel Prog2 končen izpis, ko bo imel Prog prazen izpis, in neskončen izpis, ko bo imel Prog neprazen izpis.[48] Potem bo odgovor, ki nam ga bo dal KoncenIzpis, ko mu bomo pokazali program Prog2, povedal ravno to, kar nas zanima: ali ima Prog prazen izpis ali ne. Primer takega programa Prog2 je program, ki v neskončni zanki poganja program Prog vedno na enih in istih vhodnih podatkih: če Prog sploh kaj izpiše, bo to izpisal v vsaki ponovitvi te zanke in izpis programa Prog2 bo zato gotovo neskončen; če pa Prog ne izpiše ničesar, tudi Prog2 ne bo izpisal ničesar.

```
program Prog2(Prog, Data);
begin
  while true do
    Execute(Prog, Data);
end. {Prog2}
```

Tukaj torej velja:

$$\text{Prog(Data) ne izpiše ničesar} \iff \text{KoncenIzpis(Prog2, [Prog, Data])}.$$

Če pa nas zanima, ali se dani program Prog ustavi ali zacikla, moramo pripraviti nek nov program (recimo Prog3), ki bo imel končen izpis v primeru, ko se bo Prog zaciklal, in neskončen izpis v primeru, ko se bo Prog ustavil. To lahko naredimo tako, da Prog poženemo in če oz. ko se ustavi, nekaj izpišemo; to ponavljamo v nedogled. Če se Prog ustavi, bo Prog3 v vsaki ponovitvi te zanke nekaj izpisal in bo imel zato neskončen izpis. Če se Prog zacikla s končnim izpisom, ne bo Prog3 sploh nikoli prišel do tega, da bi tudi sam nekaj izpisal,

[48] Lahko bi zastavili tudi obratno, torej da bi imel Prog2 končen izpis, ko bo imel Prog neprazen izpis, in neskončen izpis, ko bo imel Prog prazen izpis. To bi lahko dosegli tako, da bi Prog2 simuliral delovanje programa Prog (npr. kot interpreter ali emulator) in pri vsakem koraku nekaj izpisal. Če bi Prog v nekem trenutku nekaj izpisal, bi se Prog2 takoj ustavil (v tem primeru bi imel Prog2 očitno končen izpis); če pa bi se Prog ustavil, bi šel Prog2 v neskončno zanko, v kateri bi kar naprej nekaj izpisoval. Tretja možnost je še, da se Prog že sam zacikla in pri tem nikoli nič ne izpiše, to pa tudi že zagotavlja, da bo imel Prog2 v tem primeru neskončen izpis.

tako da bo skupni izpis še vedno končen. Nerodna je le tretja možnost, namreč da se Prog zacikla in ima pri tem neskončen izpis; v tem primeru je tudi izpis celotnega programa Prog3 neskončen. Na srečo pa lahko to možnost preverimo neposredno, če programu KoncenIzpis pokažemo program Prog.[49]

```
program Prog3(Prog, Data);
begin
  while true do begin
    Execute(Prog, Data);
    WriteLn('puf, puf');
  end; {while}
end. {Prog3}
```

Program Prog3(Prog, Data) da končen izpis $\Longleftrightarrow$
program Prog(Data) se zacikla in da končen izpis.
Prog(Data) se zacikla $\Longleftrightarrow$ KoncenIzpis(Prog3, [Prog, Data])
or not KoncenIzpis(Prog, Data)

Mimogrede, ni se težko prepričati, da programa KoncenIzpis sploh ni mogoče napisati. Recimo namreč, da bi tak program res imeli. Potem lahko napišemo program:

```
program Prog4(Prog);
begin
  if KoncenIzpis(Prog, [Prog]) then
    while true do WriteLn('tralala');
end. {Prog4}
```

Ali ima Prog4, če mu damo opis samega sebe kot vhodni podatek, končen izpis ali ne? Recimo, da ima končen izpis; torej klic KoncenIzpis v tretji vrstici vrne true; toda v tem primeru pade Prog4 v neskončno zanko in ima neskončen izpis. Torej je ta predpostavka napačna — imeti mora neskončen izpis; toda v tem primeru vrne KoncenIzpis vrednost false in Prog4 se takoj konča, ne da bi sploh kaj izpisal, tako da ima Prog4 končen izpis. Tako smo ugotovili, da ne more imeti Prog4, če mu pokažemo samega sebe kot vhodni podatek, niti končnega niti neskončnega izpisa, to pa je nemogoče, torej tudi ni mogoče, da bi obstajal nek program KoncenIzpis z lastnostmi, kot jih omenja naloga.[50]

[49] Lahko pripravimo tudi tak Prog3, ki bo imel končen izpis, če se Prog konča, in neskončen izpis, če se Prog zacikla. To bi lahko naredili tako, da bi Prog3 simuliral delovanje programa Prog na danih vhodnih podatkih Data in pri tem po vsakem izvršenem koraku nekaj izpisal. Če se Prog zacikla, bo imel Prog3 tako neskončen izpis, če se Prog ustavi, pa končnega. Zdaj bo odgovor na vprašanje, ali ima Prog3 končen izpis, ravno enak kot na vprašanje, ali se Prog ustavi.

[50] Več o stvareh, o katerih smo razmišljali pri tej nalogi, najdemo v učbenikih teorije izračunov (*theory of computation*), npr. M. Sipser: *Introduction to the Theory of Computation*, 1996; J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, 1979.

N: 265

R1996.3.3

Ko preberemo vse omejitve v pomnilnik (množica Omejitve v spodnjem programu), si za vsako spremenljivko S tudi zapomnimo, v katerih omejitvah se pojavlja (množica Kje[S]). V nadaljevanju bomo vzdrževali množico omejitev, ki jih je še treba uskladiti (Neusklajene); na začetku so to kar vse omejitve. Nato v vsakem koraku uskladimo eno od teh omejitev; če se pri tem kakšni od njenih spremenljivk zaloga vrednosti popolnoma izprazni, vemo, da vseh omejitev ne bo mogoče uskladiti, in lahko takoj nehamo. Drugače pa se lahko kakšni od spremenljivk zaloga vrednosti vsaj zmanjša in v tem primeru moramo ponovno pregledati omejitve, v katerih se ta spremenljivka pojavlja (kajti zaradi zmanjšanja njene zaloge vrednosti se zdaj lahko na novo zmanjša zaloga vrednosti še kakšni drugi spremenljivki, ki se pojavlja skupaj z njo v kakšni omejitvi), zato vse te dodamo v množico neusklajenih omejitev. Če se nam množica Neusklajene izprazni, to pomeni, da smo uspeli uskladiti vse omejitve, ne da bi se kakšni spremenljivki zaloga vrednosti zmanjšala.[51]

```
function Usklajenost: boolean;
begin
  Omejitve := {};
  za vsako spremenljivko S: Kje[S] := {};
  while not Eof(Input) do begin
    O := PreberiOmejitev;
    dodaj O v množico Omejitve;
    for i := 1 to ŠteviloSpr(O) do
      dodaj O v množico Kje[ImeSpr(O, i)];
  end; {while}
  Neusklajene := Omejitve;
  while Neusklajene ≠ {} do begin
    vzemi neko O iz množice Neusklajene;
    UskladiOmejitev(O);
    for i := 1 to ŠteviloSpr(O) do begin
      if PraznaZalogaVrednosti(O, i) then return false;
      if ZmanjšanaZalogaVrednosti(O, i) then
        Neusklajene := Neusklajene ∪ Kje[ImeSpr(O, i)];
    end; {for}
  end; {while}
  return true;
end; {Usklajenost}
```

V konkretni implementaciji bi množico Omejitve verjetno izvedli s seznamom, v vsakem od elementov tega seznama pa bi bila neka omejitev in še podatek

[51] Opisani postopek za usklajevanje množice omejitev in klestenje zalog vrednosti spremenljivk izvira s področja logičnega programiranja z omejitvami logičnega programiranja z omejitvami (Constraint Logic Programming — CLP). Za več o tem glej nalogo 1995.3.2, str. 217, rešitev na str. 234 in literaturo, navedeno v opombi na str. 237.

o tem, ali je ta omejitev trenutno v množici Neusklajene ali ne. Množico Kje[S] pa bi lahko predstavili s seznamom, v katerem bi bil vsak element kazalec na enega od elementov seznama Omejitve. Potem se je zelo poceni zapeljati po vseh omejitvah iz Kje[S], za vsako pa takoj vidimo, če je že v množici Neusklajene in če ni, vemo, da jo moramo tja dodati. Množica Neusklajene naj bo tudi seznam kazalcev na elemente seznama Omejitve, tako da lahko, ko vzamemo neko omejitev iz množice Neusklajene, hitro označimo (v seznamu Omejitve), da te omejitve ni več v tej množici.

Konkreten primer programa, ki vsebuje tudi postopek za usklajevanje omejitev, kakršen je bil opisan tule, si lahko ogledamo pri rešitvi naloge 1995.3.2 (str. 234).

R1996.3.4

N: 267

Če bi bilo vseh 64 bitov ključa zares naključno izbranih, bi vlomilec potreboval $2^{64}/(1000 \cdot 60 \cdot 60 \cdot 24 \cdot 365) \approx 585$ milijonov let za preizkus vseh ključev. V resnici vseh možnih ključev, ki jih lahko izbere naš program za šifriranje, ni toliko, saj program izbere začetno „naključno“ vrednost le iz trenutnega časa in vseh možnih minut v dnevu je le 1440. Ker funkcija Random vedno predvidljivo iz nekega števila izračuna novo število, 10000-kratno klicanje funkcije Random prav nič ne pripomore k naključnosti, saj vlomilec pozna naš program in lahko enak postopek ponovi. Prav tako zlaganje 64-bitnega števila iz štirih psevdonaključnih, ki so izpeljana eno iz drugega, nič ne pripomore k povečanju števila možnih ključev, saj lahko vlomilec isti postopek ponovi in dobi isti rezultat.

Kljub temu torej, da je na prvi pogled videti 64 bitov ključa precej naključnih, je vseh možnih različnih ključev, ki jih lahko izbere naš program, le 1440. Za preizkus vseh teh ključev bi vlomilec potreboval $1440/1000 = 1{,}44$ sekunde, pričakovani čas pa je polovico tega: 0,72 sekunde. Če vlomilec ve, kdaj približno smo besedilo šifirali, pa se pričakovani čas, potreben za ugibanje, še zmanjša.

Prvo izboljšavo bi lahko naredili pri izbiri začetnega „naključnega“ števila. Tega bi morali dobiti iz opazovanja zunanjih dogodkov, ki niso tako predvidljivi kot ura, npr. poslušanja šuma iz avdio kartice, merjenja časov med pritiski na tipke pri tipkanju nekega daljšega besedila, merjenju časov med prekinitvami (interrupts) diskovnega vmesnika med večjo aktivnostjo diska in podobno — najbolje kar iz več virov.[52] Če bi tako izbrali prvo 16-bitno število, ostala štiri pa še vedno izračunali eno iz drugega, bi bilo vseh možnih ključev $2^{16} = 65536$ in za preizkus potrebnih 65,5 sekund, kar ni poseben napredek. Torej se je treba v celoti odpovedati uporabi generatorja psevdonaključnih števil in z opisanim postopkom zbiranja šuma iz računalnikovega okolja zbrati vseh 64 bitov ključa.

Pripomnimo še, da veljajo podobna časovna razmerja tudi v primeru, če

[52] Glej tudi 4. nalogo za prvo skupino (str. 260).

vlomilec našega programa ne bi poznal, le več truda in računalniškega časa bi moral žrtvovati, pomagalo pa bi tudi, če bi uspel prestreči več šifriranih sporočil. Temelj sodobnega šifriranja ni v tajnosti šifrirnih algoritmov, ampak v tajnosti in neuganljivosti ključev.

REŠITEV NALOGE DRUGEGA ZAKLJUČNEGA TEKMOVANJA IZ ZNANJA RAČUNALNIŠTVA

N: 268

R1996.Z

Za začetek je pametno podatke o karticah v celoti prebrati v glavni pomnilnik, saj ga je dovolj. Če bi predelali kar naivno rešitev, bi morali za vsako kartico hraniti množico tipa **set of** 1..100, ki bi zasedala v pomnilniku $\lceil 100/8 \rceil = 13$ bytov, tako da bi vseh 100 000 kartic porabilo slabega 1,3 MB (spomnimo se, da imamo na voljo skoraj 16 MB pomnilnika). S tem bi prihranili že veliko časa, ker nam ne bi bilo treba po vsakem izžrebanem številu brati podatkov o karticah z diska. Poleg tega lahko to naredimo v času pred prvim žrebom, tako da se nam sploh ne bo štelo v čas, ki ga meri žirija.

Naivna rešitev primerja vsebino vsake kartice kar z množico doslej izžrebanih števil; če je razlika prazna množica, vemo, da je bila kartica izžrebana. (Obenem še preveri, da je številka, ki smo jo nazadnje izžrebali, napisana na kartici, saj nas zanimajo le tiste kartice, ki so se dopolnile šele ob zadnji izžrebani številki.) Naš program torej kar naprej (po vsaki izžrebani številki) računa razlike Karta − Izzrebane in si daje pri tem vsakič opraviti z vsemi stotimi biti (oz. vsemi trinajstimi byti) v teh dveh spremenljivkah, čeprav se bo rezultat v primerjavi s tistim pred zadnjim žrebanjem razlikoval le po tem, da v njem ne bo več pravkar izžrebane karte. Stavek

```
if (NovaSt in Karta) and (Karta − Izzrebane = []) then
  Zadetkov := Zadetkov + 1;
```

bi torej lahko zamenjali z

```
var Karta: array [1..StKartic] of set of 1..100;
{ ... }
if NovaSt in Karta[i] then begin
  Exclude(Karta[i], NovaSt);
  if Karta[i] = [] then Zadetkov := Zadetkov + 1;
end; {if}
```

To nam že prihrani nekaj časa, ker Exclude le izključi bit, ki predstavlja element, ki ga brišemo iz množice (torej ne gre skozi vseh sto bitov). Preverjanje, če je Karta[i] po novem prazna množica, pa je še vedno tako zahtevno kot prej, ker mora program za vseh trinajst bytov, ki jih ta spremenljivka zaseda v pomnilniku, preveriti, če v njih res ni noben bit prižgan. Zato je koristno nekje

za vsako kartico hraniti podatek o tem, koliko številk z nje je še neizžrebanih. Vsakič, ko zbrišemo neko število iz množice, bi ta števec zmanjšali in nato zelo poceni preverili, če je po novem enak 0. Na začetku bi bil ta števec enak 15 za vse kartice.

Še vedno pa po nepotrebnem tratimo čas, ko po vsakem izžrebanem številu pregledujemo *vse* kartice in za vsako preverjamo, če mogoče vsebuje pravkar izžrebano število; saj je vendar že vnaprej jasno, da nastopa posamezno število v povprečju le na $100\,000 \cdot 15/100 = 15\,000$ karticah, torej ostalih 85 000 pregledujemo brez koristi. Zato si bomo raje za vsako številko (od 1 do 100) zapomnili, na katerih karticah se pojavlja; potem se bomo morali po vsakem žrebu ubadati le s tistimi karticami, ki to številko zares vsebujejo. Zdaj pa lahko tudi ugotovimo, da množice Karta[i], ki naj bi za vsako karto povedala, katere številke so na njej še neizžrebane, sploh ne potrebujemo več (doslej smo jo namreč uporabljali le za preverjanje, ali neka kartica vsebuje pravkar izžrebano številko ali ne, po novem pa nam tega ne bo treba početi, saj bomo imeli za vsako številko seznam kartic, na katerih se pojavlja).

Kartice bi lahko preprosto oštevilčili (recimo od 0 do 99 999) in seznam kartic, na katerih se pojavlja določena številka, predstavili kar s tabelo 32-bitnih številk kartic. Vendar pa bi bila v tem primeru ta tabela lahko večja od 64 KB, saj nam nihče ne zagotavlja, da se ne pojavlja neka številka na zelo veliko karticah. Spodnji program se skuša temu izogniti tako, da kartice v mislih razdeli na skupine po tisoč kartic (odvisno od tega, koliko je res vseh kartic, imamo potem eno, deset ali pa sto takšnih skupin). Dodatna prednost tega je, da so zdaj številke kartic vedno od 0 do 999, tako da nam zanje zadostujejo 16-bitna števila.

Najprej zapišimo program za pripravo indeksnih datotek, v kateri bo za vsako številko pisalo, na katerih karticah se pojavlja. Naloga pravi, da si lahko pripravimo za največ 20 MB datotek, kar je za nas več kot dovolj: 111 000 kartic s po 15 številkami in še knjigovodski podatki (na koliko karticah se pojavlja vsaka številka; to je 111-krat po sto števil); ker uporabljamo 16-bitna števila, bo dovolj že 3 352 200 bytov. Dobljene indeksne datoteke bo kasneje uporabljal naš glavni program Tombola.

```
program PripraviIndeks;
type
  { V tabeli tipa T1000Words bo element 0 hranil število kartic
    (iz neke skupine 1000 kartic), na katerih je neka številka,
    naslednjih toliko elementov pa vsebuje indekse teh kartic. }
  P1000Words = ↑T1000Words;
  T1000Words = array [0..1000] of word;
var
  fIn: text;
  fOut: file;
  Buf: array [0..16383] of byte; { za hitrejše branje datoteke s karticami }
```

```
  Kartice: array [1..100] of P1000Words; { kartice trenutne skupine }
  i, j, k, N, StSkupin: integer;
begin
  if      ParamStr(1) = '1000' then StSkupin := 1
  else if ParamStr(1) = '10000' then StSkupin := 10
  else if ParamStr(1) = '100000' then StSkupin := 100
  else    Halt;
  Assign(fIn, 'k' + ParamStr(1) + '.txt');
  Reset(fIn); SetTextBuf(fIn, Buf, SizeOf(Buf));
  Assign(fOut, 'k' + ParamStr(1) + '.idx');
  Rewrite(fOut, 1);
  for i := 1 to 100 do New(Kartice[i]);
  for i := 0 to StSkupin - 1 do begin
    { Obdelajmo naslednjih tisoč kartic. }
    for j := 1 to 100 do Kartice[j]↑[0] := 0;
    for j := 0 to 999 do begin
      for k := 1 to 15 do begin
        Read(fIn, N);
        Kartice[N]↑[0] := Kartice[N]↑[0] + 1;
        Kartice[N]↑[Kartice[N]↑[0]] := j;
      end;
      ReadLn(fIn);
    end; {for j}
    { Shranimo podatke o teh tisoč karticah. }
    for j := 1 to 100 do BlockWrite(fOut,
      Kartice[j]↑, SizeOf(word) * (Kartice[j]↑[0] + 1));
  end; {for i}
  Close(fIn); Close(fOut);
  for i := 1 to 100 do Dispose(Kartice[i]);
end. {PripraviIndeks}
```

Tu pa je še program Tombola:

```
program Tombola;

uses Zirija;

type P1000Words = ↑T1000Words;
     T1000Words = array [0..1000] of word;
     P1000Bytes = ↑T1000Bytes;
     T1000Bytes = array [0..999] of byte;
var
  i, j, StSkupin, NovaSt: integer; W: word; N: byte;
  NStevil: array [0..99] of P1000Bytes; { št. neizžrebanih števil na karticah }
  { Katere kartice vsebujejo določeno število? Element Kartice[X, Y]↑[Z] = Q
    nam pove, da kartica št. 1000 * Y + Q vsebuje številko X.
    Pri tem lahko Z zavzame vrednosti od 1 do Kartice[X, Y]↑[0]. }
  Kartice: array [1..100, 0..99] of P1000Words;
```

```
  fIdx: file;
  Zadetkov: longint; P: P1000Words; PN: P1000Bytes;
begin
  Pripravi;
  { Ugotovi število kartic. }
  if      ParamStr(1) = '1000' then StSkupin := 1
  else if ParamStr(1) = '10000' then StSkupin := 10
  else if ParamStr(1) = '100000' then StSkupin := 100
  else    Halt;
  { Preberi podatke o karticah. }
  Assign(fIdx, 'k' + ParamStr(1) + '.idx');
  Reset(fIdx, 1);
  for i := 0 to StSkupin - 1 do begin
    for j := 1 to 100 do begin
      BlockRead(fIdx, W, SizeOf(W)); { Število kartic, na katerih je številka j. }
      GetMem(Kartice[j, i], SizeOf(Word) * (W + 1));
      Kartice[j, i]↑[0] := W;
      BlockRead(fIdx, Kartice[j, i]↑[1], W * SizeOf(Word));
    end; {for j}
    New(NStevil[i]);
    { Na vseh karticah je še 15 neizžrebanih števil. }
    for j := 0 to 999 do NStevil[i]↑[j] := 15;
  end; {for i}
  Close(fIdx);
  { Žrebaj. }
  while Zrebaj(NovaSt) do begin
    Zadetkov := 0;
    for i := 0 to StSkupin - 1 do begin
      P := Kartice[NovaSt, i]; PN := NStevil[i];
      for j := 1 to P↑[0] do begin
        W := P↑[j]; { Številka kartice. }
        N := PN↑[W] - 1; { Novo št. še neizžrebanih številk na tej kartici. }
        PN↑[W] := N;
        if (N = 0) then Zadetkov := Zadetkov + 1;
      end; {for j};
    end; {for i};
    Zapisi(Zadetkov);
  end; {while}
  { Pospravi. }
  for i := 0 to StSkupin - 1 do begin
    for j := 1 to 100 do
      FreeMem(Kartice[j, i], SizeOf(Word) * (Kartice[j, i]↑[0] + 1));
    Dispose(NStevil[i]);
  end; {for}
  Pospravi;
end. {Tombola}
```

21. državno tekmovanje v znanju računalništva (1997)

NALOGE ZA PRVO SKUPINO

R: 303

1997.1.1

Člani komisije računalniškega tekmovanja srednješolcev so pripravili podprogram Uredi za urejanje doseženih rezultatov in trdijo, da deluje.

```
const n = ...;
var a: array [0..n] of integer;

procedure Uredi;
var i, j: integer;
begin
  for i := 2 to n do begin
    j := i;
    a[0] := a[j];
    while (j > 1) and (a[j − 1] > a[0]) do begin
      a[j] := a[j − 1];
      j := j − 1;
    end;
    a[j] := a[0];
  end;
end; {Uredi}
```

Opiši osnovno idejo urejanja, ki je uporabljena v podprogramu Uredi. **Ugotovi**, ali je podpogoj (j > 1) v zanki **while** zares potreben, se pravi, ali so bili člani komisije pri sestavljanju podprograma Uredi preveč prizadevni in smejo prepisati pogoj v zanki

```
while (j > 1) and (a[j − 1] > a[0]) do begin
  a[j] := a[j − 1];
  j := j − 1;
end;
```

v pogoj

```
while a[j − 1] > a[0] do begin
  a[j] := a[j − 1];
  j := j − 1;
end;
```

1997.1.2

V podani tabeli Tabela je Dolz števil ($0 \leq \text{Dolz} \leq 100$), preostanek tabele pa je neuporabljen. R: 303

```
var Tabela: array [1..100] of integer;
    Dolz: integer;
```

Napiši program, ki tabelo spremeni tako, da v primeru zaporednih enakih števil odstrani iz tabele vse zaporedne kopije. Na koncu mora biti v spremenljivki Dolz shranjena nova dolžina tabele.

Primer:

pred brisanjem:	$3, 5, 9, 9, 1, 1, 1, 2, 3, 9, 9, 9$	$\text{Dolz} = 12$
po brisanju kopij:	$3, 5, 9, 1, 2, 3, 9$	$\text{Dolz} = 7$

1997.1.3

Pri zapisovanju slovarja slovenskega jezika na CD-ROM (speštanka) je dr. Ostropišič opazil, da ima veliko število besed enake končnice. Odločil se je, da bo izkoristil to lastnost in s primernim kodiranjem prihranil nekaj prostora. R: 303

Kot jezikoslovec sam tega seveda ne zmore, zato mu pomagaj. **Opiši postopek**, ki bo v seznamu besed poiskal takšno končnico besed, pri kateri bo zmnožek dolžine končnice in števila ponovitev besed, katerim je ta končnica skupna, največji. Če je takih končnic več, poišči katerokoli izmed njih.

Primer:

```
  BOLAN
 BONBON      Končnica    N se ponovi 5-krat, zmnožek je 5.
  SALON      Končnica   ON se ponovi 4-krat, zmnožek je 8.
 ZAKLON      Končnica  LON se ponovi 3-krat, zmnožek je 9.
PRIKLON      Končnica KLON se ponovi 2-krat, zmnožek je 8.
  SKLON
```

Pozor: končnica ON se *pojavi* petkrat, a se *ponovi* le štirikrat. V zgornjem primeru je pravilni odgovor LON.

Seznam besed je zapisan v tabeli tako, da je v vsakem elementu po ena beseda. V tabeli je veliko število (> 100) kratkih besed (< 10 znakov). Predpostavi, da lahko vse besede shraniš v pomnilnik. Besede so že urejene po abecednem redu, vendar od konca besede proti začetku, kot to kaže tudi zgornji primer.

1997.1.4

Ponudniki dostopa do Interneta (npr. slovenski ARNES in ameriški America On-Line), se vedno znova srečujejo s pomanjkanjem modemskih vstopnih linij. Izkušnje ameriških ponudnikov kažejo, da razmerje med številom uporabnikov in številom modemov ne sme presegati R: 306

deset uporabnikov na en modem. Kadar je to razmerje preseženo, so modemi neprestano zasedeni.

Ponudnik dostopa do Interneta BUTALE BBS ima na voljo N modemov. Razmerje med številom uporabnikov in številom modemov je približno $42 : 1$.

Napiši program, ki nadzoruje uporabo modemov tako, da vedno obdrži en modem prost. V primeru, ko bi moral zasesti zadnji prosti modem, program sprosti tisti modem, ki je bil do takrat najdlje zaseden. Modemi so oštevilčeni od 1 do N. Na razpolago imaš podprogram Zaseden(i), ki pove, koliko sekund je modem i že zaseden. Če je modem prost, podprogram vrne 0. Modem i sprostimo s klicem podprograma Sprosti(i).

Ali je ta rešitev primerna? Kakšne težave lahko predvidiš? Odgovor utemelji in predlagaj boljšo rešitev.

NALOGE ZA DRUGO SKUPINO

R: 308

1997.2.1

Programer Jaša je našel program, ki ga je napisal pred davnimi leti. Žal se ne spomni, kaj je hotel z njim reševati. Spomni pa se, da z njim nekaj ni bilo v redu. Seveda mu boš pomagal ti.

(a) Razloži, **kaj dela ta program**.

(b) Skrajšaj program.

(c) Za katere vhodne podatke program deluje nepravilno?

(d) Navedi primer vhodnih podatkov, za katere program izpiše dvakrat 42.

Naj samo še spomnimo, da operator $\sim$ v C-ju oz. **not** v pascalu obrne vse bite v dvojiški predstavitvi števil.

```c
#include <stdio.h>
int main()
{
  int n1, n2, c, d, e, f;
  printf("Vnesi 1. število:");
  scanf("%d", &n1);
  printf("Vnesi 2. število:");
  scanf("%d", &n2);
  c = d = 0;
  e = f = 1;
  if (n1 < 0) {
    n1 = -n1;
    e = f = -1;
  }
  if (n2 < 0) {
    n2 = -n2;
```

```
program Main(Input, Output);
var
  n1, n2, c, d, e, f: integer;
begin
  Write('Vnesi 1. število: ');
  ReadLn(n1);
  Write('Vnesi 2. število: ');
  ReadLn(n2);
  c := 0; d := 0;
  e := 1; f := 1;
  if n1 < 0 then begin
    n1 := -n1; e := -1; f := -1;
  end;
  if n2 < 0 then begin
```

```
    e = -e;
  }
  do {
    if (c % 2)
      n1 = n1 - n2;
    else
      n1 = n1 + ~n2 + 1;
    c += 1;
  } while (n1 >= n2);
  if (n1 < 0) d = 1;
  printf("%d %d\n",
    (c - d) * e, (n1 + n2 * d) * f);
}
```

```
    n2 := -n2; e := -e;
  end;
  repeat
    if Odd(c) then
      n1 := n1 - n2
    else
      n1 := n1 + not n2 + 1;
    c := c + 1;
  until n1 < n2;
  if n1 < 0 then d := 1;
  WriteLn((c - d) * e, ' ', (n1 + n2 * d) * f);
end.
```

1997.2.2

R: 310

S skrivanjem sporočil pred nepooblaščenimi pogledi se ukvarjata dve vedi: kriptografija in steganografija. Kriptografija poskuša zagotoviti takšno šifriranje dokumentov, da jih nasprotnik ne more prebrati, tudi če prestreže šifrirani dokument. Steganografija poskuša skriti vsebino dokumenta tako, da se nasprotnik sploh ne zaveda, da ima v rokah skriti dokument. V praksi se večkrat uporabljata obe metodi hkrati: šifrirani dokument še skrijemo.

Ena od možnosti za skrivanje dokumenta je, da ga skrijemo v nek drugi večji dokument, na primer v sliko ali v zvočni zapis. Pri tem večji dokument nekoliko pokvarimo, vendar na čim manj opazen način.

Vzemimo za večji dokument črno-belo sliko velikosti 256×256 točk (pikslov). Svetlost vsake točke je predstavljena s številom med 0 in 255. Če spremenimo najnižji bit, se svetlost točke spremeni kvečjemu za $1/256$, kar je manj kot 0,4 %. Tega s prostim očesom ne opazimo ali kvečjemu zaznamo kot povečan šum v sliki. V sliko lahko tako skrijemo $(256 \times 256)/8 = 8192$ osem-bitnih znakov.

```
const w = 256; h = 256;
var Slika: array [1..h, 1..w] of 0..255;
```

Napiši podprogram SkrijBesedilo, ki bo v sliko v tabelo Slika vpisal (oz. skril) besedilo, dolgo 8192 znakov, ter podprogram RazkrijBesedilo, ki bo iz slike izluščil skrito besedilo in ga izpisal.

Da ne zapletamo programa s kodiranjem koncev vrstic in se izognemo preverjanju konca vhodne datoteke, predpostavimo, da je v vhodni datoteki le ena vrstica besedila, dolga 8192 znakov.

1997.2.3

R: 311

Napiši funkcijo PrestejPodnize, ki mora vrniti število pojavitev niza PodNiz v nizu Niz. Štejejo tudi nestrnjene pojavitve, torej je lahko med črkami PodNiz-a v Niz-u tudi poljubno število drugih črk. Funkcija PrestejPodnize naj ima obliko

```
function PrestejPodnize(Niz, PodNiz: string): integer;
```

ali

```
int PrestejPodnize(char* Niz, char* PodNiz);
```

Primeri: v nizu `deafgahibjkclm` nastopajo podnizi: `abc` dvakrat (zaradi dveh `a`-jev), `bc` enkrat, `afg` enkrat in `ba` nobenkrat. V `deafgabhibjkclm` nastopa `abc` štirikrat, v `deafbgahibjkclm` pa trikrat. Niz `abcdefghij` nastopa v nizu `aabbccddeeffgghhiijj` 1024-krat.

R: 316

1997.2.4

Celota je razdeljena na N deležev, ki so realna števila med 0 in 1, njihova vsota pa je 1. Deleži so podani v tabeli:

```
const N = 20;    { N je poljuben, a ne zelo velik }
var Delez: array [1..N] of real;
```

Radi bi izpisali deleže v odstotkih, zaokrožene na cela števila. Pri tem nastane lepotni problem, saj se lahko zgodi, da vsota zaokroženih odstotkov ni točno 100.

Problemu se izognemo z „goljufanjem“ pri zaokrožanju. Odstotke zaokrožimo tako, da je skupna vsota natanko 100, skupna absolutna napaka pa čim manjša. Goljufajmo torej pri tistih deležih, ki se jim goljufija manj pozna.

Primer za $N = 5$:

delež · 100	zaokrožen		po popravku
30,9	31		31
10,4	10	$\longrightarrow$	11
20,3	20		20
20,2	20		20
18,2	18		18
100,0	99		100

Na primer, 10,4 lahko zaokrožimo na 10 ali na 11, kakor nam pač bolj ustreza. V prvem primeru je absolutna napaka 0,4, v drugem pa 0,6.

Napiši del programa, ki v tabeli podane deleže izpiše kot zaokrožene odstotke. Program naj poskrbi, da je vsota zaokroženih odstotkov 100, skupna absolutna napaka pa najmanjša.

NALOGE ZA TRETJO SKUPINO

1997.3.1

R: 317

Ker se člani komisije računalniškega tekmovanja srednješolcev raje ukvarjajo s trapastimi podatkovnimi strukturami kot s sestavljanjem pametnih nalog, so skovali funkcijo DvojnaHitrost, sedaj pa ne vedo, kaj sploh dela. Ugotovi, **kaj vrne funkcija** DvojnaHitrost, in oceni, **kolikokrat** se v najslabšem primeru izvede telo zanke **repeat**, če seznam s vsebuje n elementov. Člani komisije ti bodo za pravilno rešitev naloge zares hvaležni.

```
type
  Seznam = ↑Vozel;
  Vozel = record
            Naslednji: Seznam;
            ... podatki ...
          end;

function DvojnaHitrost(S: Seznam): boolean;
var p1, p2: Seznam;
begin
  p1 := S;
  p2 := S;
  repeat
    if p1 <> nil then p1 := p1↑.Naslednji;
    if p2 <> nil then p2 := p2↑.Naslednji;
    if p2 <> nil then p2 := p2↑.Naslednji;
  until (p1 = nil) or (p2 = nil) or (p1 = p2);
  DvojnaHitrost := (p1 <> nil) and (p2 <> nil);
end; {DvojnaHitrost}
```

1997.3.2

R: 318

Novoizvoljeni župan Gropel kraja Kraljana je želel izvedeti, kateri ljudje v njegovem mestu so najpomembnejši. Nekako mu je v roke prišel seznam meščanov (v datoteki `ligenj.doc`), ki so med seboj prijatelji.

Župan je sklepal takole: moč meščana je enaka številu njegovih prijateljev. Začetna pomembnost vsakega meščana je ena. Pomembnost meščanov se prenaša z enega na drugega tako, da vsak meščan, ki ima močnejše prijatelje, svojo pomembnost enakomerno razdeli mednje.[53] Njegova pomembnost pri tem postane nič. Najpomembnejše meščane dobimo, ko prenašanje pomembnosti med meščani ni več možno.

[53] Pozor: „moč“ in „pomembnost“ sta tu dva popolnoma ločena pojma. Pomembnost se prenaša med meščani, moč pa ostaja ves čas nespremenjena. Moč samo vpliva na to, kako se prenaša pomembnost.

Napiši algoritem, ki čim učinkoviteje določi pomembnosti meščanov in izpiše številke meščanov z neničelno pomembnostjo. Pri tem imaš na voljo naslednje funkcije in podprograme:

- **function** PreberiMescane: integer;
  Prebere podatke o meščanih in vrne njihovo število.
- **procedure** Tekoci(m: integer);
  Meščan m postane tekoči meščan.
- **function** Naslednji: integer;
  Vrne naslednjega prijatelja tekočega meščana ali 0, če so vsi pregledani.

Kaj veš povedati o rešitvah naloge?

R: 320

1997.3.3

Program BIOS (Basic Input/Output System) nekega računalnika za shranjevanje sistemskih nastavitev uporablja pomnilniški čip tipa *Flash* ROM. Tovrstni čipi obdržijo vpisano informacijo tudi po izklopu napajanja. Sprva je čip prazen in na njem so zapisane same ničle. V čip lahko vpisujemo le enice. Brisanje (vpisovanje ničel) je sicer možno, a dolgotrajno in skrajša življenjsko dobo čipa, zato ga ne bomo uporabljali.

BIOSu je čip predstavljen kot niz podatkovnih besed:

```
const MaxFlashROM = ...;
var FlashROM: array [1..MaxFlashROM] of integer;
```

BIOS vpisuje cela števila v čip s podprogramom Vpisi, pri čemer se vpišejo samo enice:

```
procedure Vpisi(Naslov, Podatek: integer);
begin
  FlashROM[Naslov] := FlashROM[Naslov] or Podatek;
end; {Vpisi}
```

Ob vsakem vpisovanju zapiše BIOS na čip najprej velikost zapisa (N), temu pa sledi $N-1$ besed (integerjev) podatkov. Zapisi se nizajo eden za drugim, zadnjemu zapisu pa sledi nepopisano področje samih ničel.

Ob vsakem zagonu sistema se zagonski program sprehodi prek vseh zapisov do zadnjega in ga prebere. Število zapisov se med delovanjem sistema veča, zato program za iskanje zadnjega zapisa porabi vse več časa.

Vaša naloga je, da **izdelate učinkovit algoritem**, ki poskuša podatke v čipu Flash ROM spremeniti tako, da bo zagonski program čim hitreje našel zadnji zapis. Pri tem sme spreminjati samo informacijo o velikosti posameznih zapisov.

1997.3.4

R: 321

Podjetje DOMAČA PAMET—TUJE IDEJE je ugotovilo, da njihovi zaposleni pogosto zahtevajo ene in iste spletne strani, ki se vedno znova prenašajo prek njihove povezave.

Zato so se odločili, da bodo naredili strežnik, ki bo zahteve prestregel, pogledal, če je stran že bila zahtevana, in jo v tem primeru podal kar iz vmesnega pomnilnika na disku. Najeli so te, da **napišeš dva podprograma**:

Zahtevek(Naslov: string; Tok: integer), ki bo poklican vsakič, ko bo kateri od uporabnikov zahteval stran, in

Prispelo(Podatki: string; Tok: integer), ki bo poklican vsakič, ko preko linije iz Interneta pridejo podatki za zahtevo, povezano s tem tokom. Če je podatkov konec (sprejeti so bili vsi podatki), bo Dolzina(Podatki) enako 0, sicer bo Dolzina(Podatki) $>$ 0.

Vsi prenosi podatkov potekajo v tokovih. Vsak tok je označen s pozitivnim celim številom. Iz vmesnega pomnilnika bodo na tok vedno prišli vsi podatki iz zahteve (ali pa nič), na druge tokove pa lahko podatki prihajajo „po kapljah“, vsakič po nekaj zlogov (byteov). Tip string je zagotovljeno dovolj velik za vsako zahtevo.

Na voljo imaš naslednje funkcije:

Zahtevaj(Naslov: string): integer — Sproži zahtevo in vrne številko toka.

Poslji(Podatki: string; Tok: integer) — Na tok Tok pošlje podatke.

Shranjen(Naslov: string; **var** Podatki: string): integer — Če so podatki v vmesnem pomnilniku, vrne 0 in podatke v parametru Podatki. Če podatkov v vmesnem pomnilniku ni, vrne številko toka, na katerega naj pošljemo podatke, da se bodo shranili v vmesni pomnilnik.

Vedeti je treba, da bo pri polni obremenitvi strežnik moral ustreči nekaj tisoč zahtevam na uro.

TRETJE ZAKLJUČNO TEKMOVANJE V ZNANJU RAČUNALNIŠTVA

Navodila za obnašanje med tekmovanjem

Naloga tekmovalcev je napisati tri programe, ki iz danih vhodnih podatkov, zapisanih v vhodni datoteki `input.txt`, za dano nalogo izračunajo pravilen rezultat in ga zapišejo v datoteko `output.txt`. Čas reševanja bo 5 ur, pred začetkom reševanja pa bo imel vsak tekmovalec vsaj 15 minut časa za prilagajanje delovnega okolja za računalnikom.

Na vsakem računalniku se nahaja direktorij `c:\rtk\`, ki vsebuje poddirektorija `borlandc` (Borland C/C++ 3.1) in `bp` (Borland Pascal 7.0) in ukazno

datoteko `rtk.bat`, ki nastavi poti za zagon prevajalnikov. Po zagonu `rtk.bat` se Borland C požene z ukazom `bc`, Borland Pascal pa z `bp`.

Tekmovalci smejo delati le na direktoriju `c:\rtk\` in nižje, nikakor pa ne smejo brati in pisati datotek drugje, ker gre za računalnike, na katerih baje teče izobraževalni proces.

Med tekmovanjem niso dovoljeni pogovori med tekmovalci. Tekmovalci ne smejo imeti knjig ali disket. Ni dovoljeno goljufati na vse mogoče standarne in nestandardne načine. Tekmovalcem ni dovoljeno uporabljati drugih funkcij računalnika (mreže, programov, ipd.) razen tistih, ki so predvidene (DOS okno na direktoriju `c:\rtk\` in programa `bc` in `bp`). Glasnost tipkanja naj ne presega 40 dB.

Vsako nepravilnost bodo člani komisije kaznovali s takojšnjo ali naknadno diskvalifikacijo tekmovalca.

Navodila za oddajo rezultatov

Tekmovalci morajo rezultate zapisati na dve disketi, ki ju pred uradnim koncem tekmovanja oddajo komisiji. Po uradnem koncu komisija ne bo več sprejemala disket. Na vsaki disketi mora biti napisano ime tekmovalca.

Na disketi za oddajo komisiji se morajo nahajati 3 datoteke:

- `ceste.exe` — rešitev prve naloge,
- `tocke.exe` — rešitev druge naloge,
- `km.exe` — rešitev tretje naloge

in izvorna koda programov!

Naj še enkrat spomnimo, da morajo vsi trije programi brati podatke iz datoteke `input.txt`, rezultate pa morajo napisati v datoteko `output.txt`.

Ocenjevanje rešitev

Tekmovanju bo sledilo javno ocenjevanje, med katerim bo komisija vsako rešitev preverila z 10 različnimi vhodnimi datotekami (`input.txt`). Uspešna rešitev za dano vhodno datoteko bo tista, ki bo v 10 sekundah vrnila pravilen rezultat.

Naloge

R: 323

1997.Z.1

V deželi *Transalpeniji* gradijo cestno omrežje, ki naj bi povezovalo vsa deželna mesta. **Napiši program**, ki ugotovi, koliko medsebojno še nepovezanih skupin mest je v deželi, in rezultat izpiše.

Vsakemu mestu pripišemo število med vključno 1 in M. Neposredne povezave med njimi so podane v obliki parov (i, j), kar pomeni, da je mesto i povezano z mestom j. Povezave so dvosmerne: če je mesto i povezano z mestom j, je tudi mesto j povezano z mestom i.

Dve mesti sta povezani, če obstaja med njima zaporedje povezav, ki vodi iz enega v drugo mesto. Skupina mest ni povezana z drugo skupino mest, če nobeno mesto iz prve skupine ni povezano z nobenim mestom iz druge skupine.

Podatki o povezavah med mesti se nahajajo v datoteki `input.txt`, kjer se v prvi vrstici nahaja število M, v drugi vrstici število N, ki predstavlja število vseh neposrednih povezav med mesti, v nadaljnjih N vrsticah pa se nahajajo pari števil, ki predstavljajo neposredne povezave med mesti. Povezave se lahko ponovijo. Povezava lahko povezuje mesto s samim seboj.[54]

Program mora v datoteko `output.txt` izpisati število medsebojno nepovezanih skupin mest. V skupini mest mora biti vedno vsaj eno mesto.

Primer povezane skupine treh mest v datoteki `input.txt`:

```
3
3
1 2
2 3
3 1
```

Pravilni odgovor v datoteki `output.txt` je `1`.

Primer nepovezane skupine štirih mest v datoteki `input.txt`:

```
4
2
1 2
3 4
```

Pravilni odgovor v datoteki `output.txt` je `2`.

1997.Z.2

Na ravnini je danih n točk. Vsaka točka je podana s parom koordinat (x, y). **Napišite program**, ki izračuna največje število točk, ki ležijo v nekem pravokotniku velikosti $a \times b$, pri čemer rotiranje pravokotnika ni dovoljeno (stranici pravokotnika morata torej biti vzporedni koordinatnima osema). R: 325

Program naj iz datoteke `input.txt` prebere vhodne podatke po naslednjem zaporedju:

$a\ b$
n
$x_1\ y_1$
$x_2\ y_2$
. . . . .
$x_n\ y_n$

Predpostavite, da število točk, torej n, ne bo večje od 10 000. Kot rezultat naj program izpiše v datoteko `output.txt` največje število točk, ki ležijo v pravokotniku velikosti $a \times b$.

[54] V prvotnem besedilu naloge ni bilo podane nobene omejitve glede velikosti števil M in N, kar ni ravno lepo. Pogled na testne primere, na katerih so preizkušali rešitve tekmovalcev, pokaže, da sta bila M in N vedno med vključno 1 in 30 000. Poleg tega se v dveh od desetih testnih primerov pojavlja kot številka mesta tudi 0; tedaj so torej mesta oštevilčena od 0 do M, drugače pa od 1 do M.

Primer vhodne datoteke:

```
3.0 2.0
6
0.0 0.0
1.0 0.0
1.0 1.0
4.0 1.0
0.0 2.0
1.0 3.0
```

Pripadajoča izhodna datoteka:

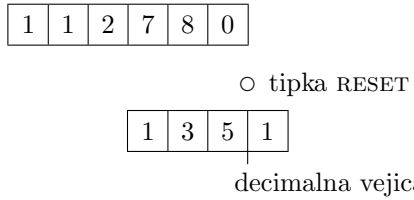
```
4
```

R: 336

1997.Z.3

Borut se pogosto vozi z avtomobilom. Čeprav je vzoren voznik, mu pogled vendarle pogosto zaide na števec kilometrov. Pri tem si je zamislil svojevrsten problem, ki pa ga ni znal rešiti, zato vas prosi za pomoč.

Števec je sestavljen iz dveh delov (velikega in malega števca). Veliki števec je sestavljen iz šestih števk in kaže skupne prevožene kilometre avtomobila. Mali števec ima 4 števke, pri čemer zadnja kaže desetine kilometra. Mali števec lahko v poljubnem trenutku postavimo na 000,0 s pritiskom na tipko RESET.



Naloga je **narediti program**, ki zna pri poljubnem stanju obeh števcev doseči stanje, kjer je vseh 10 števk obeh števcev med seboj različnih, tako da prevozimo čim manj kilometrov. Pri tem lahko poljubnokrat pritisnemo tipko RESET.

Vhodna podatka sta začetno stanje velikega in malega števca in sta zapisana v datoteki `input.txt` vsak v svoji vrstici. Primer:

```
000000
000.0
```

Privzemimo, da sta oba števca ravnokar dosegla svojo vrednost, tako da moramo prevoziti natanko 1 km, da se obrne veliki števec, oziroma 0,1 km, da se obrne mali.

Rezultat naj bo število kilometrov na velikem števcu, ko bo prvič mogoče doseči (lahko tudi takoj), da so vse števke različne. Izpiše naj se v datoteko `output.txt`. Rezultat za gornji primer je:

```
012345
```

REŠITVE NALOG ZA PRVO SKUPINO

R1997.1.1

N: 292

Podprogram Uredi uredi n elementov tabele a na intervalu 1..n z algoritmom navadnega vstavljanja s čuvajem. To pomeni, da v vsaki ponovitvi zanke **for** vstavi vrednost, ki je na začetku izvajanja procedure v elementu a[i], na pravo mesto med vrednosti a[1], a[2], ..., a[i − 1] in s tem poveča urejeni del tabele na interval a[1], a[2], ..., a[i]. Postopek se konča, ko vstavi vrednost elementa a[n] in s tem doseže urejenost cele tabele a.

Podpogoj j > 1 prvič ni izpolnjen v trenutku, ko velja j = 1. A ker vrednost elementa a[i], ki jo vstavljamo v že urejeni del tabele, med vstavljanjem hranimo v elementu a[0], tudi podpogoj a[j − 1] > a[0] pri j = 1 ni izpolnjen, zanka **while** pa se zato pri j = 1 gotovo ustavi. To pomeni, da lahko podpogoj j > 1 odstranimo.

R1997.1.2

N: 293

S števcem f se sprehajamo po tabeli, novo stanje tabele pa pri tem postopoma nastaja v prvih celicah tabele (Tabela[1..t]). Vsak element (razen prvega) primerjamo s prejšnjim elementom; če je enak, se zanj ne zmenimo, drugače pa ga dodamo na konec nove tabele, torej na mesto Tabela[t + 1]. Ker se f poveča v vsakem koraku, t pa le občasno, je f vedno večji ali enak t, tako da nam ni treba skrbeti, da bi si povozili kakšne podatke, ki jih še nismo prebrali.

```
var Tabela: array [1..100] of integer;
    Dolz: integer;

procedure OdstranjevanjeDuplikatov;
var f, t: integer;
begin
  if Dolz > 1 then begin
    t := 1;
    for f := 2 to Dolz do
      if Tabela[t] <> Tabela[f] then
        begin t := t + 1; Tabela[t] := Tabela[f] end;
    Dolz := t;
  end; {if}
end; {OdstranjevanjeDuplikatov}
```

R1997.1.3

N: 293

Problem lahko rešimo z dvema gnezdenima zankama. V zunanji zanki se premikamo po besedah, v notranji pa za besede od trenutne besede naprej gledamo, v koliko zadnjih črkah se ujemajo s trenutno. Ko pridemo do take, ki se s trenutno ne ujema niti v eni zadnji črki, lahko nehamo, saj so besede urejene po končnicah in tudi v bodoče ne bi našli nobene besede, ki bi se s trenutno ujemala v zadnjih nekaj črkah.

Še ena izboljšava je, da ne gledamo besed od trenutne naprej, ampak upoštevamo še, koliko besedam mora biti skupna neka na novo odkrita končnica, če hoče biti boljša od doslej najboljše znane. Na primer, če ima najboljša doslej znana končnica zmnožek 55, naše besede pa so dolge po največ deset znakov, bo vsaka končnica, ki je skupna vsaj dvema besedama, dolga največ devet znakov, torej mora biti skupna v resnici vsaj $\lceil 56/9 \rceil + 1 = 8$ besedam, če naj ima sploh kaj možnosti, da postane boljša od najboljše možne. Torej ni potrebno primerjati trenutne besede kar takoj z naslednjo, temveč jih lahko šest preskočimo in šele sedmo naslednjo primerjamo s trenutno. (Nobena končnica, ki bi bila skupna manj kot toliko besedam, namreč nima možnosti dobiti zmnožka nad 55.) Če imata tidve skupno neko končnico, imajo tudi vse med njima to končnico, saj so besede urejene po končnicah.

```
const n = ...; MaxDolz = 10;
type Tabela = array [1..n] of string;

function Koncnica(s: Tabela): string;
var i, j, k, Zmnozek, NajZmnozek: integer;
begin
  NajZmnozek := 0; Koncnica := s[1];
  for i := 1 to n do begin
    j := i + ((NajZmnozek + 1) + (MaxDolz - 1) - 1) div (MaxDolz - 1);
    if j > n then break; { odslej bi vedno imeli j > n }
    while j <= n do begin
      k := 0;
      while (k < Length(s[i])) and (k < Length(s[j])) do
        if s[i, Length(s[i]) - k] = s[j, Length(s[j]) - k] then k := k + 1
        else break;
      if k = 0 then { Besede od s[j] naprej nimajo s s[i] skupne }
        break;      { nikakršne končnice več. Takoj se lahko lotimo naslednjega i. }
      { Besedam s[i..j] je skupna končnica dolžine k. }
      Zmnozek := k * (j - i);
      if Zmnozek > NajZmnozek then begin
        { To je najboljša doslej znana končnica — zapomnimo si jo. }
        Koncnica := Copy(s[i], Length(s[i]) - k + 1, k);
        NajZmnozek := Zmnozek;
      end; {if}
      j := j + 1;
    end; {while}
  end; {for}
end; {Koncnica}
```

Slabost tega postopka je, da je lahko število parov besed (torej število parov (i, j)), ki jih moramo pregledati, če imamo smolo, sorazmerno s kvadratom števila vseh besed. Zato je časovna zahtevnost tega postopka $O(n^2)$, kar je lahko neugodno, če je besed veliko. Primer neugodnega zaporedja besed je na

primer zaporedje vseh $n = 2^m$ besed, dolgih po m znakov in sestavljenih iz samih črk a in b. (Za $m = 3$ bi dobili *aaa, baa, aba, bba, aab, bab, abb, bbb.*) Končnica dolžine k je skupna 2^{m-k} besedam in zato dobi zmnožek $(2^{m-k}-1)\cdot k$. Najboljša je zato kar končnica dolžine 1,[55] ki bi jo naš program opazil že pri $i = 1$. Odtlej je torej NajZmnozek enak $2^{m-1} - 1$, zato pri vsakem naslednjem i preizkušamo vrednosti j od $i + \lceil 2^{m-1}/(m-1) \rceil \le i + 1 + 2^{m-1}/(m-1)$ naprej. Dokler je i v prvi polovici tabele, torej $i \le 2^{m-1}$, bo šla notranja zanka do $j = 2^{m-1} + 1$ (do prve besede, ki se konča na b namesto na a). Za dovolj velike m, na primer $m \ge 5$, je $2^{m-1}/(m-1) \le 2^{m-1}/4 = 2^m/8$. Dokler je i še v prvi osmini tabele ($i \le 2^m/8$), mora j torej obiskati vsaj vse nize od $i + 1 + 2^m/8 \le 2^m/4 + 1$ do konca prve polovice zaporedja, torej do vključno $2^m/2$. To pa je vsaj $2^m/4$ nizov in to se nam zgodi pri vsaj $2^m/8$ vrednostih i (tudi pri $i = 1$, saj tam obiščemo celo vseh 2^m nizov). Že zaradi tega je število izvedb prireditve k := 0 vsaj $(2^m/4)(2^m/8) = n^2/32$. Časovna zahtevnost tega programa je torej vsaj $O(n^2)$.

Reševanja tega problema se lahko lotimo tudi drugače in bolj učinkovito. Recimo, da imamo besede že urejene po abecedi (od konca besede naprej) v neki tabeli $s[1..n]$ in da za besedo $s[i]$ vemo, da ima zadnjih j črk skupnih z besedami $s[i-1], \ldots, s[i-g[i,j]+1]$, ne pa tudi z besedo $s[i-g[i,j]]$. Potem so za vsak j (od 1 do dolžine besede $s[i]$) zmnožki $j \cdot (g[i,j]-1)$ vsekakor kandidati za najboljši zmnožek, ki ga moramo pri naši nalogi poiskati. Če torej izračunamo $g[i,j]$ za vse i in j, ne bo težko poiskati najboljšega zmnožka. Lepo pa je to, da lahko $g[i,j]$ učinkovito računamo, če že poznamo $g[i-1,j]$: če se besedi $s[i]$ in $s[i-1]$ ujemata v zadnjih j črkah, je $g[i,j] = g[i-1,j]+1$, drugače pa je $g[i,j] = 1$ (ta možnost obvelja tudi v primeru, če je $s[i-1]$ krajša od j črk, in v primeru, ko je $i = 1$). Zato tudi ni treba hraniti tabele $g[i,j]$ za vse vrednosti i, pač pa le za trenutni i in za $i-1$ (medtem ko jo za i še računamo), prejšnje pa lahko sproti pozabljamo. Najboljši doslej najdeni zmnožek si zapomnimo kot b, indeks niza, kjer se ta končnica pojavlja, v bi, njeno dolžino pa v bj.

```
const n = ...; MaxDolz = 10;
type Tabela = array [1..n] of string;

function Koncnica2(s: Tabela): string;
var
  g: array [1..MaxDolz] of integer;
  i, j, b, bi, bj, c, Dolz, PrejDolz: integer;
begin
  b := 0; bi := 1; bj := 0; Dolz := Length(s[1]);
  for j := 1 to Dolz do g[j] := 1;
```

[55] Zakaj je najboljša prav ta končnica? Če se k poveča za 1, se v zmnožku $(2^{m-k}-1)\cdot k$ prvi faktor zmanjša na manj kot polovico, drugi pa se največ podvoji, verjetno pa se poveča manj kot za toliko. Zato se celoten zmnožek zmanjšuje, če večamo k.

```
  for i := 2 to n do begin
    PrejDolz := Dolz; Dolz := Length(s[i]); j := 0;
    while (j < PrejDolz) and (j < Dolz) do
      if s[i, Dolz − j] = s[i − 1, PrejDolz − j] then begin
        j := j + 1; g[j] := g[j] + 1;
        c := j * (g[j] − 1);
        if c > b then { nova najboljša končnica }
          begin b := c; bi := i; bj := j end;
      end
      else break; { konec ujemanja }
    while j < Dolz do
      begin j := j + 1; g[j] := 1 end;
  end; {for}
  Koncnica2 := Copy(s[bi], Length(s[bi]) − bj + 1, bj);
end; {Koncnica2}
```

Lepo pri tej rešitvi je, da je količina dela, ki ga imamo, v najslabšem primeru sorazmerna s skupno dolžino vseh nizov, s katerimi moramo delati. Bolje kot to že skoraj ne bi moglo biti, saj je tudi čas, potreben za branje nizov, sorazmeren s skupno dolžino vseh nizov.

N: 293

R1997.1.4

Problem lahko rešimo z naslednjim podprogramom, ki bi ga moral sistem poklicati vsakič, ko se nek modem na novo zasede. Podprogram gre po vseh modemih in ugotavlja, kateri je že najdlje zaseden; na koncu tistega pač sprosti.

```
const N = ...; { Število vseh modemov. }

procedure KlicanObZasedbi;
var VsiZasedeni: boolean; i, T, KateriNajdlje, KolikoNajdlje: integer;
begin
  VsiZasedeni := true; i := 1;
  while VsiZasedeni and (i <= N) do begin
    T := Zaseden(i);
    if T = 0 then
      VsiZasedeni := false
    else if (i = 1) or (T > KolikoNajdlje) then
      begin KateriNajdlje := i; KolikoNajdlje := T end;
    i := i + 1;
  end; {while}
  if VsiZasedeni then Sprosti(KateriNajdlje);
end; {KlicanObZasedbi}
```

Če pa bi nas sistem obveščal tudi o tem, kdaj se nek modem iz kakršnih koli razlogov sprosti, bi lahko vzdrževali seznam zasedenih modemov, urejenih recimo padajoče po trajanju zasedenosti. Potem bi lahko zelo hitro ugotovili,

kateri je že najdlje zaseden: ni nam treba z zanko iti po vseh modemih, ampak vemo, da je najdlje zaseden kar tisti na začetku seznama.

```
const N = ...; { Število vseh modemov. }
var
  { Spremenljivka Prvi pove, kateri modem je že najdlje zaseden,
    Zadnji pa, kateri je zaseden najmanj časa. Če ni zaseden
    noben modem, sta Prvi in Zadnji enaka 0. }
  Prvi, Zadnji, StZasedenih: integer;
  { Prej[i] je številka modema, ki je v seznamu zasedenih neposredno pred
    modemom i; podobno je Nasl[i] številka tistega neposredno za i.
    Če i ni zaseden, sta v Prej[i] in Nasl[i] pač neki nesmiselni vrednosti.
    Na koncih seznama velja: Prej[Prvi] = 0 in Nasl[Zadnji] = 0. }
  Prej, Nasl: array [1..N] of integer;

procedure Inicializacija;
begin
  { Seznam zasedenih modemov je na začetku prazen. }
  Prvi := 0; Zadnji := 0; StZasedenih := 0;
end; {Inicializacija}

procedure KlicanObZasedbi(Modem: integer);
begin
  { Dodajmo novi modem na konec seznama. }
  Prej[Modem] := Zadnji; Nasl[Modem] := 0;
  if Zadnji = 0 then Prvi := Modem { seznam je bil prej prazen }
  else Nasl[Zadnji] := Modem;      { novi modem je naslednik bivšega zadnjega }
  Zadnji := Modem;                 { novi modem je zdaj po novem pač zadnji }
  { Po potrebi sprostimo najdlje zasedeni modem. }
  StZasedenih := StZasedenih + 1;
  if StZasedenih = N then Sprosti(Prvi);
end; {KlicanObZasedbi}

procedure KlicanObSprostitvi(Modem: integer);
begin
  StZasedenih := StZasedenih - 1;
  { Zbrišimo ta modem iz seznama. Predhodnik in naslednik, ki sta prej kazala
    na ta modem, morata po novem kazati drug na drugega. Posebej obravnavamo
    primere, ko je zbrisani modem prvi in/ali zadnji v seznamu. }
  if Prej[Modem] = 0 then Prvi := Nasl[Modem]
                     else Nasl[Prej[Modem]] := Nasl[Modem];
  if Nasl[Modem] = 0 then Zadnji := Prej[Modem]
                     else Prej[Nasl[Modem]] := Prej[Modem];
end; {KlicanObSprostitvi}
```

Slabost predlaganega pristopa k reševanju prezasedenosti modemov se pokaže v primerih, ko uporabniki na veliko oblegajo modeme. Ko pokličemo, sicer

takoj dobimo zvezo, vendar vsak naslednji klicatelj prekine zvezo enemu izmed tistih uporabnikov, ki so povezani že dlje kot mi, tako da po N klicih pridemo na vrsto mi in nam sistem prekine zvezo. Tako je vsak uporabnik mogoče povezan le nekaj sekund in v tem času najbrž še ne more narediti ničesar uporabnega. Če se uporabniki v takem primeru poskušajo povezati znova in znova, se stanje le še poslabša.

Boljša rešitev bi verjetno bila, če ne bi vztrajali na tem, da mora biti en modem vedno prost. Lahko bi se na primer odločili, da uporabniku ne bomo prekinili zveze, če ni povezan že vsaj nekaj časa (recimo x minut). Tako se bo sicer lahko zgodilo, da kak nov klicatelj preprosto ne bo mogel takoj dobiti zveze, vendar po drugi strani lahko vsakdo, ko zvezo enkrat dobi, v miru dela vsaj x minut. Pri izboru števila x bi morali upoštevati gostoto klicev (koliko klicev na uro), številom modemskih linij, povprečni čas, ki ga uporabniki prebijejo na zvezi, pa tudi želeni delež uspešnih klicev (v vsaj koliko odstotkih primerov naj uporabnik, ki pokliče naše modeme, tudi res dobi zvezo z enim od njih).

Pravzaprav s stališča ponudnika dostopa do Interneta metanje uporabnikov z zveze niti ni tako zelo koristno, saj jih lahko s tem prekine sredi kakšnega pomembnega dela (in si s tem nakoplje njihovo nezadovoljstvo), od tega, da se bo hip zatem povezal s tako sproščenim modemom nek drug uporabnik (namesto da bi tisti prejšnji nadaljeval z delom), pa nima ponudnik nobene posebne koristi. To, da uporabniki preprosto ne morejo do Interneta, če so vsi modemi zasedeni, je za ponudnika škodljivo šele, če postanejo zaradi tega tako nezadovoljni, da začnejo uporabljati storitve kakšnega drugega ponudnika. Če uporabnikov nočemo izgubljati, moramo torej sproti dokupovati modeme in skrbeti, da razmerje med številom uporabnikov in številom modemov ne postane previsoko.

REŠITVE NALOG ZA DRUGO SKUPINO

N: 294

R1997.2.1

(a) Program deli prvo število z drugim in izpiše celi del ter ostanek. V glavni zanki odšteva `n2` od `n1` in pri tem počasi povečuje celi cel `c`.[56] Poseben primer je, če je `n1` manjši od `n2` — v tem primeru odšteje enkrat, čeprav ne bi smel nobenkrat, kar rešimo s tem, da `d` postavimo na 1 (sicer pa na 0). Glavna zanka dela pravzaprav z absolutnima

[56] Namen prireditve `n1 := n1 + not n2 + 1` je zmanjšati `n1` za `n2` (enako kot pri prireditvi `n1 := n1 − n2`). Pri tem se program zanaša na to, da so cela števila v našem računalniku predstavljena z dvojiškim komplementom (kar v praksi ponavadi tudi drži); če so spremenljivke tipa `integer` dolge recimo k bitov, to pomeni, da bo negativno število $-a$ (za nek $a > 0$) predstavljeno z zaporedjem k bitov, ki ima dejansko vrednost $2^k - a$. Če pa v pozitivnem številu a negiramo vseh k bitov, dobimo zaporedje bitov z dejansko vrednostjo $(2^k - 1) - a$; če temu še prištejemo 1, dobimo torej ravno zaporedje bitov z vrednostjo $2^k - a$, ki (če delamo s predznačenimi števili) predstavlja negativno število $-a$. Tako torej s prištevanjem vrednosti `not n2 + 1` pravzaprav odštevamo `n2`.

vrednostma obeh števil, pred tem pa postavimo `e` na -1, če sta bili različno predznačeni, sicer pa na 1, `f` pa dobi predznak deljenca `n1`. S tem zagotovimo, da za prvotni vrednosti `n1` in `n2` ter za števili, ki ju program na koncu izpiše (recimo jima `k` in `o`) vedno velja zveza `n1 = k * n2 + o`, pa tudi $|o| < |n2|$. Količnik se po deljenju zaokroži proti 0, torej navzdol, če je bil pozitiven, in navzgor, če je bil negativen.

(b) Ukinemo lahko spremenljivke `c`, `d`, `e` in `f`, pobrišemo vse pripadajoče stavke, ukinemo zanko in spremenimo izpis v:

```
        printf("%d %d", n1 / n2, n1 % n2);
oz.     WriteLn(n1 div n2, ' ', n1 mod n2);
```

Treba pa je priznati, da v primeru, ko sta `n1` in/ali `n2` negativna, ni tako zelo zanesljivo, če bo tako poenostavljen program res dajal enake rezultate kot prvotni, kajti pri negativnih operandih definirajo različni procesorji, jeziki in prevajalniki celi del količnika in ostanek na različne načine. Razlikujejo se predvsem po tem, ali natančni rezultat deljenja zaokrožijo vedno proti 0 ali pa vedno navzdol (ta razlika se seveda pozna le, če sta operanda različno predznačena in je rezultat deljenja zato negativen); ostanek je potem običajno določen tako, da velja `(n1 / n2) * n2 + (n1 % n2) = n1`. Pri zaokrožanju proti 0 je ostanek enako predznačen kot deljenec, pri zaokrožanju navzdol pa je ostanek predznačen enako kot delitelj. Še tretja možnost je, da bi zahtevali, naj bo ostanek vedno nenegativen. V vsakem primeru pa je ostanek po absolutni vrednosti manjši od delitelja.

		Zaokrožanje količnika proti 0		Zaokrožanje količnika navzdol		Ostanek nenegativen	
n_1	n_2	n_1/n_2	$n_1 \% n_2$	n_1/n_2	$n_1 \% n_2$	n_1/n_2	$n_1 \% n_2$
11	5	2	1	2	1	2	1
11	-5	-2	1	-3	-4	-2	1
-11	5	-2	-1	-3	4	-3	4
-11	-5	2	-1	2	-1	3	4

Pri jezikih C in C++ je bila na primer odločitev za vrsto zaokrožanja pri deljenju dolgo časa prepuščena piscem prevajalnikov (verjetno z namenom, da bi lahko vrnili kar rezultat, ki ga izračuna procesorjev ukaz za deljenje, pa kakršnokoli obliko zaokrožanja ta že pač uporablja), v standardu C99 pa so uvedli zaokrožanje proti 0. Intelovi procesorji zaokrožajo proti 0, zato je ta oblika zaokrožanja vsaj na njih prevladovala že prej. Java in C# zaokrožata proti 0, Python, Oberon in Haskell pa vedno zaokrožijo navzdol. Pri standardnem pascalu zaokroža **div** proti 0, **mod** pa je definiran le pri pozitivnem delitelju (pri negativnem velja računanje **mod** za napako) in to tako, da je vedno nenegativen; to žal pomeni, da velja `n2 * (n1 div n2) + (n1 mod n2) = n1` samo za nenegativne deljence `n1` (ali pa če se deljenje izide), sicer pa ne.

Vendar pa se mnogi pascalski prevajalniki te definicije ne držijo in računajo `n1 mod n2` kot `n1 - (n1 div n2) * n2`. Slabost definicije, ki uporablja zaokrožanje proti 0, je na primer naslednja: če povečamo `n1` za `n2`, bi nemara pričakovali, da se bo količnik povečal za 1, ostanek pa se ne bo spremenil; vendar pri tej definiciji to ne drži vedno (težava nastopi, če se spremeni predznak deljenca `n1`).[57]

(c) Preveriti moramo, če je drugo število enako 0, saj ne smemo deliti z 0 (obstoječi program bi se v tem primeru zaciklal, saj vrednosti `n1` v glavni zanki sploh ne bi spreminjal). Še ena slabost obstoječega programa je, da uporablja `n1 + not n2 + 1`, da bi izračunal `n1 - n2`. Vrednost `not n2 + 1` je enaka $-$`n2` v primeru, če so števila v računalniku predstavljena z dvojiškim komplementom, drugače pa to ni nujno res. Na kakšnih starih in/ali eksotičnih računalnikih bi utegnili naleteti tudi na drugačne predstavitve celih števil (na primer eniški komplement, kjer bi že `not n2` sam po sebi imel vrednost $-$`n2`).

(d) Program dvakrat izpiše 42 pri poljubnem paru celih števil (a, b), za kateri velja, da je $a = 42(b + 1)$ in hkrati $b > 42$. Primer: 1848 in 43.

N: 295

R1997.2.2

Za skrivanje sporočila ni treba drugega, kot da beremo znake enega za drugim, nato pa pri vsakem znaku jemljemo iz njega posamezne bite in vsakega vpišemo v najnižji bit enega piksla slike. Pri razkrivanju sporočila ravnamo ravno obratno; iz zaporednih pikslov jemljemo spodnji bit in te bite združujemo po osem skupaj v znake, ki jih potem izpisujemo. V obeh primerih si pomagamo s preprostimi operacijami nad biti: `n mod 2` vrne najnižji bit števila `n`, `n div 2` zamakne `n` za eno mesto v desno (najnižji bit se pri tem izgubi), `n * 2` pa zamakne `n` za eno mesto v levo (v najnižji bit pride vrednost 0).

```
const w = 256; h = 256;
var Slika: array [1..h, 1..w] of 0..255;

procedure SkrijBesedilo;
var
  iw, ih, j, ic, bit: integer;
  c: char;
begin
  iw := 1; ih := 1;
  while ih <= h do begin
    Read(c); ic := Ord(c);
    { Shrani 8 bitov znaka v zaporedne piksle, začni z najnižjim bitom (lsb). }
    { Predpostavimo, da je širina slike mnogokratnik 8. }
```

[57] Nekaj literature: obširna razprava o zaokrožanju pri deljenju je bila v skupini *comp.lang.c++.moderated* v začetku februarja 2000; Daan Leijen: *Division and modulus for computer scientists*, `http://www.cs.uu.nl/~daan/`; Raymond T. Boute: *The Euclidean definitions of the functions div and mod*, ACM Transactions on Programming Languages and Systems, 14(2):127–144, April 1992.

```
      for j := 1 to 8 do begin
        bit := ic mod 2; ic := ic div 2; { Izlušči naslednji bit znaka. }
        Slika[ih, iw] := (Slika[ih, iw] div 2) * 2 + bit;
        iw := iw + 1;
      end; {for}
      if iw > w then begin iw := 1; ih := ih + 1 end;
    end; {while}
end; {SkrijBesedilo}

procedure RazkrijBesedilo;
var
  ic, i, j: integer;
  k, kp: integer; { k šteje bite; kp = 2^k }
begin
  k := 0; kp := 1; ic := 0;
  for i := 1 to h do
    for j := 1 to w do begin
      { Izlušči zaporedne bite iz pikslov in jih združi po 8 v en znak. }
      { Pazimo na obratni vrstni red bitov: najnižji bit je v prvem pikslu. }
      ic := ic + kp * (Slika[i, j] mod 2); k := k + 1; kp := 2 * kp;
      if k >= 8 then begin { Znak je kompleten. }
        Write(Chr(ic));
        k := 0; kp := 1; ic := 0; { Pripravi se na nov znak. }
      end; {if}
    end; {for}
  WriteLn;
end; {RazkrijBesedilo}
```

R1997.2.3

N: 295

Podnize lahko štejemo rekurzivno. Recimo, da bi radi prešteli pojavitve podniza $p[1..m]$ v nizu $s[1..n]$. Vsaka pojavitev podniza p se mora končati pri eni od pojavitev njegovega zadnjega znaka, $p[m]$, v nizu s; če je to na primer i-ti znak s-ja, je preostanek neka pojavitev niza $p[1..m-1]$ v nizu $s[1..n-1]$. Pojavitve podniza p v s, ki se končajo pri $s[i]$, lahko torej preštejemo z rekurzivnim klicem, ki bo obdeloval za en znak krajši podniz. Na koncu moramo sešteti vse tako najdene pojavitve, torej po vseh primernih i (takih, za katere je $s[i] = p[m]$). Rekurzija se konča, ko je podniz dolg le še en znak — takrat moramo le prešteti vse pojavitve tega znaka v nizu s.

Če označimo število pojavitev podniza $p[1..i]$ v nizu $s[1..j]$ z $f(i,j)$, smo tako dobili rekurzivno zvezo:

$$f(i,j) = \sum_{1 \le k \le j,\ s[k]=p[i]} f(i-1,k-1) \qquad (\star)$$

s posebnimi primeri

$$\begin{array}{rcl} f(1,j) & = & \text{število pojavitev znaka } p[1] \text{ v nizu } s[1..j], \\ f(i,j) & = & 0, \text{če } i > j \text{ (niz prekratek za podniz).} \end{array}$$

Primera za $f(1,j)$ pravzaprav ni treba obravnavati posebej, saj lahko definiramo $f(0,j) = 1$ in potem $f(1,j)$ računamo po splošni formuli ($\star$).

Toda po formuli ($\star$) bi morali $f(i,j)$ računati z zanko po k. Temu se lahko izognemo in prihranimo nekaj časa, če rekurzivni razmislek zastavimo malo drugače. Vsaka pojavitev niza $p[1..i]$ v nizu $s[1..j]$ se ali konča pri znaku $s[j]$ (kar je seveda mogoče le, če je $p[i] = s[j]$; v tem primeru bo preostanek te pojavitve kar neka pojavitev niza $p[1..i-1]$ v nizu $s[1..j-1]$) ali pa ne (in v tem primeru je taka pojavitev hkrati tudi pojavitev celega niza $p[1..i]$ v nizu $s[1..j-1]$. Tako dobimo namesto ($\star$) preprostejšo formulo

$$f(i,j) = \begin{cases} f(i,j-1) + f(i-1,j-1) & : \text{ če } p[i] = s[j] \\ f(i,j-1) & : \text{ sicer.} \end{cases}$$

Primer programa, ki računa to funkcijo:

```
function PrestejPodnize(p, s: string): longint;

  { Izračuna, kolikokrat se pojavi p[1..i] v nizu s[1..j]. }
  function f(i, j: integer): integer;
  begin
    if i > j then f := 0
    else if i = 0 then f := 1
    else if s[j] = p[i] then f := f(i, j - 1) + f(i - 1, j - 1)
    else f := f(i, j - 1);
  end; {f}

var m, n: integer;
begin
  m := Length(p); n := Length(s);
  PrestejPodnize := f(m, n);
end; {PrestejPodnize}
```

Slabost tega postopka je, da je lahko zelo neučinkovit. Skupno število pojavitev podniza p v nizu s, ki ga na koncu vrne podprogram PrestejPodnize, je nastalo v okviru rekurzivnih klicev podprograma f s seštevanjem enic in ničel, ki jih prispevata stavka f := 1 in f := 0. Torej, če ima p v s-ju r pojavitev, se mora stavek f := 1 izvesti r-krat, kar je neprijetno, če je r velik. In že pri razmeroma kratkih nizih je število pojavitev podniza lahko veliko: niz $a_1a_2\cdots a_m$ (če so $a_1, \ldots, a_m$ same različne črke) ima v nizu $a_1a_1a_2a_2\cdots a_ma_m$ kar 2^m pojavitev.

Postopek lahko izboljšamo, če opazimo, da je vrednost, ki jo vrne funkcija f, odvisna le od vrednosti parametrov i in j — edine preostale zunanje vrednosti,

ki jih f uporablja, so oba niza in njuni dolžini, te pa se nikoli ne spreminjajo. Torej, če f dvakrat pokličemo z istima vrednostma parametrov, mora obakrat vrniti isti rezultat. Zato si je pametno ta rezultat ob prvem klicu zapomniti v kakšni tabeli in ob kasnejših klicih takoj vrniti to vrednost, namesto ta jo gremo računat ponovno od začetka (temu včasih pravijo *memoizacija*; lepše bi temu verjetno lahko rekli „pomnjenje“). Spodnji program si rezultat klica f(i, j) zapomni v tabeli t[i, j] (na začetku postavimo vse t[i, j] na -1, da bomo vedeli, da teh vrednosti še nismo izračunali).

```
function PrestejPodnize(p, s: string): longint;
var m, n: integer; t: array [1..MaxDolzPodniza, 1..MaxDolzNiza] of integer;

  function f(i, j: integer): integer;
  begin
    if i > j then f := 0
    else if i = 0 then f := 1
    else begin
      if t[i, j] < 0 then
        if s[j] = p[i] then t[i, j] := f(i, j - 1) + f(i - 1, j - 1)
        else t[i, j] := f(i, j - 1);
      f := t[i, j];
    end; {if}
  end; {f}

var i, j: integer;
begin
  m := Length(p); n := Length(s);
  for i := 1 to m do for j := 1 to n do t[i, j] := -1;
  PrestejPodnize := f(m, n);
end; {PrestejPodnize}
```

(Tabelo t bi lahko alocirali tudi dinamično, da bi imela natančno $m \times n$ celic.)

Zdaj se f(i, j) kliče največ dvakrat za vsak par vrednosti i in j — namreč ob prvem klicu f(i + 1, j + 1) in ob prvem klicu f(i, j + 1), ob kasnejših klicih teh dveh funkcij pa bosta onidve vrnili svoj rezultat iz tabele t in ne bosta rekurzivno klicali f(i, j). Količina časa, ki ga naš program tako porabi, je torej $O(m \cdot n)$, saj imamo z vsako celico tabele t konstantno veliko dela.

Program pa lahko naredimo še malce bolj elegantnega, če upoštevamo, katere rezultate rekurzivnih klicev utegnemo potrebovati, ko bomo računali $f(i, j)$: to sta le $f(i, j-1)$ in $f(i-1, j-1)$. Torej lahko takrat, ko računamo $f(i, j)$, že pozabimo vrednosti $f(i', j')$ za $j' < j - 1$. Če torej računamo vrednosti f sistematično po naraščajočih j in pri vsakem j-ju po vseh i, je dovolj, če v tabeli t hranimo vrednosti $f(i, j-1)$. Vrednosti $f(i, j)$ za trenutni j pa je koristno računati po padajočih i, ker nam to zagotavlja, da po tistem, ko izračunamo $f(i, j)$, vrednosti $f(i, j-1)$ ne bomo več potrebovali, zato jo lahko

v tabeli takoj povozimo z vrednostjo $f(i, j)$. Rekurzivnim klicem se lahko tako povsem odpovemo in vse naredimo z dvema gnezdenima zankama:

```
function PrestejPodnize(p, s: string): longint;
var m, n, i, j, iMax: integer;
    t: array [0..MaxDolzPodniza] of integer;
begin
  m := Length(p); n := Length(s);
  { Izračunajmo najprej f(i, 1) za vse i. Tu gledamo le
    prvi znak niza s in f(i, 1) je 0 pri za vse i > 1. }
  t[0] := 1; { Ker smo videli, da je koristno vzeti f(0, j) = 1. }
  for i := 1 to m do t[i] := 0;
  if p[1] = s[1] then t[1] := 1;
  { Izračunajmo zdaj f(i, j) za vse večje j. }
  for j := 2 to n do begin
    if j > m then iMax := m else iMax := j;
    for i := iMax downto 1 do
      { V celicah t[i + 1..m] so vrednosti f(i + 1, j), ..., f(m, j),
        v celicah t[1..i - 1] pa vrednosti f(1, j - 1), ..., f(i, j - 1).
        Izračunajmo f(i, j) in jo vpišimo v t[i]. }
      if p[i] = s[j]
      then t[i] := t[i] + t[i - 1]    { f(i, j) = f(i, j - 1) + f(i - 1, j - 1) }
      else  t[i] := t[i];             { f(i, j) = f(i, j - 1) }
  end; {for j}
  PrestejPodnize := t[m];
end; {PrestejPodnize}
```

Tehniki, ki smo jo uporabili pri tej (in pravzaprav tudi pri prejšnji) različici, pravimo *dinamično programiranje*. Količina porabljenega časa je tudi tokrat $O(m \cdot n)$, čeprav je v praksi ta različica mogoče malo hitrejša od prejšnje, ker tista porabi precej časa za knjigovodske opravke pri rekurzivnih klicih. Prihranili smo tudi precej pomnilnika, saj za tabelo `t` potrebujemo le še $O(m)$ namesto $O(mn)$ pomnilnika. Je pa zato pri tej zadnji različici precej več možnosti, da se zmotimo pri kakšnih indeksih.

Prireditev t[i] := t[i] je v zgornjem podprogramu seveda povsem odveč in bi jo bilo v praksi najpametneje izpustiti. Notranja zanka (po i) torej spreminja tabelo `t` le pri tistih i, za katere je $p[i] = s[j]$, zato je dovolj, če obiščemo le te i, ostale pa preskočimo. Lahko bi si torej za vsak j pripravili seznam vseh primernih i in se potem le sprehodili po tem seznamu; časovna zahtevnost bi bila zdaj sorazmerna s skupno dolžino teh seznamov, torej s številom parov (i, j), za katere je $p[i] = s[j]$. V najslabšem primeru jih je sicer $m \cdot n$ in takrat ničesar ne pridobimo, pri bolj realističnih nizih (ki nimajo samih enakih črk) pa jih utegne biti precej manj. Do takšnih seznamov lahko pridemo tako, da

črke p-ja in s-ja uredimo in „zlijemo“.[58]

```
function PrestejPodnize(p, s: string): longint;
var m, n, i, j, iu, ju: integer;
    t: array [0..MaxDolzPodniza] of integer;
    su, iuZadnji: array [1..MaxDolzNiza] of integer;
    pu: array [1..MaxDolzPodniza] of integer;
begin
  m := Length(p); n := Length(s);
  for ju := 1 to n do su[ju] := ju;
  Preuredi elemente tabele su tako, da bo za vse ju veljalo:
  (s[su[ju]] < s[su[ju + 1]]) or ((s[su[ju]] = s[su[ju + 1]]) and (su[ju] < su[ju + 1]));
  for iu := 1 to m do pu[iu] := iu;
  Preuredi elemente tabele pu tako, da bo za vse iu veljalo:
  (p[pu[iu]] < p[pu[iu + 1]]) or ((p[pu[iu]] = p[pu[iu + 1]]) and (pu[iu] < pu[iu + 1]));
  iu := 0;
  for ju := 1 to n do begin
    j := su[ju];
    while iu < m do begin
      i := pu[iu + 1];
      if (p[i] > s[j]) or ((p[i] = s[j]) and (i > j)) then break;
      iu := iu + 1;
    end; {while};
    iuZadnji[j] := iu;
  end; {for ju}
```

Za urejanje bi načeloma lahko uporabili katerega koli od mnogih znanih algoritmov za urejanje. Po tej inicializaciji imamo v tabeli pu števila i od 1 do m, urejena po naraščajoči vrednosti $p[i]$ in pri enakih $p[i]$ še po naraščajočih i. Vrednost iuZadnji[j] = iu pa nam pove, da je pu[iu] zadnji element tabele pu, ki se še nanaša na črko s[j] (torej da je p[pu[iu]] = s[j]) in je istočasno tudi manjši ali enak j (torej da je pu[iu] ≤ j). Če takega elementa sploh ni, je iuZadnji[j] enak 0 ali pa kaže na nek iu, pri katerem je p[pu[iu]] < s[j]. Nadaljujemo lahko tako kot pri prejšnji rešitvi, le glavno zanko bomo morali malo predelati:

```
  t[0] := 1; { Tu se ni nič spremenilo. }
  for i := 1 to m do t[i] := 0;
  if p[1] = s[1] then t[1] := 1;
  for j := 2 to n do begin { Tale glavna zanka se malo spremeni. }
    iu := iuZadnji[j];
    while iu > 0 do begin
      i := pu[iu]; if p[i] <> s[j] then break;
```

[58] J. W. Hunt, T. G. Szymanski: *A fast algorithm for computing longest common subsequences*. CACM, 20(5):350–353, May 1977.

```
        t[i] := t[i] + t[i − 1];
        iu := iu − 1;
      end; {while}
    end; {for j}
    PrestejPodnize := t[m];
end; {PrestejPodnize}
```

Če sta niza p in s dovolj dolga in pride do ujemanja $p[i] = s[j]$ pri dovolj malo parih (i, j), bo prihranek zaradi manjšega števila izvajanj notranje zanke (**while** iu > 0 v zadnji različici rešitve) večji od časa, ki smo ga porabili na začetku za pripravo tabele iuZadnji.

N: 296

R1997.2.4

Če bi na začetku vsak delež d_i zaokrožili navzdol, torej na $\lfloor d_i \rfloor$, bi gotovo dobili vsoto, manjšo ali enako 100. Recimo, da je vsota $100 - k$; potem moramo popraviti k deležev — za zdaj so vsi zaokroženi navzdol, mi pa jih moramo zaokrožiti navzgor. Pametno se je najprej lotiti tistih, pri katerih je zaokroževanje navzdol naredilo največjo napako, torej tisth z največjo $d_i - \lfloor d_i \rfloor$ (kajti pri teh bo napaka $\lceil d_i \rceil - d_i$ po zaokrožanju navzgor najmanjša). Če bi bilo deležev veliko, bi jih bilo koristno urediti po padajoči napaki, spodnji program pa vsakič poišče naslednjega kar s pregledom celotne tabele vseh deležev.

```
const n = 20;
var
  Delez: array [1..n] of real;
  Odst: array [1..n] of integer;
  Min: real;
  j, IndMin, Vsota: integer;
begin
  PreberiAliIzracunajDeleze;    { S tem se ne bomo ukvarjali, saj naloga tako ali
                                  tako pravi, da so deleži že podani v tabeli. }
  Vsota := 0;
  for j := 1 to n do begin
    Delez[j] := Delez[j] * 100;
    Odst[j] := Trunc(Delez[j]); Vsota := Vsota + Odst[j];
  end; {for}
  while Vsota < 100 do begin
    IndMin := 1; Min := Odst[IndMin] − Delez[IndMin];
    for j := 2 to n do
      if Odst[j] − Delez[j] < Min then
        begin IndMin := j; Min := Odst[j] − Delez[j] end;
    Odst[IndMin] := Odst[IndMin] + 1; Vsota := Vsota + 1;
  end; {while}
  for j := 1 to n do WriteLn(Odst[j]);
end.
```

Zanki **while** bi lahko prihranili nekaj ponovitev, če bi na začetku zaokrožili vsak delež k najbližjemu celemu številu, ne pa vseh navzdol. Potem bi morali ločiti dve možnosti: lahko je vsota zaokroženih deležev premajhna in moramo kot doslej povečati nekaj deležev, ki so bili zaokroženi navzdol (in sicer tiste z največjo $d_i - \lfloor d_i \rfloor$; lahko pa je vsota prevelika in moramo zmanjšati nekaj deležev, ki so bili zaokroženi navzgor (tiste z največjo $\lceil d_i \rceil - d_i$). Seveda bi lahko uporabili tudi kak algoritem za urejanje in števila eksplicitno uredili po tem, za koliko so se ob zaokrožanju spremenila.

REŠITVE NALOG ZA TRETJO SKUPINO

R1997.3.1

Kazalca p1 in p2 potujeta vzdolž seznama s. Kazalec p1 se pri vsaki izvedbi telesa zanke **repeat** premakne naprej po seznamu s za en element, kazalec p2 pa za dva elementa. Kazalec p2 potuje torej dvakrat hitreje vzdolž seznama s kot kazalec p1. N: 297

V mejnem primeru, ko velja $s = \mathbf{nil}$ in $n = 0$, se telo zanke **repeat** izvede enkrat, funkcija DvojnaHitrost pa vrne false. Predpostavimo sedaj, da seznam ni prazen, torej $n > 0$. Če kazalec p2 med potovanjem vzdolž seznama dobi vrednost **nil**, je dosegel zadnji, torej n-ti element seznama in zanka **repeat** se izteče, funkcija pa vrne vrednost false. Pri n elementih v seznamu kazalec p2 potrebuje n premikov, da dobi vrednost **nil**, za to pa potrebuje $\lceil n/2 \rceil$ iteracij **repeat**. Obstaja pa še možnost, da kazalec p2 nikoli ne dobi vrednosti **nil**. To se zgodi v primeru, ko kazalec Naslednji n-tega elementa kaže na k-ti element seznama, za nek $k \leq n$. Seznam torej lahko vsebuje zanko, v kateri so elementi z indeksi $k, k+1, \ldots, n$. V k-tem koraku kazalec p1 doseže zanko elementov v seznamu, kazalec p2 pa je tudi že v zanki. A ker kazalec p1 potuje s korakom 1, kazalec p2 pa s korakom 2, se z vsako ponovitvijo zanke **repeat** razdalja med kazalcema v zanki zmanjša za 1. To pomeni, da bo kazalec p2 prej ali slej ujel kazalec p1, vrednosti kazalcev bosta enaki in zanka **repeat** se bo iztekla, funkcija DvojnaHitrost pa bo vrnila vrednost true.

Oštevilčimo v mislih elemente zanke s številkami od 0 (k-ti element seznama) do $n - k$ (n-ti element). Kazalec p1 je najprej naredil $k - 1$ korakov, da je dosegel zanko; kazalec p2 je v tem času naredil dvakrat toliko korakov, torej je po tistem, ko je prišel do zanke (do k-tega elementa seznama), naredil še $k - 1$ korakov. S tem je prišel na element s številko $(k - 1) \bmod (n - k + 1)$. Ker je p1 trenutno na elementu 0 in se mu p2 v vsaki iteraciji zanke približa za en element, bo porabil p2 še $(0 - (k - 1)) \bmod (n - k + 1)$ iteracij, da ga bo ujel. Tu je mišljeno, naj mod vedno vrača vrača vrednosti od 0 do $n - k$ (torej $a \bmod m := a - m\lfloor a/m \rfloor$; na primer: $(-22) \bmod 6 = 2$). Negativnega operanda $-(k - 1)$ se lahko znebimo tudi tako, da mu prištejemo nek večkratnik delitelja (saj na mod to ne bo vplivalo); na primer, ker je $k \leq n$,

je $n-k+1 \geq 1$, zato je $(k-1)(n-k+1) \geq (k-1)$; zato je

$$\begin{aligned}[-(k-1)] \bmod (n-k+1) &= [(k-1)(n-k+1)-(k-1)] \bmod (n-k+1) \\ &= (k-1)(n-k) \bmod (n-k+1),\end{aligned}$$

pri čemer je zdaj deljenec, $(k-1)(n-k)$, gotovo nenegativen. Dobljena formula odpove le v primerih, ko je $k=1$, saj sta tam `p1` in `p2` že na začetku skupaj in formula napove 0 iteracij; v resnici pa se takrat izvede n iteracij. Tako smo dobili:

$$\textit{št. iteracij} = \begin{cases} 1, & \text{če } n = 0 \\ \lceil n/2 \rceil, & \text{če } n > 0 \text{ in ni zanke} \\ n, & \text{če } n > 0 \text{ in } k = 1 \\ k-1+[(k-1)(n-k) \bmod (n-k+1)] & \text{sicer.} \end{cases}$$

Skratka, funkcija DvojnaHitrost vrne vrednost `true`, če v seznamu `s` obstaja zanka, sicer pa vrednost `false`. V najslabšem primeru (pri praznem seznamu) se telo zanke **repeat** izvede največ $(n+1)$-krat (če je n število elementov v seznamu).

N: 297

R1997.3.2

Na to, kako se prenaša pomembnost med meščani, vpliva le njihova moč, ta pa se ob prenašanju pomembnosti nič ne spreminja. Torej je vseeno, v kakšnem vrstnem redu meščani prenašajo svojo pomembnost na svoje močnejše prijatelje; v vsakem primeru bomo na koncu dobili enak razpored pomembnosti. Poleg tega se pomembnost prenaša na vedno močnejše ljudi, ker pa je moč navzgor omejena (če imamo n meščanov, ima lahko posameznik moč največ $n-1$, saj več kot toliko prijateljev pač ne more imeti), se mora tak postopek prenašanja pomembnosti zagotovo prej ali slej končati. Torej nam ni treba skrbeti, da bi se nam kak postopek zaciklal ali pa da rešitev ne bi bila enolična.

Meščani z močjo 0 so po definiciji brez prijateljev in se torej njihova pomembnost ne bo nikoli spreminjala; ničesar ne bodo razdajali in ničesar dobivali. Z njimi se nam torej sploh ni treba ukvarjati. — Meščani z močjo 1 ne bodo nikoli dobili nič pomembnosti od drugih: dobili bi jo lahko le od takih z močjo 0, toda če ima nekdo moč 0, pomeni, da nima prijateljev in torej svoje pomembnosti ne bo delil. Torej, ko meščani z močjo 1 razdelijo svojo začetno pomembnost (če jo imajo komu dati), se njihova pomembnost v bodoče prav gotovo ne bo več spreminjala. Zato je pametno najprej opraviti z njimi, potem pa vemo, da lahko odslej pozabimo nanje. — Ko smo opravili to, vidimo, da meščani z močjo 2 tudi ne bodo dobili nič več pomembnosti: lahko bi jo dobili le od takih z močjo 1, toda slednji so doslej že razdelili vse, kar so imeli, in tudi v bodoče ne bodo delili ničesar več. Če torej zdaj meščani z močjo 2 razdelijo svojo pomembnost med svoje močnejše prijatelje, bomo tudi pri njih že dobili končno stanje pomembnosti.

Tako lahko nadaljujemo proti vse močnejšim meščanom. Preden se začnemo ukvarjati s tistimi z močjo k, smo za vse šibkejše meščane že izračunali njihovo končno pomembnost, tako da meščani z močjo k v prihodnje gotovo ne bodo prejeli nič dodatne pomembnosti več; meščani z močjo k lahko potem razdelijo svojo pomembnost med svoje močnejše prijatelje in s tem dobimo njihove dokončne pomembnosti, ki se v bodoče ne bodo več spreminjale.

```
program Ravbarji;
const MaxN = ...;
var n, i, u, v, StMocnejsih: integer;
    StZMocjo, NaslZMocjo: array [0..MaxN − 1] of integer;
    Moc, PoMoci, Mocnejsi: array [1..MaxN] of integer;
    Pomembnost: array [1..MaxN] of real; Delez: real;
begin
  n := PreberiMescane;
  { Določimo moč vsakega meščana. }
  for u := 1 to n do begin
    Moc[u] := 0; Tekoci(u);
    while Naslednji <> 0 do Moc[u] := Moc[u] + 1;
  end; {for}
  { Koliko jih ima določeno moč? }
  for i := 0 to n − 1 do StZMocjo[i] := 0;
  for u := 1 to n do StZMocjo[Moc[u]] := StZMocjo[Moc[u]] + 1;
  { V tabeli PoMoci bomo pripravili številke meščanov po naraščajoči moči.
    NaslZMocjo[i] pove, kam je treba vpisati naslednjega meščana z močjo i. }
  NaslZMocjo[0] := 1;
  for i := 1 to n − 1 do NaslZMocjo[i] := NaslZMocjo[i − 1] + StZMocjo[i − 1];
  for u := 1 to n do begin
    PoMoci[NaslZMocjo[Moc[u]]] := u;
    NaslZMocjo[Moc[u]] := NaslZMocjo[Moc[u]] + 1;
  end; {for}
  { Prenašajmo pomembnosti od šibkejših k močnejšim. }
  for u := 1 to n do Pomembnost[u] := 1.0;
  for i := 1 to n do begin
    u := PoMoci[i];
    { Poglejmo, kateri u-jevi prijatelji so močnejši od njega. }
    Tekoci(u); StMocnejsih := 0; v := Naslednji;
    while v <> 0 do begin
      if Moc[v] > Moc[u] then
        begin StMocnejsih := StMocnejsih + 1; Mocnejsi[StMocnejsih] := v end;
      v := Naslednji;
    end; {while}
    if StMocnejsih > 0 then begin
      { Razdelimo u-jevo pomembnost med močnejše prijatelje. }
      Delez := Pomembnost[u] / StMocnejsih;
      Pomembnost[u] := 0;
```

```
        while StMocnejsih > 0 do begin
          v := Mocnejsi[StMocnejsih]; StMocnejsih := StMocnejsih − 1;
          Pomembnost[v] := Pomembnost[v] + Delez;
        end; {while}
      end; {if}
    end; {for}
    { Izpišimo meščane z neničelno pomembnostjo. }
    for u := 1 to n do if Pomembnost[u] > 0 then
      WriteLn(u, ' ', Pomembnost[u]:0:5);
end. {Ravbarji}
```

Če imamo n meščanov in m prijateljstev med meščani, je časovna zahtevnost tega postopka $O(n + m)$, prostorska pa $O(n)$ (če odmislimo prostor, ki ga verjetno uporabljajo PreberiMescane, Tekoci in Naslednji za hranjenje podatkov o meščanih in njihovih prijateljstvih). Če ima vsak meščan bolj malo prijateljev, je $m = O(n)$, lahko pa je vsak prijatelj skoraj vseh drugih in je tedaj $m = O(n^2)$.

Če ima vsak meščan enako število prijateljev, do prenašanja pomembnosti sploh ne pride — vsi obdržijo pomembnost 1.

N: 298

R1997.3.3

Iz čipa beremo zapise enega za drugim. V tabelo Indeksi si zapišemo položaj začetka zapisa v čipu, v tabelo Cene pa vpišemo število skokov, ki jih moramo narediti, da pridemo od prvega zapisa do trenutnega. To je v najslabšem primeru za ena večje od števila skokov, ki nas privedejo do zapisa, ki stoji pred trenutnim. Mogoče pa gre tudi z manj skoki; zato preverimo, če lahko s spremembo velikosti kakšnega izmed prejšnjih zapisov pridemo do trenutnega in to tako, da se bo število skokov čimbolj zmanjšalo. Če smo tak zapis našli, si njegovo zaporedno številko zapišemo v tabelo OdKod. V nasprotnem primeru pa v tabelo OdKod zapišemo zaporedno številko prejšnjega zapisa.

Ko pridemo do zadnjega zapisa, se na zadnjem mestu v tabeli Cene nahaja število skokov, ki jih potrebujemo, da pridemo do zadnjega zapisa.

Na koncu moramo pot z najmanjšim številom skokov zapišati še v Flash ROM. Zdaj uporabimo tabelo Odkod, ki brana od zadaj na način Zapis := OdKod[Zapis] vsebuje vse zapise, ki jim moramo spremeniti velikost.

```
var
  Indeksi, Cene, OdKod: array [1..MaxFlashROM] of integer;
  Zadnji, Lokacija, Velikost, Zapis: integer;
begin
  Indeksi[1] := 1;
  Cene[1] := 0;
  OdKod[1] := 0;
  Lokacija := FlashROM[1] + 1;
  Zadnji := 1;
```

```
  while FlashROM[Lokacija] <> 0 do begin
    Zadnji := Zadnji + 1;
    Indeksi[Zadnji] := Lokacija;
    Cene[Zadnji] := Cene[Zadnji − 1] + 1;
    OdKod[Zadnji] := Zadnji − 1;
    for Zapis := 1 to Zadnji − 2 do begin
      if Cene[Zapis] + 1 < Cene[Zadnji] then begin
        { Splačalo bi se skočiti od zapisa Zapis do zapisa Zadnji;
          preverimo, če je tak skok sploh mogoč. }
        Velikost := (Lokacija − Indeksi[Zapis]) or FlashROM[Indeksi[Zapis]];
        if Indeksi[Zapis] + Velikost = Lokacija then begin
          Cene[Zadnji] := Cene[Zapis] + 1;
          OdKod[Zadnji] := Zapis;
        end; {if}
      end; {if}
    end; {for}
    Lokacija := Lokacija + FlashROM[Lokacija];
  end; {while}
  while Zadnji <> 1 do begin
    Zapis := OdKod[Zadnji];
    Vpisi(Indeksi[Zapis], Indeksi[Zadnji] − Indeksi[Zapis]);
    Zadnji := Zapis;
  end; {while}
end.
```

R1997.3.4

N: 299

Ko dobimo od uporabnika nov zahtevek, moramo najprej preveriti, če so zahtevani podatki že v vmesnem pomnilniku. Če so, nam jih funkcija Poslji tudi vrne in jih moramo samo še poslati naprej uporabniku. Drugače pa nam Poslji pove številko toka, kamor naj pošiljamo te podatke (ko jih bomo dobili iz omrežja), da se bodo shranili v vmesni pomnilnik. Obenem bo pametno te podatke pošiljati tudi uporabniku, ki jih je sploh najprej zahteval. Oba tokova (uporabnikovega in tistega za vmesni pomnilnik) shranimo v eni od celic tabele Zahteve, za indeks pa uporabimo številko toka, po kateri prihajajo podatki do nas iz Interneta (ta tok nam vrne podprogram Zahtevaj).

```
const MaxZahtev = 10000; { Največ dovoljenih istočasnih zahtev. }
type DvaTokova = record DoUporabnika, VPomnilnik: integer end;
var Zahteve: array [1..MaxZahtev] of DvaTokova;

procedure Zahtevek(Naslov: string; Tok: integer);
var
  Podatki: string;
  VmTok, NovTok: integer;
begin
```

```
  VmTok := Shranjen(Naslov, Podatki);
  if VmTok = 0 then begin
    Poslji(Podatki, Tok);
    { Ker v nalogi ni opisano, kako se tokove zapira, si mislimo, da se
      tok zapre, če mu pošljemo kos podatkov ničelne dolžine. Konec koncev
      se takšno delovanje zahteva tudi od našega podprograma Prispelo,
      pa tudi on pričakuje enako od tokov, s katerimi dela. }
    Poslji('', Tok);
  else begin
    NovTok := Zahtevaj(Naslov);
    Zahteve[NovTok].DoUporabnika := Tok;
    Zahteve[NovTok].VPomniknik := VmTok;
  end; {if}
end; {Zahtevek}

procedure Prispelo(Podatki: string; Tok: integer);
begin
  Poslji(Podatki, Zahteve[Tok].DoUporabnika);
  Poslji(Podatki, Zahteve[Tok].VPomnilnik);
  if Length(Podatki) = 0 then begin
    Zahteve[Tok].DoUporabnika := 0;
    Zahteve[Tok].VPomnilnik := 0;
  end; {if}
end; {Prispelo}
```

Opisana rešitev ima lahko težave v primerih, ko prispe kmalu po prvem zahtevku za nek naslov še en zahtevek na isti naslov. Recimo, da smo nekaj podatkov s tega naslova že dobili (in shranili v vmesni pomnilnik), ne pa vseh; kaj zdaj vrne funkcija Shranjen, ko jo pokličemo ob drugem prejetem zahtevku? Če pravi, da podatki za ta naslov še niso shranjeni, bomo poslali v Internet še eno zahtevo za ta naslov in tako po nepotrebnem tratili pasovno širino; če pa reče, da so podatki za ta naslov že na voljo, nam bo vrnila seveda le tisto, kar se je dotlej za ta naslov že nakapljalo, mi pa bomo to uporabniku posredovali naprej, kot da bi bili to že vsi podatki za ta naslov. Boljša rešitev bi bila, da bi uporabniku v takem primeru takoj posredovali vse podatke, ki so s tega naslova že prišli, obenem pa bi njegov tok dodali na seznam „porabnikov“ (ali „odjemalcev“ ali „poslušalcev“); podprogram Prispelo bi posredoval na novo prispele podatke vmesnemu pomnilniku in še vsem porabnikom. Vsaka celica tabele Zahteve bi poleg toka za pisanje v vmesni pomnilnik vsebovala še kazalec na začetek seznama porabnikov. Potrebovali bi še način, kako ugotoviti, katera celica tabele Zahteve se nanaša na posamezni naslov (če sploh katera); lahko bi si pomagali z razpršeno tabelo ali pa naprtili funkciji Zahtevaj, naj to opravi namesto nas in nam vrne številko toka, s katerega že zdaj prihajajo podatki za isto zahtevo.

REŠITVE NALOG TRETJEGA ZAKLJUČNEGA TEKMOVANJA IZ ZNANJA RAČUNALNIŠTVA

R1997.Z.1

N: 300

Definirajmo sosede nekega mesta kot vsa tista mesta, ki so z njim povezana neposredno. Če se postavimo v neko poljubno mesto, lahko takole odkrijemo vsa mesta, ki so povezana z njim: neko mesto je povezano z našim, če je naš sosed, pa tudi, če je sosed nekega takega mesta, ki je povezano z našim. Vsakič, ko za neko mesto prvič odkrijemo, da je povezano z našim, moramo torej pregledati še njegove sosede, saj so tudi oni povezani z našim mestom. Da ne bi istega mesta odkrivali po večkrat (če se da na primer od našega mesta do tistega priti po več poteh), si v neki tabeli (Obiskani v spodnjem programu) zapomnimo, če smo ga že odkrili. Spodnji program uporablja tabelo Sklad, da si zapomni, katera mesta je sicer že odkril, ni pa še preiskal njihovih sosedov. Ko neko mesto prvič odkrijemo, ga torej dodamo na ta sklad, nato pa bo prej ali slej že prišlo na vrsto, da si bomo ogledali še njegove sosede. Ko ni na skladu nobenega mesta več, pomeni, da smo odkrili vsa mesta, ki so povezana z našim. To je torej ena skupina mest, zdaj pa lahko vsa ta mesta v mislih pobrišemo (dovolj bo že to, da so v tabeli Obiskani označena kot obiskana) in se posvetimo kakšnemu drugemu mestu, s katerim doslej še nismo imeli opravka. Tako bomo odkrili naslednjo skupino in tako dalje; ko pa so označena vsa mesta, je naše delo končano.

Za zgoraj opisani postopek je torej koristno, če imamo za vsako mesto pri roki seznam njegovih sosedov. To lahko brez težav pripravimo iz seznama neposrednih povezav, ki ga preberemo iz vhodne datoteke: neposredna povezava od u do v pomeni, da je v sosed mesta u, mesto u pa je sosed mesta v. V tabeli StSosedov bomo hranili podatek o tem, koliko sosedov ima posamezno mesto. Sezname sosedov bomo zaradi učinkovitosti in preprostosti hranili kar v eni sami dolgi tabeli: na začetku bodo sosedje prvega mesta, nato sosedje drugega mesta in tako naprej. Sosedje mesta u se začnejo na indeksu PrviSosed[u]; teh vrednosti ni težko izračunati s pomočjo tabele StSosedov. Spodnji program upošteva še to, da bo moral teči v realnem načinu, kjer posamezna tabela ne more biti daljša od 64 KB. Če ima naše omrežje 30 000 neposrednih povezav, pomeni to skupno 60 000 sosedov, vsak od njih pa zasede v pomnilniku 2 byta, tako da bomo morali to tabelo razbiti na dva kosa. Zato element, ki bi moral priti na indeks x, hranimo v resnici v celici Sosedje[x **mod** 2]↑[x **div** 2].

```
program SkupineMest;
const
  MaxM = 30000; { Največje možno število mest. }
  MaxN = 30000; { Največje možno število neposrednih povezav. }
type
  SosedjeT = array [0..MaxN − 1] of integer;
```

```
  PSosedjeT = ↑SosedjeT;
  StSosedovT = array [1..MaxM] of word;
  ObiskaniT = array [1..MaxM] of boolean;
var
  { i-ta neposredna povezava povezuje mesti PovOd↑[i] in PovDo↑[i]. }
  PovOd, PovDo: PSosedjeT;
  { Sklad in tabelo Obiskani uporabljamo med preiskovanjem omrežja. }
  Sklad: PSosedjeT; Obiskani: ↑ObiskaniT;
  { Pri n neposrednih povezavah je lahko skupna dolžina vseh seznamov sosedov
    do 2 * n. Tolikšna tabela ne bi šla v en 64-KB segment, zato jo razbijmo na
    dva kosa. Element x zapišimo v Sosedje[x mod 2]↑[x div 2]. Sosedje mesta i
    so na indeksih od PrviSosed↑[i] do PrviSosed↑[i] + StSosedov↑[i] − 1. }
  Sosedje: array [0..1] of PSosedjeT;
  StSosedov, PrviSosed: ↑StSosedovT;
  f: text; n, m, i, u, v, SP, StSkupin: integer; j: word;
begin
  { Preberimo podatke o cestnem omrežju. }
  New(Sosedje[0]); New(Sosedje[1]);
  New(PovOd); New(PovDo); New(StSosedov); New(PrviSosed);
  Assign(f, 'input.txt'); Reset(f);
  ReadLn(f, m); ReadLn(f, n);
  for i := 1 to m do StSosedov↑[i] := 0;
  for i := 1 to n do begin
    ReadLn(f, u, v); PovOd↑[i] := u; PovDo↑[i] := v;
    StSosedov↑[u] := StSosedov↑[u] + 1;
    StSosedov↑[v] := StSosedov↑[v] + 1;
  end; {for}
  Close(f);
  { Zdaj za vsako mesto vemo, s kolikimi je neposredno povezano.
    Pripravimo sezname sosedov za vsako mesto. }
  PrviSosed↑[1] := 1;
  for i := 1 to m − 1 do PrviSosed↑[i + 1] := PrviSosed↑[i] + StSosedov↑[i];
  for i := 1 to m do StSosedov↑[i] := 0;
  for i := 1 to n do begin
    u := PovOd↑[i]; v := PovDo↑[i];
    { Dodajmo v med u-jeve sosede in u med v-jeve. Lahko je u = v in
      zato moramo StSosedov↑[u] povečati, še preden uporabimo StSosedov↑[v].
      No, lahko bi tudi že med branjem vhodne datoteke preskočili vse povezave
      od u do u in na koncu branja primerno zmanjšali n. }
    j := PrviSosed↑[u] + StSosedov↑[u];
    StSosedov↑[u] := StSosedov↑[u] + 1;
    Sosedje[j mod 2]↑[j div 2] := v;
    j := PrviSosed↑[v] + StSosedov↑[v];
    StSosedov↑[v] := StSosedov↑[v] + 1;
    Sosedje[j mod 2]↑[j div 2] := u;
  end; {for}
```

```
  Dispose(PovOd); Dispose(PovDo); New(Sklad); New(Obiskani);
  { Zdaj lahko poiščemo in preštejemo skupine povezanih mest. }
  StSkupin := 0; for i := 1 to m do Obiskani↑[i] := false;
  for i := 1 to m do if not Obiskani↑[i] then begin
    { Preglejmo vsa mesta, povezana z i. Na skladu si bomo zapomnili,
      kaj moramo še pregledati. SP je število mest na skladu. }
    StSkupin := StSkupin + 1; SP := 1; Sklad↑[SP] := i; Obiskani↑[i] := true;
    while SP > 0 do begin
      u := Sklad↑[SP]; SP := SP − 1;
      { Obiščimo zdaj vse u-jeve sosede, če jih nismo obiskali že kdaj prej. }
      for j := PrviSosed↑[u] to PrviSosed↑[u] + StSosedov↑[u] − 1 do begin
        v := Sosedje[j mod 2]↑[j div 2];
        if not Obiskani↑[v] then
          begin SP := SP + 1; Sklad↑[SP] := v; Obiskani↑[v] := true end;
      end; {for j}
    end; {while}
  end; {for i}
  { Izpišimo rezultate, počistimo pomnilnik. }
  Assign(f, 'output.txt'); Rewrite(F); WriteLn(f, StSkupin); Close(f);
  Dispose(Sosedje[0]); Dispose(Sosedje[1]); Dispose(StSosedov);
  Dispose(PrviSosed); Dispose(Sklad); Dispose(Obiskani);
end. {SkupineMest}
```

Ta program se ne meni za napake v vhodnih podatkih, ki jih omenja opomba na strani 301. Če bi hoteli dovoliti tudi mesto s številko 0, bi morali v definiciji tipov StSosedovT in ObiskaniT zamenjati 1..MaxM z 0..MaxM; zanke, ki gredo od 1 do m, bi morale iti od 0 do m; PrviSosed↑[1] := 0 bi zamenjali s PrviSosed↑[0] := 0; in da mesto s številko 0 v primerih, ko ga sploh ni v omrežju, ne bi dobilo samostojne skupine, bi morali na koncu pogledati, če je StSosedov↑[0] = 0 in v tem primeru zmanjšati StSkupin za 1.

R1997.Z.2

Mislimo si v ravnini nek pravokotnik velikosti $a \times b$, s stranicami, vzporednimi s koordinatnima osema; skratka takega, s kakršnimi se ukvarja naša naloga. Recimo, da se njegova leva stranica ne dotika nobene izmed danih n točk. Potem bi lahko ta pravokotnik premaknili malo v desno (toliko, da bi se začela leva stranica dotikati kakšne točke), pa ne bi bilo zaradi tega premika v njem nič manj točk kot prej, saj ni na levi strani nobena točka padla iz njega, na desni pa je vanj mogoče celo prišla kakšna nova. Enako bi lahko razmišljali za premik navzgor, če se spodnja stranica pravokotnika prvotno ni dotikala nobene točke. Ta razmislek nam pove, da je dovolj, če se pri iskanju pravokotnika, ki vsebuje največ točk, omejimo na tiste, ki imajo tako na spodnji kot na levi stranici kakšno točko (ni pa nujno, da sta to dve različni točki — lahko je ena sama, če je ravno v spodnjem levem oglišču pravokotnika).

N: 301

Recimo, da nas trenutno zanimajo pravokotniki, ki se z levo stranico dotikajo točke (x_0, y_0). Ker je višina pravokotnikov vedno b, mora biti y-koordinata spodnje stranice v tem primeru z intervala $[y_0 - b, y_0]$ (če bi bila nižje, bi bil cel pravokotnik prenizko, da bi se še lahko dotikal točke (x_0, y_0), če bi bila višje, pa bi se pravokotnik začel previsoko), y-koordinata zgornje stranice pa z $[y_0, y_0 + b]$. Širina naših pravokotnikov pa je a; torej, ko se zanimamo za pravokotnike s točko (x_0, y_0) na levi stranici, bomo imeli opravka le s točkami s področja $[x_0, x_0 + a] \times [y_0 - b, y_0 + b]$.

Recimo zdaj, da smo nekako prišli do seznama vseh točk s tega področja (v skrajni sili bi lahko naredili kar zanko po vseh n točkah in za vsako posebej preverili, če leži v njem): recimo tem točkam (x_i, y_i) za $i = 1, \ldots, m$ (med njimi je tudi točka (x_0, y_0). Pravokotniki, ki nas bodo zanimali, morajo imeti eno od njih na spodnji stranici; čim si izberemo to, je položaj pravokotnika že čisto določen in je treba le še prešteti, katere od teh točk so v njem. Lahko bi šli torej z i po vseh tistih indeksih, ki imajo $y_i \le y_0$ (saj vemo, da spodnja stranica ne sme biti višje od y_0, če hočemo imeti točko (x_0, y_0) na levi stranici), in pri vsakem prešteli, katere izmed teh m točk so v pravokotniku velikosti $a \times b$, ki ima (x_0, y_0) na levi in (x_i, y_i) na desni stranici (to so obenem tudi že vse točke v tem pravokotniku, saj leži le teh m točk na takem področju, da bi utegnile ležati v tem pravokotniku).

Zdaj torej že imamo preprosto rešitev naloge:

```
program Rozine1;
const MaxN = 10000;
type  TabelaI = array [1..MaxN] of integer;
      TabelaR = array [1..MaxN] of real;
var   PtX, PtY: ↑TabelaR;                { koordinate točk }
      N: integer;                        { število točk }
      A, B: real;                        { velikost poizvedovalnega okna }
      Neigh: ↑TabelaI; nNeigh: integer;  { okolica točke (X0, Y0) }
      i, j, k, Count, Best: integer; F: text;
      X0, Y0, Y1, YCur: real;
begin
  { Preberimo vhodne podatke. }
  New(PtX); New(PtY);
  Assign(F, 'input.txt'); Reset(F);
  ReadLn(F, A, B); ReadLn(F, N);
  for i := 1 to N do ReadLn(F, PtX↑[i], PtY↑[i]);
  Close(F);
  { Pojdimo po vseh točkah. }
  Best := 0; New(Neigh);
  for i := 1 to N do begin
    { Pregledali bomo pravokotnike z i-to točko, (X0, Y0), na levi stranici.
      V Neigh pripravimo točke z X0 ≤ X ≤ X0 + A, Y0 − B ≤ Y ≤ Y0 + B;
      samo te lahko namreč ležijo v kakšnem takem pravokotniku. }
```

```
    X0 := PtX↑[i]; Y0 := PtY↑[i]; nNeigh := 0;
    for j := 1 to N do
      if    (X0 <= PtX↑[j])       and (PtX↑[j] <= X0 + A)
      and  (Y0 - B <= PtY↑[j]) and (PtY↑[j] <= Y0 + B)
      then begin nNeigh := nNeigh + 1; Neigh↑[nNeigh] := j end;
    { Poskusimo vse možne položaje spodnje stranice pravokotnika. }
    for j := 1 to nNeigh do begin
      Count := 0; Y1 := PtY↑[Neigh↑[j]];
      if Y1 > Y0 then
        continue; { Brez tega pogoja porabi program precej več časa! }
      { Koliko točk pokriva pravokotnik s spodnjo stranico pri Y1? }
      for k := 1 to nNeigh do begin
        YCur := PtY↑[Neigh↑[k]];
        if (Y1 <= YCur) and (YCur <= Y1 + B) then Count := Count + 1;
      end; {for k}
      if Count > Best then Best := Count;
    end; {for j}
  end; {for i}
  { Izpišimo rezultat in počistimo pomnilnik. }
  Assign(F, 'output.txt'); Rewrite(F); WriteLn(F, Best); Close(F);
  Dispose(Neigh); Dispose(PtX); Dispose(PtY);
end. {Rozine1}
```

Vendar pa je lahko ta rešitev, če imamo smolo, precej neučinkovita. Če je poizvedovalno območje $a \times b$ dovolj veliko, točke pa dovolj blizu skupaj, se bo pri mnogih (recimo pri $O(n)$) vrednostih i zgodilo, da bo v okolici (x_0, y_0) veliko (recimo $O(n)$) točk, tako da bosta gnezdeni zanki po j in k vsakič zahtevali $O(n^2)$ časa in je skupna časovna zahtevnost našega postopka zato v najslabšem primeru $O(n^3)$, kar je že zelo počasi. (Preprosteje povedano, naš program ima tri gnezdene zanke, ki gredo lahko v najslabšem primeru vse od 1 do n.) Če je v okolici točke (x_0, y_0) (torej: v pravokotniku $[x_0, x_0 + a] \times [y_0 - b, y_0 + b]$) povprečno m točk, pa imata gnezdeni zanki po j in k skupaj zahtevnost $O(m^2)$, pred tem pa porabimo še $O(n)$ časa, da ugotovimo, katere točke so v okolici; zahtevnost celotnega postopka je zato $O(n(n + m^2))$.

Postopek bi lahko izboljšali takole: ko postavljamo spodnjo stranico pravokotnika na vse možne točke v področju $[x_0, x_0+a]\times[y_0-b, y_0]$, moramo pri vsaki prešteti, koliko točk potem vsebuje tak pravokotnik. Toda če se spodnja stranica le malo premakne, vsebuje pravokotnik še vedno približno iste točke kot prej. Če bi torej točke, na katere postavljamo spodnjo stranico, pregledovali po naraščajočih y-koordinatah, bi ob vsakem premiku spodnje stranice kakšna točka izpadla iz pravokotnika, vrhnji del pravokotnika bi pokril nekaj novih, večina pa bi ostala nespremenjenih. Izpadla bi le tista, ki je bila pri prejšnjem položaju ravno na spodnji stranici (po novem pa se je spodnja stranica premaknila na naslednjo višjo točko in torej tista prejšnja ni več v pravokotniku), na vrhu pa bi prišlo vanjo nekaj novih. Zdaj torej za štetje, katere so v pra-

vokotniku, ni treba uporabiti zanke s števcem k, ki bi šla po vseh m točkah v okolici, pač pa lahko k le pogleda, koliko točk je na novo prišlo v pravokotnik. Zdaj zanki s števcema j in k pravzaprav potekata istočasno (prepleteno), ne pa vgnezdeno; v času, ko pride j od 1 do m, pride tudi k ravno enkrat od 1 do m. Urejanje točk po y-koordinatah lahko opravimo na samem začetku, čim preberemo točke iz vhodne datoteke; nato bomo že ob pripravi soseščine (prva zanka po j) dobili sosednje točke v naraščajočem vrstnem redu y-koordinat. Časovna zahtevnost tako spremenjenega postopka je le še $O(n^2)$ (za urejanje porabimo največ $O(n^2)$, s kakšnim učinkovitejšim algoritmom še manj; nato pa pri vsaki točki porabimo $O(n)$, da ugotovimo, katere so v njeni okolici, in nato še $O(m)$, da se z intervalom višine b sprehodimo čez te okoliške točke). Spodnji program vsebuje tri znane algoritme za urejanje točk po koordinatah: urejanje z izbiranjem (*selection sort*), urejanje z vstavljanjem (*insertion sort*) in quicksort; slednji je hitrejši od prvih dveh, vendar je tudi bolj zapleten.

```
program Rozine2;

const MaxN = 10000;
type  TabelaI = array [1..MaxN] of integer;
      TabelaR = array [1..MaxN] of real;
var   PtX, PtY: ↑TabelaR; { koordinate točk }
      N: integer;         { število točk }
      A, B: real;         { velikost poizvedovalnega okna }

  procedure SelectionSort; { Uredi točke po naraščajočih Y-koordinatah. }
  var i, j, k: integer; Y0, Y1, T: real;
  begin
    for i := 1 to N do begin
      { Naj bo k najnižja točka izmed točk i..n. }
      Y0 := PtY↑[i]; k := i;
      for j := i + 1 to N do begin
        Y1 := PtY↑[j];
        if Y1 < Y0 then begin Y0 := Y1; k := j end;
      end; {for j}
      { Zamenjajmo k s točko i. }
      T := PtX↑[i]; PtX↑[i] := PtX↑[k]; PtX↑[k] := T;
      T := PtY↑[i]; PtY↑[i] := PtY↑[k]; PtY↑[k] := T;
    end; {for i}
  end; {SelectionSort}

  procedure InsertionSort; { Uredi točke po naraščajočih Y-koordinatah. }
  var i, j: integer; X0, Y0: real;
  begin
    for i := 2 to N do begin
      { Zaporedje točk 1..i − 1 je že urejeno. Vstavimo vanj točko i. }
      j := i − 1; X0 := PtX↑[i]; Y0 := PtY↑[i];
```

```
        while j > 0 do begin
          if PtY↑[j] <= Y0 then break;
          PtX↑[j + 1] := PtX↑[j]; PtY↑[j + 1] := PtY↑[j]; j := j − 1;
        end; {while}
        PtX↑[j + 1] := X0; PtY↑[j + 1] := Y0;
      end; {for}
    end; {InsertionSort}

    procedure QuickSort(L, R: integer); { Uredi točke L..R po naraščajočih Y. }
    var i, j: integer; T, M: real;
    begin
      while L < R do begin
        M := PtY↑[(L + R) div 2]; i := L; j := R;
        { Razdelimo točke na tiste pod in tiste nad Y = M. }
        while i < j do begin
          { Invarianta: točke L..i imajo Y ≤ M, točke j..R pa Y ≥ M. }
          while PtY↑[i] < M do i := i + 1;
          while PtY↑[j] > M do j := j − 1;
          if i <= j then begin
            T := PtX↑[i]; PtX↑[i] := PtX↑[j]; PtX↑[j] := T;
            T := PtY↑[i]; PtY↑[i] := PtY↑[j]; PtY↑[j] := T;
            i := i + 1; j := j − 1;
          end; {if}
        end; {while}
        { Točke L..j imajo Y ≤ M, točke i..R pa Y ≥ M.
          Manjšo od teh dveh skupin uredimo z rekurzivnim klicem, večje pa
          se bomo lotili v naslednji ponovitvi zunanje zanke while. }
        if j − L < R − i then begin QuickSort(L, j); L := i end
        else begin QuickSort(i, R); R := j end;
      end; {while}
    end; {QuickSort}

var Neigh: ↑TabelaI; nNeigh: integer; { točke v okolici trenutne točke }

    { V Neigh vpiše točke z X0 ≤ X ≤ X0 + A, Y0 − B ≤ Y ≤ Y0 + B. }
    procedure FindNeigh1(X0, Y0: real);
    var i: integer;
    begin
      nNeigh := 0;
      for i := 1 to N do
        if    (X0 <= PtX↑[i])       and (PtX↑[i] <= X0 + A)
        and  (Y0 − B <= PtY↑[i]) and (PtY↑[i] <= Y0 + B)
        then begin nNeigh := nNeigh + 1; Neigh↑[nNeigh] := i end;
    end; {FindNeigh1}

var i, j, k, Best: integer; F: text; Y1: real;
begin {Rozine2}
```

```
  { Preberimo vhodne podatke. }
  New(PtX); New(PtY);
  Assign(F, 'input.txt'); Reset(F);
  ReadLn(F, A, B); ReadLn(F, N);
  for i := 1 to N do ReadLn(F, PtX↑[i], PtY↑[i]);
  Close(F);
  { Uredimo točke po naraščajočih Y-koordinatah. }
  InsertionSort; {QuickSort(1, N);} {SelectionSort;}
  { Pojdimo po vseh točkah. }
  Best := 0; New(Neigh);
  for i := 1 to N do begin
    { Preglejmo pravokotnike, ki imajo točko i na levi stranici. V Neigh pripravimo
      vse točke, ki bi utegnile biti v kakšnem takem pravokotniku. }
    FindNeigh1(PtX↑[i], PtY↑[i]);
    { Poskusimo vse možne položaje spodnje stranice pravokotnika. }
    k := 2;
    for j := 1 to nNeigh do begin
      Y1 := PtY↑[Neigh↑[j]];
      { Pravokotnik s spodnjo stranico pri Y1 pokriva točke Neigh↑[j..k − 1]. }
      while k <= nNeigh do
        if PtY↑[Neigh↑[k]] > Y1 + B then break else k := k + 1;
      if k − j > Best then Best := k − j;
    end; {for j}
  end; {for i}
  { Izpišimo rezultat in počistimo pomnilnik. }
  Assign(F, 'output.txt'); Rewrite(F); WriteLn(F, Best); Close(F);
  Dispose(Neigh); Dispose(PtX); Dispose(PtY);
end. {Rozine2}
```

Doslej smo, ko je bilo treba našteti točke iz območja $[x_0, x_0+a]\times[y_0-b, y_0+b]$, šli vsakič kar po vseh n točkah in za vsako preverili, če leži v njem. Toda zdaj, ko imamo točke urejene po y-koordinatah, si lahko privoščimo naslednjo izboljšavo: z bisekcijo poiščemo najnižjo točko, ki ima $y \geq y_0 - b$, nato pa pojdimo po vrsti od te točke naprej in za vsako preverimo, če ustreza tudi pogoju $x_0 \leq x \leq x_0 + b$. Ko pridemo do kakšne z $y > y_0 + b$, pa lahko takoj nehamo, saj vemo, da ne bomo našli nobene točke s primernim y več. Naštevanje točk v okolici nam zdaj v najslabšem primeru sicer še vedno lahko vzame $O(n)$ časa (če je veliko točk na pasu $y_0 - b \leq y \leq y_0 + b$), v praksi pa bo šlo najbrž vendarle hitreje (bisekcija sama zahteva le $O(\lg n)$ časa). V gornjem programu Rozine2 bi morali funkcijo FindNeigh1 zamenjati z:

```
procedure FindNeigh2(X0, Y0: real);
var i, L, R: integer;
begin
  { Bisekcija. }
  L := 1; R := N + 1;
```

```
    while L + 1 < R do begin
      { Invarianta: L = 1 ali pa PtY↑[L] ≤ Y0 − B < PtY↑[R]. }
      i := (L + R) div 2;
      if PtY↑[i] <= Y0 − B then L := i else R := i;
    end; {while}
    { Točke z Y ≥ Y0 + B se nahajajo na indeksih od L naprej. }
    nNeigh := 0;
    while L <= N do begin
      if PtY↑[L] > Y0 + B then break;
      if (X0 <= PtX↑[L]) and (PtX↑[L] <= X0 + A) then
        begin nNeigh := nNeigh + 1; Neigh↑[nNeigh] := L end;
      L := L + 1;
    end; {while}
  end; {FindNeigh2}
```

Razmislimo še o naslednji izboljšavi: ko razmišljamo o pravokotnikih, ki imajo na levi stranici točko (x_0, y_0), nas zanimajo le točke iz pasu $x_0 \le x \le x_0 + a$; med temi točkami pa nas zanimajo le tiste, za katere velja še $y_0 - b \le y \le y_0 + a$. Da bomo poceni prišli do teh slednjih, je koristno imeti točke pasu urejene po y-koordinatah; da pa ne bi bilo treba pri vsakem x_0 znova ugotavljati, katere točke ležijo v pasu, je koristno točke pregledovati po naraščajočem x_0. Tako se pas pravzaprav počasi premika proti desni in ob vsakem premiku iz njega izpade dotedanja točka x_0, na desni strani pa lahko v pas pride kakšna nova točka. Od podatkovne strukture, s katero bomo predstavili množico točk v pasu $x_0 \le x \le x_0 + a$, torej pričakujemo učinkovite operacije za dodajanje in brisanje točk ter učinkovito naštevanje točk, ki ustrezajo pogoju $y_0 - b \le y \le y_0 + b$. Zato je koristno uporabiti binarno iskalno drevo, v katerem bodo točke urejene po naraščajoči y-koordinati. Takšno drevo podpira brisanje in dodajanje v času $O(\lg n)$, naštevanje točk z intervala $[y_0 - b, y_0 + b]$ pa v $O(m + \lg n)$, če je treba našteti m točk.[59] Zdaj imamo torej tak postopek: ravnino „pometamo“ s pasom $x_0 \le x \le x_0 + a$; ko se pas premakne, pade ena točka iz njega, vanj pa pride nič ali več novih; na novem položaju naštejemo točke, ki ležijo na pasu in obenem ustrezajo pogoju $y_0 - b \le y \le y_0 + b$; prek teh pa se potem sprehodimo z dvema prepletenima zankama, tako kot pri prejšnji rešitvi. Zdaj imamo torej vsega skupaj $O(n)$ operacij z drevesom (vsako točko enkrat dodamo vanj in enkrat zbrišemo iz njega), kar zahteva skupaj $O(n \lg n)$ časa; toliko časa potrebujemo tudi za urejanje (če nimamo smole s quicksortom); in pri vsaki točki je še $O(m + \lg n)$ dela za naštevanje

[59] To dvoje se zanaša na predpostavko, da se nam drevo ne bo preveč izrodilo, ampak to se v praksi konec koncev res ne dogaja pretirano pogosto; če bi hoteli biti res varni, bi morali uporabiti kakšno bolj zapleteno različico binarnega iskalnega drevesa, npr. AVL-drevo ali rdeče-črno drevo. Lahko pa bi namesto tega (po zgledu drevesa segmentov s strani 359) izkoristili dejstvo, da že vnaprej točno vemo, kakšnih n možnih vrednosti utegne imeti y-koordinata točke, ki jo bomo dodali v drevo.

okoliških točk in sprehod čeznje z intervalom višine b. Tako imamo skupaj časovno zahtevnost $O(n(m + \lg n))$.

```
program Rozine3;
const MaxN = 10000;
type  TabelaI = array [1..MaxN] of integer;
      TabelaR = array [1..MaxN] of real;
      PTabelaI = ↑TabelaI; PTabelaR = ↑TabelaR;
var   PtX, PtY: ↑TabelaR; { koordinate točk }
      N: integer;         { število točk }
      A, B: real;         { velikost poizvedovalnega okna }
      { binarno iskalno drevo, ki vsebuje točke trenutnega pasu }
      Left, Right, Parent: ↑TabelaI; Root: integer; { urejene so po Y-koordinatah }
      { točke, urejene po eni od koordinat }
      XOrder, YOrder: ↑TabelaI; { v XOrder so vse točke, v YOrder pa neka okolica }

  { Uredi elemente tabele Order↑[L..R]. Elementi naj bodo indeksi v tabelo Coord,
    primerja pa se jih po ustrezni vrednosti v Coord. }
  procedure QuickSort(Coord: PTabelaR; Order: PTabelaI; L, R: integer);
  var i, j, T: integer; M: real;
  begin
    while L < R do begin
      M := Coord↑[Order↑[(L + R) div 2]]; i := L; j := R;
      while i < j do begin
        while Coord↑[Order↑[i]] < M do i := i + 1;
        while Coord↑[Order↑[j]] > M do j := j − 1;
        if i <= j then begin T := Order↑[i]; Order↑[i] := Order↑[j];
                             Order↑[j] := T; i := i + 1; j := j − 1 end;
      end; {while}
      if j − L < R − i then begin QuickSort(Coord, Order, L, j); L := i end
      else begin QuickSort(Coord, Order, i, R); R := j end;
    end; {while}
  end; {QuickSort}

  { V Dest doda (in poveča Count) vsa vozlišča (v poddrevesu, ki se
    začne pri Node), ki imajo Y-koordinato vsaj YMin in največ YMax. }
  procedure TreeQuery(Dest: PTabelaI; var Count: integer;
                      YMin, YMax: real; Node: integer);
  var YNode: real;
  begin
    YNode := PtY↑[Node];
    if (Left↑[Node] > 0) and (YMin <= YNode) then
      TreeQuery(Dest, Count, YMin, YMax, Left↑[Node]);
    if (YMin <= YNode) and (YNode <= YMax) then
      begin Count := Count + 1; Dest↑[Count] := Node end;
    if (Right↑[Node] > 0) and (YNode <= YMax) then
      TreeQuery(Dest, Count, YMin, YMax, Right↑[Node]);
```

```
  end; {TreeQuery}

  procedure TreeAdd(Node: integer);       { Doda vozlišče Node v drevo. }
  var P: integer; YParent, YNode: real;
  begin
    Left↑[Node] := 0; Right↑[Node] := 0;
    if Root = 0 then begin Root := Node; exit end;
    P := Root; YNode := PtY↑[Node];
    repeat
      YParent := PtY↑[P];
      if YNode < YParent then begin
        if Left↑[P] = 0 then begin Left↑[P] := Node; break end;
        P := Left↑[P];
      end else begin
        if Right↑[P] = 0 then begin Right↑[P] := Node; break end;
        P := Right↑[P];
      end; {if}
    until false;
    Parent↑[Node] := P;
  end; {TreeAdd}

  procedure TreeDel(Node: integer);       { Zbriše vozlišče Node iz drevesa. }
  var LC, RC, Next, P: integer;
  begin
    LC := Left↑[Node]; RC := Right↑[Node]; P := Parent↑[Node];
    if (LC = 0) or (RC = 0) then begin
      if LC = 0 then Next := RC else Next := LC;
    end else begin
      Next := RC;
      while Left↑[Next] > 0 do Next := Left↑[Next];
      TreeDel(Next);
      RC := Right↑[Node];
      Parent↑[LC] := Next; if RC > 0 then Parent↑[RC] := Next;
      Left↑[Next] := LC; Right↑[Next] := RC;
    end; {if}
    if Next > 0 then Parent↑[Next] := P;
    if P = 0 then Root := Next
    else if Left↑[P] = Node then Left↑[P] := Next
    else { Right↑[P] = Node } Right↑[P] := Next;
  end; {TreeDel}

var i, Best: integer; F: text;
    X0, Y0: real;           { koordinati trenutne točke; X0 določa levi rob pasu }
    Y1: real;               { trenutna koordinata spodnjega roba pravokotnika }
    XFrom, XTo: integer;    { v pasu so točke XOrder↑[XFrom..XTo − 1] }
    YCount: integer;        { koliko točk pasu ima Y-koordinato blizu Y0 }
    YFrom, YTo: integer;    { trenutni pravokotnik X0 ≤ X ≤ X0 + A, Y1 ≤ Y ≤ Y0 + B
```

```
                                    pokriva točke YOrder↑[YFrom..YTo − 1] }
begin {Rozine3}
  { Preberimo vhodne podatke. }
  New(Left); New(Right); New(Parent);
  New(XOrder); New(YOrder); New(PtX); New(PtY);
  Assign(F, 'input.txt'); Reset(F);
  ReadLn(F, A, B); ReadLn(F, N);
  for i := 1 to N do ReadLn(F, PtX↑[i], PtY↑[i]);
  Close(F);
  { Uredimo točke po naraščajočem X. }
  for i := 1 to N do XOrder↑[i] := i;
  QuickSort(PtX, XOrder, 1, N);

  { Drevo je na začetku prazno. Pometajmo ravnino.}
  { Začetni položaj levega roba pasu je tak, da gre skozi najbolj levo točko. }
  Best := 1; XFrom := 0; XTo := 1; Root := 0;
  while XFrom <= N − Best do begin

    { Premaknimo levi rob pasu do naslednje točke (proti desni).
      Prejšnja zato izpade iz pasu, lahko pa vanj pride nekaj novih. }
    if XFrom > 0 then TreeDel(XOrder↑[XFrom]);
    XFrom := XFrom + 1; X0 := PtX↑[XOrder↑[XFrom]];
    while XTo <= N do begin
      if PtX↑[XOrder↑[XTo]] > X0 + A then break;
      TreeAdd(XOrder↑[XTo]); XTo := XTo + 1;
    end; {while}
    if XTo − XFrom <= Best then continue; { Brezupno. }

    { Zanimali nas bodo pravokotniki, ki se z levo stranico dotikajo
      točke XOrder↑[XFrom]. Pripravimo si seznam točk,
      ki so v pasu in imajo Y0 − B ≤ Y ≤ Y0 + B. }
    Y0 := PtY↑[XOrder↑[XFrom]]; YCount := 0;
    TreeQuery(YOrder, YCount, Y0 − B, Y0 + B, Root);
    if YCount <= Best then continue; { Brezupno. }

    { Preletimo ta seznam z intervalom širine B. }
    YFrom := 0; YTo := 1;
    while (YFrom <= YCount − Best) and (YTo <= YCount) do begin
      { Interval bo pokrival koordinate od Y1 do Y1 + B. }
      YFrom := YFrom + 1; Y1 := PtY↑[YOrder↑[YFrom]];
      while YTo <= YCount do
        if PtY↑[YOrder↑[YTo]] > Y1 + B then break else YTo := YTo + 1;
      { Na intervalu so točke YOrder↑[YFrom..YTo − 1]. }
      if YTo − YFrom > Best then Best := YTo − YFrom;
    end; {prelet seznama YOrder z intervalom višine B}

  end; {prelet ravnine s pasom širine A}

  { Izpišimo rezultat in počistimo pomnilnik. }
  Assign(F, 'output.txt'); Rewrite(F); WriteLn(F, Best); Close(F);
```

```
  Dispose(Left); Dispose(Right); Dispose(Parent);
  Dispose(XOrder); Dispose(YOrder); Dispose(PtX); Dispose(PtY);
end. {Rozine3}
```

Opisane rešitve smo preizkusili na računalniku s procesorjem PIII 800 MHz; testni podatki so bili prav taki, kot so jih uporabljali na tekmovanju leta 1997. Spodnje meritve kažejo pravzaprav le porabo procesorjevega časa, saj odpade zaradi dovolj velikega diskovnega predpomnilnika na branje vhodnih podatkov in izpisovanje rezultatov le zanemarljivo malo časa. Program je bil prilagojen tako, da je v zanki obdelal vseh deset testnih primerov in ga ni bilo treba poganjati za vsak testni primer posebej. Pognali smo ga večkrat (desetkrat pri Rozine1 in Rozine2 + FindNeigh1 ter petdesetkrat pri Rozine2 + FindNeigh2 in Rozine3) in izračunali povprečni čas obdelave vseh desetih testnih primerov.

Rozine1	9,70 s			
Rozine2		SelectionSort	InsertionSort	QuickSort
FindNeigh1		5,28 s	4,66 s	4,52 s[60]
FindNeigh2		2,02 s	1,40 s	0,337 s
Rozine3	0,200 s			

Podobna naloga, le da temelji na diskretni mreži, je tudi #10360 v zbirki na `online-judge.uva.es` (4. naloga s študentskega tekmovanja tehnične univerze v Darmstadtu, 23. junija 2001).

[60] Tu se bo bralec mogoče vprašal, zakaj je razlika med InsertionSort in QuickSort v kombinaciji s FindNeigh1 tako majhna, v kombinaciji s FindNeigh2 pa tako velika. V resnici je razlika med trajanjem urejanja v obeh primerih enaka, težava pa je v tem, da porabi FindNeigh1 za svoje delo precej več časa, če so bili podatki urejeni s quicksortom, kot pa če so bili urejeni z vstavljanjem. Levji delež te razlike med InsertionSort + FindNeigh1 in QuickSort + FindNeigh1 nastopi pri enem samem patološkem testnem primeru, v katerem so točke kar vsi pari (x, y) za $x, y \in \{1, \ldots, 100\}$, v vhodni datoteki pa so navedeni po naraščajočih x in pri vsakem x še po naraščajočih y. Urejanje z vstavljanjem je stabilno in torej s tem, ko uredi točke po y, ostanejo pri vsakem y urejene tudi po naraščajočem x; quicksort pa jih sicer uredi po y, vendar jih ob tem znotraj vsakega y tudi pošteno premeče, tako da niso več urejene naraščajoče po x. Ker se izvedejo v stavku **if** v FindNeigh1 obakrat natanko iste primerjave pri istih i (da nam ne bi prevajalnik mešal štrene, smo tisti pogoj celo razcepili v več stavkov oblike **if not** DelPogoja **then continue**), si razlike v času izvajanja ne znamo razlagati drugače kot s tem, da je pri dostopanju do podatkov, urejenih s quicksortom, procesor pogosteje čaka, da bo dobil podatke iz pomnilnika ali predpomnilnika. To potrjuje tudi naslednji poskus: če v pogoju stavka **if** v podprogramu FindNeigh1 najprej preverimo koordinato y in šele nato koordinato x, je čas izvajanja enak ne glede na postopek, s katerim smo podatke urejali; to je smiselno, saj v tem primeru uspe pogoj glede koordinate PtY↑[i] obakrat pri natanko istih i in zato tudi koordinato PtX↑[i] preberemo obakrat pri natanko istih i, tako da je vzorec dostopov do predpomnilnika obakrat enak (je pa čas izvajanja podprograma FindNeigh1 zdaj počasnejši kot prej, celo kot prej pri quicksortu). Kakorkoli že, če QuickSort spremenimo tako, da primerja dve točki še po x, če sta imeli enak y, traja FindNeigh1 potem prav toliko, kot če bi uporabili urejanje z vstavljanjem, tako da se številka 4,52 s v gornji tabeli spremeni v 3,69 s.

N: 302

R1997.Z.3

Radi bi, da bi bile vse števke na velikem in malem števcu različne. Ker na veliki števec ne moremo vplivati drugače kot z vožnjo, moramo vsekakor prevoziti toliko kilometrov, da bo število na velikem števcu sestavljeno iz samih različnih števk. Ko pridemo do kakega takega števila (recimo x), nam ostanejo še štiri števke, ki se ta hip na velikem števcu ne pojavljajo in morajo torej nastopiti na malem števcu. Za mali števec obstaja torej zdaj (pri trenutnem stanju velikega števca) $4! = 24$ možnih vrednosti.[61] Za vsako od njih moramo preveriti, ali bi lahko mali števec nekako spravili v to stanje nekoč v tistem času, ko bo imel veliki števec vrednost x. Tu sta dve možnosti. (1) Do vrednosti m lahko mali števec pride, če smo prevozili vsaj $\lfloor m/10 \rfloor$ kilometrov, saj lahko v tem primeru mali števec v primernem trenutku (namreč ravno $\lfloor m/10 \rfloor$ kilometrov prej, preden je dosegel veliki števec svojo trenutno vrednost x) resetiramo, pa bo imel še dovolj časa, da bo prilezel od 0000 do vrednosti $10\lfloor m/10 \rfloor$ ravno v trenutku, ko bo veliki števec dosegel vrednost x; nato pa lahko prevozimo še (m mod 10)-krat po sto metrov, pa bo veliki števec še vedno imel isto vrednost, mali pa bo prišel na želeno vrednost m. (2) Druga možnost je, da imamo srečo in se mali števec znajde v pravem stanju že kar sam od sebe, ne da bi ga kaj spreminjali. Če je imel na začetku vrednost m_0, mi pa smo prevozili d kilometrov, bo imel v trenutku, ko dobi veliki števec vrednost x, mali števec vrednost $M := (m_0 + 10d) \bmod 10000$, v okviru tega kilometra pa bo dobil še naslednjih devet vrednosti, torej do vključno $M' := (m_0 + 10d + 9) \bmod 10000$, preden se bo veliki števec premaknil naprej (na $x + 1$). Zdaj moramo torej preveriti, če je želena vrednost m nekje na tem intervalu: $M \le m \le M'$. Posebej bi morali obravnavati še primer, ko pade malemu števcu vrednost z 9999 na 0000; takrat je M' manjše od M, tako da je vrednost m mogoče doseči, če je ali $M \le m$ ali pa $m \le M'$. Toda v tem primeru je M zagotovo ≥ 9991 (sicer še ne bi prišlo do padca na 0000), M' pa je zagotovo ≤ 0008 (to vrednost dosežemo, če je bil $M = 9999$; če je bil M manjši, pa še tega ne). Tedaj imajo torej vsi m, ki ustrezajo eni od enačb $M \le m$ in $m \le M'$, prve tri števke enake, tak pa naš iskani m zagotovo ne bo, saj mora vendar imeti štiri različne števke. Zato se nam s tem posebnim primerom v resnici ni treba ukvarjati.

Spodnji program kar preprosto povečuje veliki števec v korakih po en kilometer, nato pa vsakič preveri, če ga sestavlja šest različnih števk; če je tako, pregleda vseh 24 možnih stanj malega števca in za vsako od njih preveri, če ga je mogoče kako doseči. Mogoče bi si lahko poskušali možne vrednosti velikega števca (torej take, ki imajo šest različnih števk) pripraviti tudi vnaprej v kakšni tabeli, da ne bi števca povečevali v korakih po en kilometer, ampak

[61] Na primer: če imamo na voljo števke 1, 2, 3 in 4, so možne naslednje vrednosti malega števca: 1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4321. Mimogrede, pri tem razmisleku bomo stanje malega števca vedno gledali kot celo število, torej kot da v njem ne bi bilo decimalne vejice.

bi vedno kar skočili na naslednjo vrednost s šestimi različnimi števkami; toda takih vrednosti je kar $10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 = 151200$, torej sploh ne malo, tako da je vprašljivo, če bi s tem res kaj prihranili.

Za naštevanje možnih vrednosti malega števca (pri danih štirih števkah, iz katerih mora biti sestavljen) moramo nekako priti do vseh 24 permutacij teh štirih števk. Čisto dobro bi bilo že, če bi v kakšni tabeli navedli vseh 24 načinov, kako razmestiti štiri števke. Lahko pa poskusimo permutacije tudi oštevilčiti od 0 do 23 in jih tako našteti. Tu ravnamo nekako tako, kot da bi imeli številski sestav s spremenljivo osnovo. Recimo, da bi gledali permutacije k elementov. Poglejmo, kateri po velikosti (štejmo jih od 1 do $k-1$) je element na prvem mestu, torej $\pi(1)$; to pomnožimo s $(k-1)!$. Nato poglejmo, kateri po velikosti ($0..k-2$) izmed vseh preostalih $k-1$ elementov je tisti, ki je na drugem mestu v permutaciji; to vrednost (0 naj predstavlja najmanjšega, 1 drugega najmanjšega in tako naprej) pomnožimo s $(k-2)!$. Nato poglejmo, kateri po velikosti izmed preostalih $k-2$ elementov je element na tretjem mestu; to pomnožimo s $(k-3)!$. Tako nadaljujemo do konca in nazadnje vse te zmnožke seštejemo. Oglejmo si na primer razpored 8957. Tu imamo štiri elemente, torej $k = 4$. Na prvem mestu je drugi najmanjši element, kar nam da $1 \cdot (k-1)! = 6$. Ostanejo elementi $9, 5, 7$ in na drugem mestu je 9, ki je tretji najmanjši med njimi; to nam da $2 \cdot (k-2)! = 4$. Ostaneta elementa 5 in 7; na tretjem mestu je manjši od obeh, kar nam da $0 \cdot (k-3)! = 0$. Ostane element 7, ki je pač sam pri sebi neizogibno tudi najmanjši in nam da $0 \cdot (k-4)! = 0$. Če vse to seštejemo, dobimo $6 + 4 + 0 + 0 = 10$, torej razporedu 8957 ustreza število 10. Na enak način bi lahko oštevilčili tudi druge razporeditve števk 8, 9, 5 in 7 ter tako dobili vsa cela števila od 0 do $4! - 1 = 23$. Naš program pa mora uporabiti ravno obraten postopek, da bo iz števil $0, \ldots, 23$ dobil konkretne razporede. V ta namen številko razporeda najprej delimo s $(k-1)!$; celi del količnika nam pove, kateri najmanjši element je na prvem mestu, ostanek pa potem po enakem postopku dekodiramo v razpored na ostalih mestih (od drugega mesta naprej).

```
program Stevec;
var F: text; i, j, k: longint;
    Km, Mali, KmDelta, M: longint; R: real;
    Stevke, Stevke0: array [0..9] of integer;
    Ostale: set of 0..9; Nasel: boolean;
begin
  Assign(F, 'input.txt'); Reset(F);
  ReadLn(F, Km, R); Close(F);
  { Pozor: uporabiti moramo Round, ne Trunc. Slednji vedno zaokroža navzdol,
    kar ni dobro, saj je lahko kakšno decimalno število, npr. 0,1, predstavljeno
    z nekim približkom, ki je manjši od njega in je potem tudi njegov
    desetkratnik manjši od ustreznega celega števila. Če bi ga zaokrožili
    navzdol, bi dobili za 1 premajhno vrednost. }
```

```
  Mali := Round(10 * R); KmDelta := 0;
  while true do begin
    { Katere števke se ne pojavljajo na velikem števcu? }
    Ostale := [0..9]; j := Km;
    for i := 0 to 5 do begin Ostale := Ostale − [j mod 10]; j := j div 10 end;
    { Shranimo jih v tabelo Stevke0. }
    j := 0; for i := 0 to 9 do if i in Ostale then
      begin Stevke0[j] := i; j := j + 1 end;
    { Če so ostale samo štiri števke, jih poskusimo na vseh 4! = 24 načinov
      sestaviti v stanje malega števca. }
    Nasel := false;
    if j = 4 then for i := 0 to 23 do begin
      { Pripravimo (v številu M) i-to permutacijo naših štirih števk. }
      for j := 0 to 3 do Stevke[j] := Stevke0[j];
      j := i div 6; k := i mod 6; M := 1000 * Stevke[j];
      while j < 3 do begin Stevke[j] := Stevke[j + 1]; j := j + 1 end;
      j := k div 2; k := k mod 2; M := M + 100 * Stevke[j];
      while j < 2 do begin Stevke[j] := Stevke[j + 1]; j := j + 1 end;
      if k = 1 then M := M + 10 * Stevke[1] + Stevke[0]
      else M := M + 10 * Stevke[0] + Stevke[1];
      { Mogoče lahko do te vrednosti malega števca pridemo tako, da ga med
        vožnjo v primernem trenutku resetiramo. Vožnja mora biti dovolj dolga,
        da bo imel po resetiranju čas priti od 0000 do vrednosti Mali. }
      if M div 10 <= KmDelta then
        begin Nasel := true; break end;
      { Mogoče pa bo dosegel mali števec to vrednost v tem kilometru že kar
        brez našega posredovanja. V trenutnem kilometru bo imel vrednosti
        od Mali do Mali + 9. }
      if (Mali <= M) and (M <= Mali + 9) then
        begin Nasel := true; break end;
    end; {for i}
    if Nasel then break;
    { Prevozimo še en kilometer. }
    Km := (Km + 1) mod 1000000; Mali := (Mali + 10) mod 10000;
    KmDelta := KmDelta + 1;
  end; {while}
  for i := 0 to 5 do begin Stevke[i] := Km mod 10; Km := Km div 10 end;
  Assign(F, 'output.txt'); Rewrite(F);
  for i := 5 downto 0 do Write(F, Stevke[i]);
  WriteLn(F); Close(F);
end. {Stevec}
```

Program se bo zagotovo vedno ustavil — primerne vrednosti velikega števca bomo vedno znova srečevali (ne more nam jih zmanjkati), prej ali slej pa bo naša vožnja postala tudi dovolj dolga (KmDelta dovolj velika), da bi lahko z resetiranjem malega števca v primernem trenutku poskrbeli, da bo imel ta na

koncu ravno želeno vrednost. Če rešimo nalogo za vseh 10^{10} možnih začetnih stanj obeh števcev,[62] se izkaže, da je treba prevoziti povprečno po 948,27 km, vendar je dolžina vožnje močno odvisna od začetnega stanja: standardna deviacija je 2694,68 km, najdaljša potrebna vožnja pa je dolga kar 24702 km (do tega pride v primerih, ko je veliki števec na začetku enak 987643 in moramo voziti tako dolgo, dokler ne pride v stanje 012345).

[62]To ne pomeni, da moramo zgornji program pognati 10^{10}-krat — z nekaj majhnimi spremembami nam lahko izračuna rešitve za vseh 10000 možnih začetnih stanj malega števca pri nekem danem začetnem stanju velikega števca; pri tem pa ne porabi 10000-krat toliko časa kot zgornji program za eno samo začetno stanje malega števca, pač pa le nekajkrat toliko (pri naših poskusih približno štirikrat toliko). Tako lahko do rešitev za vseh 10^{10} začetnih stanj obeh števcev pridemo v nekaj minutah.

22. državno tekmovanje v znanju računalništva (1998)

NALOGE ZA PRVO SKUPINO

R: 348

1998.1.1 **Kaj izpiše** naslednji **program**? Odgovor primerno utemelji!

```
program Ego;
const
  S: array [1..21] of string = (
    'program Ego;',
    'const',
    '  S: array[1..21] of string = (',
    '  );',
    'var',
    '  i, j: integer;',
    'begin',
    '  for i := 1 to 3 do',
    '    WriteLn(S[i]);',
    '  for i := 1 to 21 do begin',
    '    Write("    """, S[i], """");',
    '    if i < 21 then Write(",");',
    '    WriteLn;',
    '  end;',
    '  for i := 4 to 21 do begin',
    '    for j := 1 to Length(S[i]) do',
    '      if (i <> 18) or (j <> 23) then',
    '        if S[i, j] = """ then S[i, j] := """";',
    '    WriteLn(S[i]);',
    '  end;',
    'end.'
  );
var
  i, j: integer;
begin
  for i := 1 to 3 do
    WriteLn(S[i]);
  for i := 1 to 21 do begin
    Write('    ''', S[i], '''');
    if i < 21 then Write(',');
    WriteLn;
  end;
  for i := 4 to 21 do begin
    for j := 1 to Length(S[i]) do
```

```
        if (i <> 18) or (j <> 23) then
          if S[i, j] = '"' then S[i, j] := '''';
      WriteLn(S[i]);
    end;
end.
```

(*Opomba:* ni mišljeno, da bi tekmovalec odgovoril, da program ne izpiše ničesar, ker da se sploh ne prevede, saj poskuša občasno spreminjati vsebino tabele `S`, ta pa je deklarirana kot konstanta. V Turbo Pascalu konstante, ki so imele ob deklaraciji eksplicitno naveden tip, niso bile nič drugega kot spremenljivke z vnaprej določeno začetno vrednostjo, kasneje pa se jih je dalo spreminjati kot običajne spremenljivke.)

1998.1.2

V urejevalniku besedil WORD BUSTER imamo neko besedilo, ki ga želimo spremeniti, kot je zahtevano na koncu naloge. R: 350

Urejevalnik pozna pojem *trenutnega položaja* (nahajališče kazalca oziroma kurzorja). Če je v vrstici n znakov, je trenutni položaj vedno med 1 in $n + 1$: bolj desno od $(n + 1)$-vega znaka trenutni položaj ne more biti, prav tako ne bolj levo od začetka vrstice. V spodnjih primerih je trenutni položaj označen s črtico pod ustreznim znakom, npr. `takole`.

Na razpolago imaš naslednje ukaze urejevalnika:

`BRISI_KONEC` — zbriše znake v vrstici od vključno znaka, na katerem se trenutno nahajamo, do konca vrstice; zbrisane znake shrani v začasni pomnilnik. Trenutni položaj se ne spremeni.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEF_`

`BRISI_ZACETEK` — zbriše znake v vrstici pred trenutnim položajem; znaki od vključno trenutnega položaja do konca vrstice se premaknejo na začetek vrstice, novi trenutni položaj je 1. Zbrisani znaki se shranijo v začasni pomnilnik.

Primer: `ABCDEFGHIJKLMN` ⟶ `GHIJKLMN`

`BRISI_ZNAK` — zbriše znak na trenutnem položaju, če nismo na koncu vrstice. Znaki od trenutnega položaja do konca vrstice se premaknejo za en znak v levo; zbrisani znak se shrani v začasni pomnilnik. Trenutni položaj se ne spremeni.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEFHIJKLMN`

`VRINI_ZBRISANO` — na trenutno mesto vrine znake iz začasnega pomnilnika, ki jih je tja shranil zadnji od ukazov `BRISI_ZACETEK` ali `BRISI_KONEC` ali `BRISI_ZNAK`. Znaki v prvotni vrstici od vključno znaka, na katerem se trenutno nahajamo, pa do konca vrstice, se premaknejo v desno za število

vrinjenih znakov. Trenutni položaj se premakne skupaj s preostankom vrstice.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEFshranjenoGHIJKLMN`

`VRINI_PRESLEDEK` — na trenutno mesto vrine en presledek; trenutni položaj se premakne za eno mesto v desno skupaj s preostankom vrstice.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEF GHIJKLMN`

`DESNO` — premakne trenutni položaj za eno mesto v desno. Če se že nahajamo tik za zadnjim znakom v vrstici, se položaj ne spremeni.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEFGHIJKLMN`

`LEVO` — premakne trenutni položaj za eno mesto v levo. Če se že nahajamo na prvem mestu v vrstici, se položaj ne spremeni.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEFGHIJKLMN`

`NA_ZACETEK_NASLEDNJE_BESEDE` — premakne trenutni položaj na začetek naslednje besede ali na konec vrstice, če smo že pri zadnji besedi. Beseda je definirana kot strnjeno zaporedje znakov, ki niso presledki.

Primeri:
```
ABC DEFGHIJ   KLMN ⟶ ABC DEFGHIJ   KLMN
ABC DEFGHIJ   KLMN ⟶ ABC DEFGHIJ   KLMN
ABC DEFGHIJ _ KLMN ⟶ ABC DEFGHIJ   KLMN
```

`NA_KONEC_PREJSNJE_BESEDE` – premakne trenutni položaj na zadnji znak prejšnje besede, ali na začetek vrstice, če smo že pri prvi besedi.

Primeri:
```
ABC DEFGHIJ   KLMN ⟶ ABC DEFGHIJ   KLMN
ABC DEFGHIJ   KLMN ⟶ ABC DEFGHIJ   KLMN
ABC DEFGHIJ _ KLMN ⟶ ABC DEFGHIJ   KLMN
```

Pred vsakim ukazom lahko stoji število ponovitev. Na primer: „`10 DESNO`“ pomeni, naj se ukaz `DESNO` izvrši desetkrat. Za vse ukaze velja, da ne storijo nič, če svoje naloge ne morejo opraviti (npr. brisanje prazne vrstice, pomik levo od prvega znaka v vrstici in podobno).

V začetku je trenutni položaj na začetku vrstice, po končanem opravilu je lahko trenutni položaj kjerkoli v vrstici.

Primer: naslednje zaporedje ukazov prestavi prvo besedo v vrstici na konec vrstice:

```
NA_ZACETEK_NASLEDNJE_BESEDE, BRISI_ZACETEK, 1000 DESNO,
VRINI_PRESLEDEK, VRINI_ZBRISANO
```

Reši naslednji nalogi:

(a) V urejevalniku je besedilo, ki ga želimo desno poravnati tako, da na začetek vrstic vrinemo ustrezno število presledkov (vsi znaki so enako široki). Nobena vrstica danega besedila ni daljša od 70 znakov in ne vsebuje presledkov niti na začetku niti na koncu vrstic. Končno poravnano besedilo naj bo široko 80 znakov. **Napiši zaporedje ukazov**, ki trenutno vrstico prestavi (poravna) desno.

(b) V vsaki vrstici je zapisano neko število med nič in dvajset milijoni. **Napiši zaporedje ukazov**, ki število v trenutni vrstici naredi bolj čitljivo tako, da vrine presledek na vsaka tri mesta, kot kažejo primeri:

```
12345000   ⟶   12 345 000
567890     ⟶   567 890
43         ⟶   43
```

1998.1.3

Podjetja vsako leto zaslužijo nekaj denarja. Ob koncu leta, ko je obračun, morajo prikazati dobiček. Zaradi hecnih zakonov se jim ne splača prikazati dobička, ki je večji kot 1000 cekinov. Naše podjetje *„Hlevi softwearskih ljubiteljev“* (HSL) si vsako leto izplača največji dobiček, ki ne presega 1000 cekinov, preostanek denarja pa prenese v naslednje leto. **Napiši program**, ki prebere, koliko denarja je zaslužilo podjetje, izpiše pa naj dobiček ob koncu leta in koliko denarja se je preneslo v naslednje leto. Pri naslednjem letu seveda upoštevaj tudi denar, ki se je prenesel iz prejšnjega leta. Primer: Če imamo na vhodu naslednje zneske zaslužkov (brez komentarjev):

R: 351

`300`	prvo leto, začetek firme
`800`	drugo leto, vzpon
`1200`	tretje leto, vse cveti
`1500`	četrto leto, nič nas ne more ustaviti
`400`	peto leto, ops!
`100`	šesto leto, leto suhih krav
`50`	sedmo leto, životarjenje
`500`	osmo leto, morda pa le ni tako slabo
`1500`	deveto leto, povratek odpisanih

mora program izpisati naslednje zneske (brez komentarjev):

`300 0`	dobiček je 300, nič prenosa v naslednje leto
`800 0`	dobiček je 800, nič prenosa v naslednje leto
`1000 200`	dobiček je 1000 ($= 1200 - 200$), prenos je 200
`1000 700`	dobiček je 1000 ($= 1500 + 200 - 700$), prenos je 700
`1000 100`	dobiček je 1000 ($= 400 + 700 - 100$), prenos je 100
`200 0`	dobiček je 200 ($= 100 + 100$), prenosa ni
`50 0`	dobiček je 50, prenosa ni

`500 0`	dobiček je 500, prenosa ni
`1000 500`	dobiček je 1000 ($= 1500 - 500$), prenos je 500

R: 351

1998.1.4

Imamo merilno napravo, ki šteje dežne kapljice, ki padejo na merilno ploskev. S podprogramsko funkcijo StDeznihKapelj lahko odčitamo, koliko dežnih kapelj je padlo od prejšnjega klica te funkcije. Kadar le rahlo rosi, bi radi, da računalnik enkrat zapiska za vsako padlo kapljico in to takoj, ko jo merilna naprava zazna. Da ne bi bilo v nalivu piskanja preveč, postavimo dodatne pogoje:

- med dvema zaporednima piskoma mora miniti vsaj ena sekunda;
- če zaradi prejšnjega pogoja ni dovoljeno zapiskati takoj, lahko pisk zamuja, vendar ne več kot za 5 sekund;
- višek dežnih kapljic, ki jih zaradi gornjih dveh omejitev ni mogoče javiti, tiho spregledamo.

Napiši program, ki ob upoštevanju danih omejitev odčitava merilno napravo in piska. Pri tem naj ne troši procesorskega časa po nepotrebnem. Na voljo imaš naslednje podprograme:

function StDeznihKapelj(CakajNajvec: integer): integer;
Ta podprogramska funkcija odčita število dežnih kapelj, padlih na merilno napravo od prejšnjega klica, in to število vrne kot funkcijsko vrednost. Podprogram se vrne takoj, če je na zalogi kakšna še neprešteta kaplja, ali pa takrat, ko nova kaplja pade — vendar na kapljo ne čaka dlje kot CakajNajvec tisočink sekunde (CakajNajvec je lahko tudi 0). Če je CakajNajvec negativno število, čakanje ni časovno omejeno.

procedure Zapiskaj;
Sproži en pisk.

procedure Zaspi(Cakaj: integer);
Podprogram zaspi za Cakaj tisočink sekunde (v tem času lahko procesor izvaja druge programe) in se po tem času vrne.

NALOGE ZA DRUGO SKUPINO

R: 352

1998.2.1

Podane so celoštevilske koordinate n točk, ki ležijo v ravnini. Vsak par teh točk določa pravokotnik, ki ima s koordinatnima osema vzporedne stranice in ena točka leži v njegovem spodnjem levem

oglišču, druga pa v zgornjem desnem oglišču.[63] **Napiši program**, ki učinkovito poišče število takih pravokotnikov, ki so kvadrati. Štej tudi izrojene kvadrate, pri katerih spodnje levo in zgornje desno oglišče sovpadata.

Število točk n je manjše od 10^5.

Območje, na katerem ležijo točke v ravnini, je znotraj kvadrata:

$$-10\,000 \le x \le 10\,000, \qquad -10\,000 \le y \le 10\,000$$

Na voljo imaš največ 1 234 567 zlogov (bytov) pomnilnika.

1998.2.2

Kriptografija je veda, ki služi marsičemu — še najbolj jo poznamo iz filmov in knjig o dogajanjih med vojno ali o kakšnih tajnih agentih ali o čem podobnem. Prav nam pa pride lahko tudi v čisto vsakdanjem življenju, pri reševanju medsosedskih problemov. Predstavljajmo si na primer ulico z n ($n > 2$) sosedi, ki sicer vedno govorijo resnico, nočejo pa drug drugemu izdati, kakšno plačo ima kateri od njih. Ker bi se radi primerjali s prebivalci iz sosednje ulice, pa morajo vseeno na nek način izvedeti, koliko skupaj zaslužijo na mesec — pri tem seveda nočejo kršiti prej omenjene kaprice (da bi komurkoli izdali višino svoje plače). Razmislite in **opišite postopek** oz. postopke, kako bi to sosedi med seboj izvedli. Opišite tudi prednosti in slabosti naštetih postopkov.

R: 354

1998.2.3

Danih imate n pravokotnikov, podanih s koordinatami levega spodnjega in desnega zgornjega oglišča $\langle (x_1, y_1), (x_2, y_2) \rangle$. Koordinate so vse celoštevilske (tipa integer). **Opiši postopek**, ki izračuna skupno ploščino, ki jo prekrivajo pravokotniki. Pri tem seveda ne pozabite upoštevati, da se pravokotniki lahko tudi prekrivajo — v takih primerih prekrite ploščine ne smemo šteti dvo- ali večkratno. Program naj izračuna skupno ploščino s čim manj izvedenimi operacijami in s čim manj uporabljenega pomnilnika.

R: 355

Podatke o pravokotnikih dobite z dvema pomožnima funkcijama:

function StPravok: integer;
: Vrne n, število vseh pravokotnikov.

function DajKoord(i: integer; **var** x1, y1, x2, y2: integer);
: V spremenljivke x1, y1, x2, y2 vrne koordinate i-tega pravokotnika. Spremenljivki x1 in y1 predstavljata levo zgornje, x2 in y2 pa desno spodnje oglišče.

[63] Kaj pa, če imamo na primer točki $(0, 1)$ in $(1, 0)$? Tidve ležita v zgornjem levem in spodnjem desnem oglišču svojega pravokotnika. Gornje besedilo iz leta 1998 torej, če ga beremo dobesedno, obljublja, da se v vhodnih podatkih to ne bo zgodilo; na primer, če imamo točki (x, y) in (x', y'), pa je $x < x'$, bo gotovo veljalo tudi $y \le y'$. Verjetnejša razlaga pa je, da so sestavljalci naloge pozabili omeniti, naj tekmovalčeva rešitev tiste pare točk (x, y) in (x', y'), za katere je $x < x'$ in $y > y'$, pri štetju kvadratov pač ignorira.

R: 361

1998.2.4

V vsaki vrstici vhodne datoteke je zapisano eno ime (niz znakov) nekega računalnika v internetu; imena se ne ponavljajo. Program mora brati imena, za vsako ime poiskati njegov omrežni naslov (niz znakov), nato pa dobljene naslove zapisati v izhodno datoteko v nespremenjenem vrstnem redu.

Trajanje iskanja imenu pripadajočega naslova je zelo različno; večino časa pri tem izgubimo s čakanjem na tuje računalnike v omrežju. Zagotovljeno je, da dobimo podatek najkasneje v nekem določenem končnem času (v ilustracijo: podatek dobimo v 1 do 100 sekundah).

Da pospešimo delo, smo razvili paralelni del programa, ki je sposoben hkrati iskati do 64 naslovov, rezultate pa vrača takoj, ko so na voljo, torej v vrstnem redu, ki večinoma ni enak vrstnemu redu zastavljenih vprašanj.

Napiši postopek, ki bo bral imena, dodeljeval delo prostim paralelnim vejam programa, od njih pobiral rezultate in jih tako sproščal za novo nalogo ter rezultate takoj, ko bo možno, izpisoval v enakem vrstnem redu, kot so bili prebrani podatki (imena računalnikov). Upoštevaj, da je lahko vhodnih podatkov poljubno mnogo in da imaš na razpolago omejen pomnilnik, ki lahko hrani največ 1000 nizov (imen ali naslovov računalnikov). **Pojasni tudi**, kako si organiziral podatkovno strukturo, potrebno za urejanje rezultatov.

Na voljo imaš naslednja podprograma:

function Obdelaj(Ime: string): integer;

Podprogram poišče eno od 64 prostih vej programa in ji naloži novo opravilo (preslikavo imena v naslov). Podprogramska funkcija se vrne takoj in kot funkcijsko vrednost vrne zaporedno številko veje, ki je prevzela opravilo (število med 1 in 64). Če nobena veja ni prosta, vrne 0, opravila pa si ne zapomni.

function PoberiRezultat(**var** Naslov: string): integer;

Podprogram poišče eno od vej programa, ki je že opravila svojo nalogo (po potrebi počaka na prvo tako), ter vrne najdeni naslov, kot funkcijsko vrednost pa vrne številko veje, iz katere je prišel rezultat. S prevzemom rezultata se ta veja sprosti in čaka na novo opravilo.

NALOGE ZA TRETJO SKUPINO

R: 363

1998.3.1

Dan je načrt mestnih ulic — v bistvu graf, podan s seznami sosednjih križišč za posamezno križišče. V teh seznamih je vrstni red pomemben — sledijo si v smeri urinega kazalca. V posameznem križišču se srečajo tri ali štiri ulice (točke, kjer se srečata samo dve ulici, ne imenujemo križišče). Znane so tudi dolžine ulic. Mestni očetje so sprejeli odlok, da je odslej prepovedano zavijati na levo.

Sestavi oz. **opiši postopek**, ki bo poiskal najkrajšo pot, ki upošteva odlok, iz danega križišča v drugo dano križišče (oziroma ugotovil, da ta ne obstaja).

Lahko predpostavite, da imate že dano funkcijo, ki poišče najkrajšo pot med dvema križiščema, ki pa ne upošteva odloka. Ta funkcija obvlada tudi mestne načrte z enosmernimi ulicami, čeprav takih ulic v prvotnem mestnem načrtu ni. Naloga je torej lahko tudi: kako predelati dani načrt mestnih ulic tako, da boste lahko uporabili dano funkcijo in s tem odgovorili na osnovno vprašanje.

1998.3.2

R: 364

Pri podjetju Paranoid d. o. o. so izdelali elektronsko ključavnico, ki jo je moč odkleniti le s posebnim elektronskim ključem. Pri odklepanju preda ključavnica ključu vprašanje v obliki poljubnega 16-bitnega števila, na osnovi katerega ključ izračuna pripadajoč 128-bitni odgovor in ga vrne ključavnici. Če odgovor ustreza vprašanju, ključ odklene ključavnico.

Jedro elektronskega ključa je procesor Merlin, ki teče s taktom 571 MHz, v enem ciklu izvede en ukaz in ima poleg 16-bitnega programskega števca še dva 8-bitna registra. Poleg procesorja je v elektronskem ključu le še bralni pomnilnik (ROM), ki vsebuje program za izračun odgovorov.

Na računalnik imaš priklopljeno testno ključavnico, v katero lahko vložiš ključ. **Opiši postopek**, ki ugotovi, ali se program za izračun odgovorov pri poljubnem vprašanju vedno ustavi, na voljo pa imaš naslednji dve funkciji:

- **procedure** Vprasaj(Vprasanje: integer); — Preda ključu 16-bitno vprašanje. Ključ ob tem „resetira“ — če je dotlej kaj računal, ga prekine in ključ se začne takoj ukvarjati z novim vprašanjem.
- **function** Odgovor(**var** Odg: **array** [1..16] **of** byte): boolean — Vrne false, če ključ še računa, sicer pa vrne true in v Odg ključev odgovor na zadnje vprašanje.

1998.3.3

R: 364

Med n različnimi števili, zapisanimi v urejeni tabeli, bi radi poiskali tak par števil, da bo njuna vsota enaka 100. V spodnjem zaporedju je primeren par $10 + 90$.

2 7 10 14 24 31 35 44 45 60 67 69 75 90 95

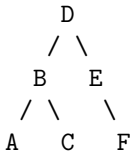
Kako se lotiti dela? Če bi slepo preizkušali vse možne pare, jih seštevali in preverjali njihove vsote, bi za n števil potrebovali $\frac{1}{2}n(n-1)$ seštevanj — že pri tisoč številih bi bilo seštevanj skoraj pol milijona, pri milijon številih pa že pol bilijona...

Gre hitreje? Za začetek poskusi kar na roko poiskati še kak par z vsoto 100 v gornji tabeli, nato pa izumljeni postopek zapiši v obliki podprograma IzpisiPare(a: **array** [1..n] **of** integer), ki izpiše vse primerne pare. Če ti uspe nalogo rešiti z manj kot n seštevanji, si na pravi poti.

R: 365

1998.3.4

Ko je bila Valentina še majhna, se je rada igrala z binarnimi drevesi. Tvorila je drevesa s samimi različnimi velikimi črkami v vozliščih. Tole je primer takšnega drevesa:



(*Opomba:* pri drevesih te vrste velja tudi v primerih, ko ima neko vozlišče enega samega otroka ali poddrevo, da je ta otrok ali levi ali pa desni. V gornjem drevesu je na primer `F` desni otrok vozlišča `E` in če bi bil `F` namesto tega levi otrok, to že ne bi bilo več enako drevo.)

Da bi ta drevesa shranila za kasneje, je za vsako drevo zapisala dva niza. Prvega je zapisala kot: levo poddrevo, koren, desno poddrevo. Drugega pa je zapisala po nivojih (najprej prvi nivo, nato drugi, ..., vsak nivo pa od leve proti desni). Za zgornji primer je dobila `ABCDEF` in `DBEACF`.

Valentina je upala, da bo iz takih parov nizov lahko rekonstruirala svoja drevesa. Ko je čez leta to želela storiti, je ugotovila, da je to možno, vendar le zato, ker je vsako drevo zapisala z dvema nizoma. Bilo pa je to delo precej zamudno. Zato te prosi, da bi ji napisal program, ki bi ji pri tem pomagal.[64]

Program naj prebere oba niza (kot smo ju opisali zgoraj), drevo pa naj izpiše kot niz oblike: levo poddrevo, desno poddrevo, koren. Naš zgornji primer bi torej zapisali `ACBFED`.

REŠITVE NALOG ZA PRVO SKUPINO

N: 340

R1998.1.1

Program izpiše samega sebe.

V konstanti `S` je po en niz za vsako vrstico programa, razen za tiste, ki definirajo sámo konstanto `S`. Program najprej izpiše vrstice do začetka definicije konstante `S`, nato uporabi nize iz tabele `S`, da izpiše še vrstice, ki definirajo `S` (vsak niz mora oviti v narekovaje in dodati še zamik na začetku vrstic), nato pa lahko izpiše še preostanek programa. Tam, kjer so v programu enojni narekovaji (znak `'`), vsebujejo nizi v tabeli `S` dvojne narekovaje (znak `"`), pri izpisovanju pa jih zamenjamo z enojnimi. Izjema je eden, ki mora vendarle ostati (23. znak v `S[18]`).

Program, ki izpiše samega sebe, je običajno sestavljen iz dveh delov, A in B; A definira neko konstanto S, ki vsebuje kodo dela B; B pa izpiše najprej A (pri čemer si lahko pomaga z vsebino konstante S) in nato še samega sebe (tu pa v bistvu preprosto izpiše S).

[64] To je naloga H z lokalnega ACMovega študentskega tekmovanja univerze v Ulmu v programiranju (Ulm, 1997); #536 v zbirki na `online-judge.uva.es`.

Ostane le še vprašanje, kako A in B sestaviti in povezati skupaj v program, ki res spoštuje vsa pravila izbranega programskega jezika in je njegov izpis res natančno enak njegovi izvorni kodi. Ta del problema so v bistvu le tehnikalije, odvisne od sintakse in drugih pravil našega programskega jezika.

V našem primeru je A sestavljen iz vrstic 4 do 24, torej tistih z vsebino tabele `S`, ostalo pa je B. Tehnikalije so v pascalu povezane predvsem z narekovaji. Če B vsebuje kaj enojnih narekovajev, jih mora A vsebovati podvojene (na primer: niz `x'y` moramo v pascalski izvorni kodi zapisati kot `'x''y'`), vendar pa jih bo B v konstanti S potem videl le enojno. Zato mora B, ko izpisuje del A, vsak enojni narekovaj izpisati dvakrat. Možne so še druge rešitve; A bi lahko v konstanti S na mestih, kjer B vsebuje enojne narekovaje, definiral kak drug znak, ki se drugače v kodi ne pojavlja, npr. `"`. B bi moral potem pri izpisovanju dela A vse znake `"` zamenjati z enojnimi narekovaji; seveda pa s tem B-jeva koda že vsebuje en znak `"` (da lahko z njim primerja vsebino konstante `S`) in tega pri izpisovanju A-ja ne sme spremeniti v `'`; v ta namen bi lahko na primer pazil, v kateri vrstici in stolpcu se ta `"` pojavlja in tistega pač ne bi spremenil v `'` (pri programu v nalogi smo se odločili prav za to rešitev). Lahko pa bi B uporabil tudi ASCII kodo znaka `"`, tako da bi namesto znaka `"` vseboval B le nedolžni niz `Chr(34)`. Toda isti prijem bi lahko uporabili že na samem začetku; sploh ni nujno, da se v B-ju pojavlja znak `'`, saj ga lahko nadomestimo s `Chr(39)`, ko bomo izpisovali narekovaje v definiciji konstante `S`; za izpis presledkov pa uporabimo `Chr(32)`, da jih ne bo treba v izvorni kodi dela B navajati v narekovajih kot `' '`. Vendar je prijem s kodami ASCII po svoje manj eleganten, ker mora predpostaviti, da našega programa ne bodo uporabljali na računalniku s kakšnim zelo eksotičnim naborom znakov.

Če ima programski jezik ugodne lastnosti, je pisanje programov, ki izpišejo sami sebe, lahko precej preprostejše. V jeziku C smo lahko s pomočjo funkcije `printf` malo bolj jedrnati:

```
#include <stdio.h>
char*s="#include <stdio.h>%cchar*s=%c%s%c,%c*t=%c%s%c,%c*u=%c%s%c;%c",
*t="int main(){int n='%cn',q='%c%c',b='%c%c';%c  %sreturn 0;}%c",
*u="printf(s,n,q,s,q,n,q,t,q,n,q,u,q,n);printf(t,b,b,q,b,b,n,u,n);";
int main(){int n='\n',q='\"',b='\\';
  printf(s,n,q,s,q,n,q,t,q,n,q,u,q,n);printf(t,b,b,q,b,b,n,u,n);return 0;}
```

Delu A v tem primeru ustrezajo nizi `s`, `t` in `u`, ostalo pa je del B. S pomočjo spremenljivk `n`, `q` in `b` se lahko izognemo potrebi po tem, da bi se v morali v nizih `s`, `t` in `u` pojavljati nadležni znaki, zaradi katerih bi bila vsebina takega niza med tekom programa drugačna od zaporedja znakov v izvorni kodi, s katero je bil niz definiran.

Python pa ima funkcijo `repr(x)`, ki vrne[65] kot niz kar košček izvorne kode, s

[65] Vsaj pri standardnih tipih; razredi, ki jih napiše uporabnik, lahko sami določijo, kaj naj vrne `repr` na primerkih tega razreda.

kakršnim bi bilo treba v programu inicializirati neko spremenljivko, da bi dobila vrednost `x`. Zato so nam prihranjene manipulacije z narekovaji in samega sebe izpiše že naslednji enovrstični program (ki pa ima, kot lahko opazimo, zelo podobno strukturo kot gornji pascalski program):

```
s = 's = %s; print s %% repr(s)'; print s % repr(s)
```

Do skrajnosti gredo v to smer kakšne tradicionalne oblike BASICa:

```
10 LIST
```

Še en razvpit primer pa je popolnoma prazen program (ki, odkrito povedano, res ne izpiše ničesar), ki je dobil leta 1994 na tekmovanju obfuscated C nagrado za najhujšo zlorabo pravil. `:-)`

Na spletu je precej zanimivih strani o programih, ki izpišejo samega sebe (v angleščini se takemu programu pogosto pravi *quine*, v spomin na logika Willarda Quinea). S podobnim razmislekom kot pri takih programih lahko sestavimo tudi na primer dva programa, ki izpišeta drug drugega, ipd.

N: 341

R1998.1.2

(a) Na začetek vrstice najprej vrinimo 80 presledkov in tako poskrbimo, da bo cela vrstica vsebovala vsaj 80 znakov. Če potem obdržimo le zadnjih 80 znakov, bo to celotna vsebina prvotne vrstice in še toliko presledkov na začetku, da bo vrstica desno poravnana.

```
80 VRINI_PRESLEDEK, 70 DESNO, 80 LEVO, BRISI_ZACETEK
```

Nekateri urejevalniki imajo omejitve glede največjega dovoljenega števila znakov v vrstici. Gornja rešitev bi lahko v eni vrstici dobila do 150 znakov (največ 70 od prej in še 80 vrinjenih presledkov). Recimo, da naš urejevalnik ne dovoli toliko znakov in bi pri vrivanju presledkov na začetku vrstice začel rezati znake s konca vrstice, če bi le-ta postala predolga. V tem primeru je bolje vrivati presledke na koncu in jih nato premakniti na začetek. Recimo, da je vrstica na začetku dolga n znakov. Če vrinemo na koncu 80 presledkov, bo rezultat dolg vsaj 80 znakov; in če zdaj zbrišemo vse razen prvih 80 znakov, bomo dobili kar prvotno vrstico, podaljšano za $80 - n$ presledkov. Če zdaj te presledke premaknemo na začetek vrstice, bomo dobili ravno to, kar iščemo: prvotno vrstico, poravnano desno na dolžino 80.

```
70 DESNO, 80 VRINI_PRESLEDEK, 150 LEVO, 80 DESNO,
BRISI_KONEC, NA_KONEC_PREJSNJE_BESEDE, DESNO,
BRISI_KONEC, 70 LEVO, VRINI_ZBRISANO
```

(b) Premaknimo se na konec vrstice, nato pa za tri mesta v levo in vrinimo presledek; nato za štiri mesta v levo (mimo pravkar vrinjenega presledka in še naslednjih treh števk) in spet vrinimo presledek. To je že dovolj, če je

število v tej vrstici res največ dvajset milijonov (saj ima tako le osem števk in ni treba vriniti več kot dveh presledkov); vendar pa smo v primeru, ko je število krajše, zdaj mogoče vrinili presledek ali dva preveč. Zato je pametno iti na začetek vrstice (kar zahteva še največ tri premike v levo) in nato pobrisati vse do začetka prve besede — s tem se znebimo morebitnih odvečnih presledkov. Toda „pobrisati vse do začetka prve besede“ ni čisto trivialno: če pred prvo besedo (v našem primeru: pred prvim sklopom števk) ni nobenega presledka, bi bili na začetku vrstice obenem že tudi na začetku prve besede in ukaz `NA_ZACETEK_NASLEDNJE_BESEDE` bi nas prestavil na začetek *druge* besede. Zato je bolje pred tem vriniti na začetek vrstice še en presledek, se postaviti nanj in nato skočiti na začetek naslednje besede (kar bo zdaj zagotovo res pomenilo na začetek prve besede).

```
8 DESNO,
3 LEVO, VRINI_PRESLEDEK,
4 LEVO, VRINI_PRESLEDEK,
3 LEVO, VRINI_PRESLEDEK, LEVO,
NA_ZACETEK_NASLEDNJE_BESEDE, BRISI_ZACETEK
```

R1998.1.3

Program mora le slediti navodilom naloge. Prenos iz prejšnjega leta hranimo v spremenljivki Prenos (na začetku dobi vrednost 0), nato pa vsako leto prištejemo dobiček tistega leta in tisto, kar sega čez 1000, razglasimo za prenos v prihodnje leto.

N: 343

```
program UstvarjalnoKnjigovodstvo;
var Prenos, Dobicek: integer;
begin
  Prenos := 0;
  while not Eof do begin
    ReadLn(Dobicek);
    Prenos := Prenos + Dobicek;
    if Prenos > 1000 then Dobicek := 1000 else Dobicek := Prenos;
    Prenos := Prenos - Dobicek;
    WriteLn(Dobicek, ' ', Prenos);
  end; {while}
end. {UstvarjalnoKnjigovodstvo}
```

R1998.1.4

V spremenljivki NaZalogi bomo hranili število dežnih kapelj, ki smo jih že odčitali z merilne naprave, vendar zanje še nismo zapiskali (niti se nismo odločili, da jih bomo ignorirali). Če je NaZalogi = 0, lahko funkciji StDeznihKapelj naročimo, naj čaka na naslednjo kapljo, sicer pa ji naročimo, naj se vrne takoj in s tem le preverimo, če je padlo od zadnjega klica še kaj novih kapelj. Nato, če je NaZalogi večja od 0, jo zmanjšamo za 1, zapiskamo in počakamo eno sekundo.

N: 344

Zahtevo, naj kapljice, ki bi morale na svoj pisk čakati več kot pet sekund, tiho ignoriramo, lahko upoštevamo na naslednji način. Recimo, da bi vsako kapljico, za katero na novo izvemo ob trenutnem klicu funkcije StDeznihKapelj, dodali na nek seznam (pravzaprav vrsto), kjer bi čakala, da izvedemo pisk zanjo. Če je kapljica v trenutku, ko jo na seznam dodamo, n-ta po vrsti, bo njen pisk prišel na vrsto čez $n-1$ sekund (za prvo kapljico bomo zapiskali takoj, za drugo čez eno sekundo in tako naprej). In ker smo jo na seznam ravnokar dodali, vemo, da je padla nekje med predzadnjim in zadnjim klicem funkcije StDeznihKapelj, torej pred največ eno sekundo, saj lahko med dvema klicema te funkcije v našem programu mine največ ena sekunda (če smo po prejšnjem klicu zapiskali in nato zaspali za toliko časa). Torej, če je $n \leq 5$, bo prišel njen pisk na vrsto čez največ štiri sekunde, padla pa je pred največ eno sekundo, torej pisk še ne bo prepozen; če pa je $n > 5$, bo prišel pisk na vrsto čez pet ali več sekund, torej bo v vsakem primeru prepozen (tudi pri $n = 6$, saj mora od padca do trenutka, ko bomo mi res izvedli prvi pisk, tudi miniti vsaj neka majhna količina časa, kar bo skupaj s petimi sekundami do šestega piska res pomenilo strogo več kot pet sekund od padca te kapljice do njenega piska). Torej lahko iz seznama vržemo vse kapljice razen prvih petih. No, ker pa naš program ene kapljice prav nič ne razlikuje od druge, zanje ni treba res imeti seznama, ampak je dovolj, če si zapomnimo, koliko bi jih v tem seznamu bilo; ravno to pa nam pove spremenljivka NaZalogi. To moramo torej zmanjšati na 5, če je po klicu StDeznihKapelj zrastla čez 5.

```
program Dez;
var NaZalogi: integer;
begin
  NaZalogi := 0;
  while true do begin
    if NaZalogi > 0
      then NaZalogi := NaZalogi + StDeznihKapelj(0)
      else NaZalogi := NaZalogi + StDeznihKapelj(-1);
    if NaZalogi > 5 then NaZalogi := 5;
    if NaZalogi > 0 then
      begin Zapiskaj; NaZalogi := NaZalogi - 1; Zaspi(1000) end;
  end; {while}
end. {Dez}
```

REŠITVE NALOG ZA DRUGO SKUPINO

N: 344

R1998.2.1

Naj bosta (x, y) in (x', y') dve točki, ki ležita v spodnjem levem in zgornjem desnem oglišču nekega kvadrata s stranico a. Potem mora veljati $x' = x + a$ in $y' = y + a$, torej $x' - y' = (x+a)-(y+a) = x-y$. Taki dve točki torej prepoznamo po tem, da imata obe enako razliko med x- in y-koordinato; z drugimi besedami, če bi skozi vsako

točko poslali premico, ki oklepa z absciso kot 45°, bi bila to obakrat ena in ista premica.

Zato lahko ravnamo takole: pripravimo si tabelo $a[-20\,000..20\,000]$, katere elementi bodo cela števila; na začetku postavimo vse elemente na nič. Z elementom $a[i]$ bomo šteli, koliko točk leži na premici $x - y = i$. Imejmo še nek števec, ki pove, koliko kvadratov smo že našli (recimo mu S). Potem za vsako točko (x, y) izračunamo $x - y$ in vemo, da smo doslej odkrili $a[x - y]$ točk z enako razliko med koordinatama in da naša nova točka tvori po en kvadrat z vsako od njih. (Seveda lahko naša nova točka tvori kvadrate tudi s kakšnimi točkami, ki jih bomo prebrali šele kasneje, ampak tiste kvadrate bomo pač šteli takrat, pri tistih točkah, tako da nam ni treba skrbeti, da bi kakšnega spregledali.) Torej povečajmo S za $a[x - y]$, nato pa povečajmo $a[x - y]$ za 1, da v mislih dodamo pravkar prebrano točko med tiste s takšno razliko koordinat.

Na koncu bo S ravno število iskanih kvadratov. Lepo pri tem postopku je tudi to, da nam ni treba v pomnilniku hraniti koordinat vseh točk. Količina porabljenega časa je $O(n)$ — premo sorazmerna s številom točk (če predpostavimo, da je zaloga vrednosti naših koordinat, torej v našem primeru od 0 do 20 000, vnaprej določena in omejena).

```
const StTock = ...;
var
  x, y: array [1..StTock] of integer;
  a: array [−20000..20000] of integer;
  j, s: integer;
begin
  ... { preberi koordinate }
  for j := −20000 to 20000 do a[j] := 0;
  s := 0;
  for j := 1 to StTock do begin
    i := x[j] − y[j];
    s := s + a[i];
    a[i] := a[i] + 1;
  end; {for}
  WriteLn('Število kvadratov: ', s);
end.
```

Možna različica zgornje rešitve je tudi ta, da ne bi sproti povečevali števca S, ampak bi se na koncu sprehodili po vseh elementih tabele a. Vrednost $a[i]$ nam pove, da smo videli $a[i]$ točk z razliko koordinat $x - y = i$; toliko točk pa tvori $a[i] \cdot (a[i] - 1)/2$ kvadratov. To moramo sešteti po vseh i, pa dobimo skupno število vseh kvadratov. Slabost v primerjavi s prejšnjim postopkom je ta, da se moramo zdaj še enkrat sprehajati po celi tabeli a; po drugi strani pa zdaj prihranimo po eno seštevanje pri vsaki točki (vrstica `s := s + a[i]` v gornjem programu).

Preprostejša, a časovno veliko bolj potratna rešitev pa je, da pogledamo vse pare točk in za vsak par preverimo, ali točki ležita v ogliščih kvadrata (torej: ali je $x' - x = y' - y$). Zdaj moramo imeti koordinate vseh točk v pomnilniku, kar pri gornji rešitvi ni nujno, količina opravljenega dela pa je sorazmerna s kvadratom števila točk (če je točk n, moramo pregledati $n(n-1)/2$ parov točk).

```
const StTock = ...;
var
  x, y: array [1..StTock] of integer;
  i, j, s: integer;
begin
  ... { preberi koordinate }
  s := 0;
  for i := 1 to StTock - 1 do
    for j := i + 1 to StTock do
      if x[j] - x[i] = y[j] - y[i] then
        s := s + 1;
  WriteLn('Število kvadratov: ', s);
end.
```

N: 345

R1998.2.2

Sosedje naj si med seboj podajajo kalkulator (ali pa tablo) z dosedanjo vsoto svojih plač. Prvi naj si skrivaj izbere neko veliko naključno število, ga vtipka v kalkulator (in si ga zapomni) ter mu prišteje svojo plačo. Nato poda kalkulator naprej; vsakdo prišteje svojo plačo trenutni vsoti in poda kalkulator spet naslednjemu sosedu. Ko to naredijo vsi, naj prvi spet odšteje tisto svoje začetno število. Ostala bo ravno vsota njihovih plač, pri tem pa nobeden od njih ni točno vedel, kakšna je vsota plač tistih, ki so vnesli svoje plače že pred njim, saj niso vedeli, kakšno je tisto začetno število, ki si ga je izmislil prvi sosed. No, paziti morajo le še na to, da si lahko kalkulator zapomni zadnjo izvedeno operacijo, tako da naj po prištevanju svoje plače mogoče prištejejo še 0 ali izvedejo kakšno drugo nekoristno operacijo.

Morebitna slabost opisanega postopka je, da se lahko dva soseda zarotita proti nekemu tretjemu in izvesta njegovo plačo. Morata le poskrbeti, da se bosta v vrstnem redu računanja znašla tik pred in tik za njim; potem lahko primerjata vrednost, ki je bila v kalkulatorju, preden ga je tisti tretji sosed dobil, in vrednost, ko je dal kalkulator naprej; razlika med njima je ravno njegova plača. Da bi takšne zarote otežili, bi lahko na primer rekli, naj si vsakdo izmisli več čudnih števil, ki se bodo seštela ravno v njegovo plačo. Potem bi si sosedje kalkulator podajali večkrat, po možnosti vsakič v drugačnem in naključno izbranem vrstnem redu. Vsakdo bi, ko bi dobil kalkulator, prištel naslednje izmed svojih števil; tako bi morali sosedje, če bi hoteli izvedeti njegovo plačo, vedeti za vsak krog posebej, kaj je prištel takrat. Zato za zaroto

zdaj nista dovolj le dva, ampak bi se je morali udeležiti vsi, ki so bili v kakšnem od krogov tik pred ali tik za njim.

R1998.2.3

N: 345

Pomagali si bomo s tehniko „pometanja“ ali preleta ravnine (*plane sweep*). Označimo unijo naših pravokotnikov z U. Spomnimo se, da imajo vsi naši pravokotniki stranice, vzporedne s koordinatnima osema; recimo, da bi ravnino razrezali na vodoravne pasove pri vseh tistih y-koordinatah, kjer ima kateri od naših pravokotnikov svojo zgornjo ali spodnjo stranico. Kot vidimo iz slike na strani 357, velja zdaj za vsakega od nastalih pasov naslednje: če pogledamo katerokoli vodoravno premico znotraj tega pasu, pokriva U na tej premici vedno ene in iste x-koordinate; recimo, da imajo ti intervali pokritih x-koordinat skupno dolžino d, pas, v katerem to opazujemo, pa omejujeta premici $y = c$ in $y = c'$. Iz tega sledi, da ima presek lika U in opazovanega pasu ploščino $d \cdot (c' - c)$. Če bi zdaj podobno naredili z vsemi pasovi in dobljene ploščine sešteli, bi dobili ravno ploščino celega lika U. Zdaj imamo tak postopek:

A1 Naj ima i-ti pravokotnik spodnji levi kot (x_{i1}, y_{i1}) in zgornji desni kot (x_{i2}, y_{i2}). Postavimo $S := 0$; s prištevanjem bo S na koncu postala ploščina celega lika U.

A2 Ravnino bo treba prerezati pri vseh y_{i1} in vseh y_{i2}. Pripravimo si torej nek seznam trojic, ki bo vseboval za vsak i trojici $\langle y_{i1}, i, 1\rangle$ in $\langle y_{i2}, i, 2\rangle$; vse te trojice uredimo po naraščajoči y-koordinati. Drugi dve komponenti nam povesta, od katerega pravokotnika in katerega roba (zgornjega ali spodnjega) je določena trojica prišla; to bo prišlo kasneje še prav. Oštevilčimo dobljene y-koordinate v naraščajočem vrstnem redu kot y_j, $j = 1, \ldots, 2n$.

A3 (Ta korak izvedemo po enkrat za vsak j od 1 do $2n-1$.) Ogledali si bomo presek U in pasu $y_j \le y \le y_{j+1}$. Naj bo T unija intervalov $[x_{i1}, x_{i2}]$ po vseh tistih i, za katere je $y_{i1} \le y_j$ in $y_{j+1} \le y_{i2}$. Naj bo d skupna dolžina te unije intervalov. Povečajmo S za $(y_{j+1} - y_j) \cdot d$.

A4 Na koncu tega postopka je S ravno ploščina celega lika U.

Oglejmo si zdaj podrobneje delo z unijo intervalov v točki A3. Recimo, da bi T vsebovala intervale $[1, 3]$, $[2, 5]$, $[0, 6]$ in $[8, 18]$; dolžina unije teh štirih intervalov je $d = 16$, saj pokrivajo vsega skupaj x-koordinate od 0 do 6 in od 8 do 18. Nekatere podintervale, na primer $[1, 5]$, sicer pokriva po več intervalov iz T, vendar je za skupno dolžino pomembno le, ali je nek podinterval sploh pokrit ali ne, če je pokrit večkratno, pa ga moramo v dolžino unije še vedno šteti le enkrat. Navedeni štirje intervali pravzaprav razrežejo os x na disjunktne podintervale: $[0, 1]$ (ki ga pokriva en sam interval iz T, namreč $[0, 6]$); $[1, 2]$

(ki ga pokrivata dva, namreč $[0,6]$ in $[1,3]$); $[2,3]$ (ki ga pokrivajo trije, poleg prejšnjih dveh še $[2,5]$); $[3,5]$ (ki ga pokrivata dva, $[0,6]$ in $[2,5]$); $[5,6]$ (ki ga pokriva samo $[0,6]$); $[6,8]$ (ki ga ne pokriva nobeden od intervalov iz T); in $[8,18]$ (ki ga pokriva samo $[8,18]$). Če bi torej za dano množico intervalov preučili, kako nam razrežejo os x na disjunktne podintervale, bi morali potem le še sešteti dolžine tistih podintervalov, ki jih pokriva vsaj en interval iz T. Nov disjunktni podinterval se začne pri vsaki taki x-koordinati, kjer ima kakšen od intervalov v T krajišče (ali levo ali pa desno). To bi lahko naredili s postopkom, podobnim tistemu iz točke A2 zgoraj:

B1 Recimo, da imamo v T intervale $[x_{i1}, x_{i2}]$ za nekaj vrednosti i. Pripravimo seznam vseh parov $\langle x_{i1}, 1\rangle$ in $\langle x_{i2}, 2\rangle$ za vse intervale $[x_{i1}, x_{i2}]$ v T. Uredimo te pare po naraščajoči x-koordinati. Recimo, da tako urejene oštevilčimo kot $\langle x_k, t_k\rangle$ za $k = 1, \ldots, 2m$ (če je m število intervalov v T). Postavimo $d := 0$, $c := 0$. V d bomo zbirali skupno dolžino unije intervalov, v c pa bomo za trenutni podinterval vzdrževali podatek, koliko intervalov iz T ga pokriva.

B2 Ponovimo koraka B3 in B4 za vsak k od 1 do $2m-1$:

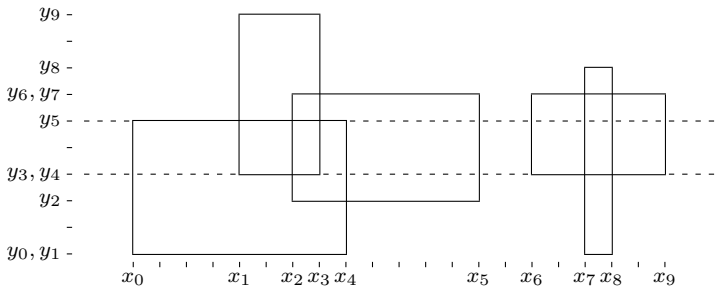
B3 Zdaj gledamo interval od x_k do x_{k+1}. Če je $t_k = 1$, je x_k levo krajišče nekega intervala in moramo c povečati za 1, ker podinterval $[x_k, x_{k+1}]$ pokriva en T-jev interval več kot podinterval $[x_{k-1}, x_k]$ pred njim. Če pa je $t_k = 2$, je x_k desno krajišče nekega intervala in moramo c zmanjšati za 1. ter jih vse skupaj uredili po naraščajoči x-koordinati.

B4 Če je zdaj $c > 0$, je podinterval $[x_k, x_{k+1}]$ pokrit z vsaj enim od intervalov iz T, zato povečajmo d za $x_{k+1} - x_k$. Če pa je $c = 0$, pustimo d pri miru.

B5 Na koncu tega postopka je v d ravno dolžina unije vseh intervalov iz T.

Pokazati je mogoče, da zahteva urejanje m elementov (če uporabimo dovolj učinkovit algoritem) v najslabšem primeru čas $O(m \log m)$. Če bi torej uporabili ta postopek v točki A3 zgornjega algoritma, bi potrebovali za tisto točko v najslabšem primeru $O(n \log n)$ časa (čas, potreben za to, da ugotovimo, kateri pravokotniki sploh segajo v trenutni pas, je le $O(n)$ in bi se izgubil v času urejanja v točki B1), izvesti pa jo moramo $O(n)$-krat, tako da bi celoten postopek lahko pri n pravokotnikih porabil do $O(n^2 \log n)$ časa. Na srečo pa lahko to še precej izboljšamo.

Opazimo namreč lahko, da sta si množici intervalov pokritih x-koordinat za dva sosednja pasova zelo podobni. Ker smo rezali na pasove pri vsaki višini, kjer ima kateri od pravokotnikov svoj zgornji ali pa spodnji rob, se dva sosednja pasova glede pokrivanja x-koordinat razlikujeta le na enega od



Primer naloge s petimi pravokotniki. Oznake $x_0, \ldots, x_9$ predstavljajo x-koordinate njihovih levih in desnih stranic v naraščajočem vrstnem redu, podobno pa $y_0, \ldots, y_9$ za y-koordinate njihovih spodnjih in zgornjih stranic.

Če se omejimo na pas ravnine med premicama $y = y_4$ in $y = y_5$, vidimo, da pripadajo našim pravokotnikom tu naslednji intervali x-koordinat: $[x_0, x_4]$, $[x_1, x_3]$, $[x_2, x_5]$, $[x_6, x_9]$ in $[x_7, x_8]$, torej po eden za vsak pravokotnik, ki sega v ta pas. Seveda se ti intervali prekrivajo in je njihova unija preprosto $[x_0, x_5] \cup [x_6, x_9]$. Skupna dolžina te unije je $13 + 5 = 18$ enot, širina pasu pa je 2 enoti, tako da ta pas k površini unije opazovanih pravokotnikov prispeva 36 enot.

Podobno bi za pas od y_1 do y_2 ugotovili, da prispeva $9 \times 2 = 18$ enot, pas od y_2 do y_3 prispeva $14 \times 1 = 14$ enot, pas od y_5 do y_6 prispeva 14 enot, pas od y_7 do y_8 štiri enote in pas od y_8 do y_9 še šest enot. Tako vidimo, da je ploščina unije teh petih pravokotnikov enaka $18 + 14 + 36 + 14 + 4 + 6 = 92$ enot.

---

naslednjih dveh načinov: če je med njima spodnji rob i-tega pravokotnika, je pokrit v zgornjem od opazovanih dveh pasov tudi interval $[x_{i1}, x_{i2}]$, ki v spodnjem mogoče še ni bil; in če je med njima zgornji rob i-tega pravokotnika, interval $[x_{i1}, x_{i2}]$ v zgornjem pasu ni več pokrit, v spodnjem pa je še bil (no, lahko je tudi v zgornjem pasu še vedno pokrit, deloma ali pa celo v celoti, če ga pokrivajo kakšni drugi intervali). Torej, če bi imeli množico pokritih intervalov T predstavljeno na primeren način, bi jo morali ob prehodu z enega pasu na naslednjega le še malo popraviti, ne bi pa je bilo treba v celoti sestavljati na novo.

Drugo koristno opažanje je, da so krajišča intervalov, s katerimi imamo v množici T opraviti, vedno take x-koordinate, pri katerih ima eden od naših n pravokotnikov svoj levi ali desni rob; vseh možnih krajišč je torej največ $2n$, čeprav se v posameznem pasu pri marsikaterem od njih mogoče ne zgodi nič zanimivega, če tistega pravokotnika v tem pasu pač ni. Recimo, da imamo v našem primeru opravka z $n = 5$ pravokotniki in jim po x-koordinatah ustrezajo intervali $[1, 3]$, $[2, 5]$, $[0, 6]$, $[8, 18]$ in $[15, 20]$. Najfinejše razbitje na podintervale, s katerim bi utegnili imeti pri množici T opravka, je torej razbitje na $\langle 0, 1, 2, 3, 5, 6, 8, 15, 18, 20 \rangle$. Tu imamo vseh $2n = 10$ krajišč, ki nam opisujejo skupno devet podintervalov. Vse, kar moramo v danem trenutku vedeti o množici T, da lahko izračunamo dolžino unije pokritih intervalov, je podatek

o tem, katere od teh 9 podintervalov pokriva vsaj eden od intervalov, ki so trenutno v T. Ko dodamo v T nek nov interval, se pokritost nekaterih podintervalov poveča za 1, ko pa iz T nek interval zbrišemo, se pokritost nekaterih zmanjša za 1. Algoritma A in B lahko torej zamenjamo s takšnim postopkom:

C1 Naj ima i-ti pravokotnik spodnji levi kot (x_{i1}, y_{i1}) in zgornji desni kot (x_{i2}, y_{i2}). Postavimo $S := 0$; s prištevanjem bo S na koncu postala ploščina celega lika U.

C2 Pripravimo si seznam trojic $\langle y_{i1}, i, 1\rangle$ in $\langle y_{i2}, i, 2\rangle$ za vse i in jih uredimo po naraščajoči y-koordinati; oštevilčimo dobljene y-koordinate v naraščajočem vrstnem redu kot y_j, $j = 0, \ldots, 2n-1$.

C3 Pripravimo si seznam trojic $\langle x_{i1}, i, 1\rangle$ in $\langle x_{i2}, i, 2\rangle$ za vse i in jih uredimo po naraščajoči x-koordinati; oštevilčimo dobljene x-koordinate v naraščajočem vrstnem redu kot x_j, $j = 0, \ldots, 2n-1$. Za vsak pravokotnik si lahko zdaj tudi zapišemo, kateri v tem vrstnem redu sta x-koordinati, ki predstavljata njegov levi in desni rob; naj bo torej u_{i1} položaj koordinate x_{i1} v tem vrstnem redu, u_{i2} pa položaj koordinate x_{i2}.

C4 T naj bo množica trenutno pokritih intervalov; na začetku je prazna.

C5 (Ta korak izvedemo po enkrat za vsak j od 0 do $2n-1$.) Če je y_j spodnja stranica i-tega pravokotnika, dodaj $[x_{u_{i1}}, x_{u_{i2}}]$ v T; sicer pa je y_j zgornja stranica i-tega pravokotnika in zbriši $[x_{u_{i1}}, x_{u_{i2}}]$ iz T. Naj bo zdaj d skupna dolžina unije intervalov v T. Povečajmo S za $(y_{j+1} - y_j) \cdot d$.

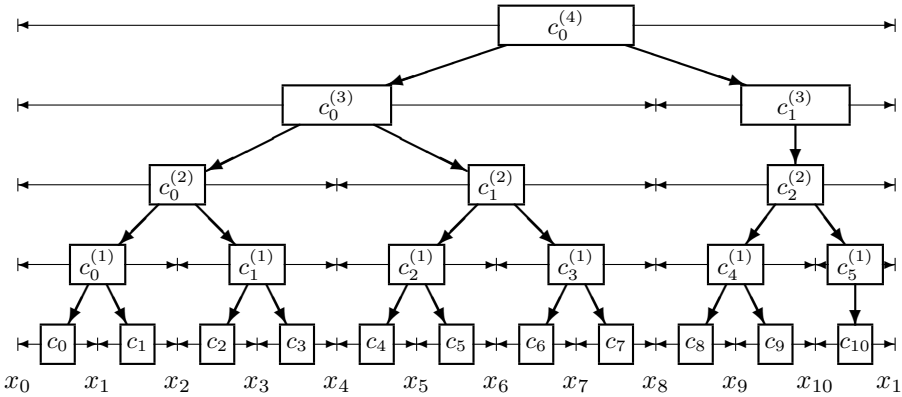
C6 Na koncu tega postopka je S ravno ploščina celega lika U.

Preprosta oblika podatkovne strukture, s katero bi lahko predstavili množico T, bi bila tabela $2n-1$ števil, v kateri bi j-to število (recimo mu c_j) povedalo, kolikokrat je pokrit interval $[x_j, x_{j+1}]$. Operacije na njej bi izvajali takole:

Inicializacija podatkovne strukture T:
    for $j := 1$ **to** $2n-1$ **do** $c_j := 0$;
    $d := 0$
Dodajanje intervala $[x_u, x_v]$ v T:
    for $j := u$ **to** $v-1$ **do**
        $c_j := c_j + 1$;
        if $c_j = 1$ **then** $d := d + (x_{j+1} - x_j)$
Brisanje intervala $[x_u, x_v]$ iz T:
    for $j := u$ **to** $v-1$ **do**
        $c_j := c_j - 1$;
        if $c_j = 0$ **then** $d := d - (x_{j+1} - x_j)$

Lahko se zgodi, da je interval, ki ga dodajamo, zelo širok in moramo zato spremeniti veliko elementov tabele c. V najslabšem primeru bo dodajanje ali brisanje intervala trajalo $O(n)$ časa. Ker moramo po eno tako operacijo izvesti v vsaki izvedbi koraka C5, ta pa se izvede $O(n)$-krat, je skupna zahtevnost algoritma $O(n^2)$ (v primerjavi s tem je čas $O(n \log n)$ za urejanje v točkah C2 in C3 nepomemben). To je že malo bolje od prejšnjega algoritma, zares dobro pa še vedno ni.

Še nadaljnjo pomembno izboljšavo pa dobimo, če nad tabelo c postavimo še drevesasto strukturo z več manjšimi tabelami. Vpeljimo tabelo $c^{(1)}$ z n elementi; vsak od njih bo predstavljal pokritost dveh sosednjih podintervalov: element $c_j^{(1)}$ se bo nanašal na podinterval $[x_{2j}, x_{2(j+1)}]$. (Če n ni večkratnik števila 2, si mislimo, da zadnji element tabele $c^{(1)}$ predstavlja le zadnji podinterval, ne pa dveh skupaj. Podobno ravnamo tudi pri kasnejših tabelah.) Podobno vpeljimo $c^{(2)}$ z $\lceil n/2 \rceil$ elementi, pri čemer $c_j^{(2)}$ predstavlja podinterval $[x_{4j}, x_{4(j+1)}]$. Tako nadaljujemo do tabele $c^{(K)}$ za $K = \lceil \lg 2n \rceil$; ta ima en sam element, ki predstavlja celoten interval $[x_0, x_{2n-1}]$. Takšni drevesasti strukturi pravijo „drevo segmentov“ (*segment tree*).



Primer drevesa segmentov za $n = 6$ intervalov s skupno 12 krajišči $x_0, \ldots, x_{11}$, ki določajo 11 osnovnih podintervalov. Vodoravne puščice pri vsakem vozlišču drevesa kažejo, kateri interval x-koordinat predstavlja to vozlišče.

Namen te podatkovne strukture je, da v primerih, ko moramo dodati ali brisati nek interval $[x_u, x_v]$ in je v veliko večji od u, ne bo treba spreminjati elementov c_j za vsak posamezen j od u do $v - 1$, ampak bo dovolj že, če bomo spremenili peščico primerno izbranih elementov v višje ležečih tabelah, saj vsak od njih zaleže za več elementov prvotne tabele c_j. Števec $c_j^{(k)}$ tako predstavlja interval $[x_{2^k \cdot j}, x_{2^k \cdot (j+1)}]$. Doslej nam je vrednost d predstavljala skupno dolžino unije vseh pokritih intervalov; zdaj pa bomo pri vsakem vozlišču drevesa hranili ločeno vrednost $d_j^{(k)}$, ki pomeni skupno dolžino unije vseh

pokritih intervalov v poddrevesu, ki se začenja pri tem vozlišču, če se delamo, kot da ni nobeden od prednikov tega vozlišča pokrit že sam po sebi. Tisto, kar smo doslej imenovali d, je tako pravzaprav $d_0^{(K)}$.

Dodajanje intervala $[x_u, x_v]$ *v* $c_j^{(k)}$*:*

1 če je $u \le 2^k \cdot j$ in $2^k \cdot (j+1) \le v$:
2 povečaj $c_j^{(k)}$ za 1
3 sicer:
4 če je $u < 2^k \cdot (j+1/2)$: dodaj $[x_u, x_v]$ v $c_{2j}^{(k-1)}$.
5 če je $v \ge 2^k \cdot (j+1/2)$: dodaj $[x_u, x_v]$ v $c_{2j+1}^{(k-1)}$.
6 popravi $d_j^{(k)}$:
7 če je $c_j^{(k)} > 1$, naj bo $d_j^{(k)} \leftarrow x_{2^k \cdot (j+1)} - x_{2^k \cdot j}$ (interval pokrit v celoti)
8 sicer, če je $k = 0$, naj bo $d_j^{(k)} \leftarrow 0$ (nepokrit list)
9 sicer naj bo $d_j^{(k)} \leftarrow d_{2j}^{(k-1)} + d_{2j+1}^{(k-1)}$ (seštejmo pokritost podintervalov)

Mi kot uporabniki drevesa moramo ta rekurzivni postopek za dodajanje pognati pri korenu drevesa, torej pri $c_0^{(K)}$. Rekurzija se izteče, ko pade k na 0; element $c_j^{(0)}$ je pravzaprav kar c_j. Pogoj v vrstici 1 je v tem primeru zagotovo izpolnjen, saj smo pri razbitju na osnovne podintervale, ki jih predstavljajo elementi c_j, že upoštevali leve in desne stranice vseh pravokotnikov, torej vrednosti x_u in x_v, s katerima imamo zdajle opravka, ne moreta razbiti osnovnega podintervala $[x_j, x_{j+1}]$, ki ga predstavlja števec c_j; če je do klica sploh prišlo, to že zanesljivo pomeni, da je ta interval v celoti vsebovan v $[x_u, x_v]$. V vrsticah 4 in 5 izvedemo rekurzivna klica istega podprograma. Po spremembah, ki se zgodijo v vrsticah 1–5, pa moramo v vrsticah 6–9 še popraviti vrednost $d_j^{(k)}$.

Brisanje je povsem podobno, le da moramo v vrstici 2 vrednost $c_j^{(k)}$ zmanjšati za 1 namesto povečati za 1. Inicializacija podatkovne strukture je preprosto v tem, da pripravimo vse tabele in postavimo njihove elemente (c-je in d-je) na 0.

Koliko dela imamo pri dodajanju nekega novega intervala $[x_u, x_v]$ v T? Opazimo lahko, da ob vsakem klicu podprograma za dodajanje $[x_u, x_v]$ v $c_j^{(k)}$ (ki predstavlja interval $[x_L, x_D]$ za $L = 2^k \cdot j$, $D = 2^k \cdot (j+1)$) velja $x_u < x_D$ in $x_v > x_L$. Zdaj lahko ločimo klice štirih vrst glede na to, ali $[x_u, x_v]$ vsebuje x_L, x_D, nobeno ali obe: (*A*) $x_u \le x_L < x_v < x_D$; (*B*) $x_L < x_u < x_D \le x_v$; (*C*) $x_L < x_u < x_v < x_D$; (*D*) $x_u \le x_L < x_D \le x_v$. Če premislimo, kakšni rekurzivni klici utegnejo nastati iz klica posamezne vrste, jih lahko zapišemo takole: $A \to A|DA|D$, $B \to B|BD|D$, $C \to D|C|BA|BD|DA$, $D \to \varepsilon$. S tem mislimo, da lahko iz klica tipa A nastane ali en klic tipa A ali pa en klic tipa D ali pa en klic tipa D in en klic tipa A, ipd. Iz klica tipa D ne nastane noben rekurzivni klic, kar smo ponazorili z ε. Če imamo zdaj seznam klicev, ki se izvedejo na nivoju k, lahko do klicev za en nivo nižje ($k-1$) pridemo tako, da

vsako črko v seznamu zamenjamo z desno stranjo enega od gornjih pravil. Na primer, če so se na nekem nivoju izvedli klici ADB, se lahko na naslednjem izvedejo $ADDB$ ali pa DDB ali pa AB ipd. Iz oblike pravil vidimo, da (ne glede na to, kakšnega tipa je bil začetni klic za dodajanje v koren) na nobenem nivoju ne bo več kot en klic tipa A, en tipa B in en tipa C; klici tipa D pa lahko nastanejo le iz klicev tipov A, B in C s prejšnjega nivoja in torej tudi ne morejo biti več kot trije. Zato na nobenem nivoju prav gotovo ne more biti več kot šest klicev. Vsak klic pa sam po sebi zahteva konstantno veliko dela, tako da je vsega skupaj le $O(K) = O(\lg n)$ dela. Vsaka od $O(n)$ iteracij točke C5 porabi torej $O(\lg n)$ časa, inicializacija vseh tabel $c^{(k)}$ le $O(n)$, urejanje v točkah C2 in C3 pa še vedno $O(n \lg n)$ kot doslej. Celoten postopek torej porabi $O(n \lg n)$ časa. Pokazati se da, da asimptotično boljšega algoritma ni.[66]

R1998.2.4

N: 346

Naloga zahteva, naj izpisujemo odkrite naslove v enakem vrstnem redu, v kakršnem smo prebirali imena računalnikov iz vhodne datoteke. To pomeni, da če procesor, ki je iskal naslov $(n+1)$-vega računalnika iz vhodne datoteke, konča svoje delo prej kot tisti, ki je iskal naslov n-tega, bomo morali čakati še na tega slednjega, preden bomo lahko izpisali naslova obeh računalnikov. Seveda je neugodno, če bi moral procesor, ki je hitro našel naslov $(n+1)$-vega, zdaj čakati brez dela, da bo nek drug procesor prej našel še naslov n-tega; bolje je, če lahko procesorju, ki je našel naslov $(n+1)$-vega računalnika, takoj dodelimo v iskanje neko novo ime, naslov, ki ga je našel, pa si nekje zapomnimo. Pri tem moramo upoštevati, da imamo za te naslove na voljo omejeno mnogo pomnilnika; če na primer procesor, ki išče naslov n-tega računalnika, zavlačuje toliko časa, da smo medtem že našli naslove za tisoč naslednjih računalnikov, nam ne preostane drugega, kot da ostane nekaj procesorjev brez dela, medtem ko čakamo še na naslov n-tega računalnika, saj več kot toliko že najdenih naslovov ne moremo hraniti v pomnilniku.

Spodnji program lahko hrani največ ShranjenNaslovM naslovov računalnikov. Zanje uporablja tabelo ShranjenNaslov, ki deluje kot krožni medpomnilnik; naslovi so v njej shranjeni v takšnem vrstnem redu, v kakršnem smo prebrali imena računalnikov iz vhodne datoteke, veljavni elementi te tabele pa so na indeksih od IndNajnizjeZapSt do pred (ZapSt **mod** ShranjenNaslovM) + 1. Za vsako ime, ki ga damo v iskanje kakšnemu procesorju, si zapomnimo, kam v tej tabeli bo treba na koncu shraniti njegov naslov. Ko izvemo naslov za indeks IndNaj-

[66] Več o tovrstnih algoritmih se najde v kakšni knjigi o računski geometriji. Na primer: Franco P. Preparata, Michael Ian Shamos: *Computational Geometry: An Introduction*, 2nd ed., Springer, 1988, kjer govori o našem problemu razdelek 8.4, o drevesu segmentov pa razdelek 1.2.3.1. Še en primer naloge, ki zahteva unijo pravokotnikov, je naloga A ("Atlantis") z ACMovega študentskega tekmovanja v programiranju za "Mid-Central Europe" (Freiburg, 19. nov. 2000); opis naloge s testnimi primeri je npr. na `http://www.acm.inf.ethz.ch/ProblemSetArchive/B_EU_MCRC/2000/`.

nizjeZapSt, ga lahko izpišemo, mogoče pa tudi še nekaj naslednjih naslovov, če smo te našli že prej.

```
program Naslovi(Input, Output);
const
  StProcesorjev = 64;
  ShranjenNaslovM = 900; { dolžina tabele ShranjenNaslov }
type
  Niz = packed array [1..64] of char;
var
  ImeRac: Niz;
  { tabela urejenih, še neizpisanih naslovov }
  ShranjenNaslov: array [1..ShranjenNaslovM] of Niz;
  { zaporedna številka, ki je v obdelavi na posameznem procesorju }
  PripadajocaZapSt: array [1..StProcesorjev] of integer;
  { indeks, kjer je shranjen naslov s trenutno najnižjo zaporedno številko }
  IndNajnizjeZapSt: integer;
  { največja zap. številka, ki jo je možno trenutno hraniti v tabeli ShranjenNaslov }
  NajvecjaZapSt: integer;
  ZapSt: integer;              { zaporedna številka prebranega podatka }
  vObdelavi: integer;          { število imen, ki so v paralelni obdelavi }
  j: integer;

  function Obdelaj(Ime: Niz): integer; external;
  function PoberiRezultat(var Naslov: Niz): integer; external;

procedure IzpisiKarMores;
{ Vzame en rezultat iz obdelave, ga uvrsti na pravo mesto v tabelo ShranjenNaslov
  in izpiše, kolikor se je nabralo urejenega dela rezultatov v tej tabeli. }
var
  j, Kam: integer;
  Naslov: Niz;
begin
  j := PoberiRezultat(Naslov); vObdelavi := vObdelavi - 1;
  Kam := (PripadajocaZapSt[j] - 1) mod ShranjenNaslovM + 1;
  ShranjenNaslov[Kam] := Naslov;
  while ShranjenNaslov[IndNajnizjeZapSt] <> '' do begin
    WriteLn(ShranjenNaslov[IndNajnizjeZapSt]);
    ShranjenNaslov[IndNajnizjeZapSt] := '';
    IndNajnizjeZapSt := IndNajnizjeZapSt mod ShranjenNaslovM + 1;
    NajvecjaZapSt := NajvecjaZapSt + 1;
  end; {while}
end; {IzpisiKarMores}

begin {Naslovi}
  ZapSt := 0; vObdelavi := 0;
  IndNajnizjeZapSt := 1; NajvecjaZapSt := ShranjenNaslovM;
```

```
  for j := 1 to ShranjenNaslovM do ShranjenNaslov[j] := ' ';
  while not Eof do begin
    ReadLn(ImeRac); ZapSt := ZapSt + 1;
    while (vObdelavi >= StProcesorjev) or (ZapSt > NajvecjaZapSt) do
      IzpisiKarMores;
    j := Obdelaj(ImeRac);
    PripadajocaZapSt[j] := ZapSt; vObdelavi := vObdelavi + 1;
  end; {while}
  while vObdelavi > 0 do IzpisiKarMores;
end. {Naslovi}
```

REŠITVE NALOG ZA TRETJO SKUPINO

R1998.3.1

N: 346

Funkcija za iskanje najkrajših poti, ki smo jo dobili podano, ne upošteva odloka o tem, kako je dovoljeno v posameznem križišču zavijati. Z drugimi besedami, če se nekaj ulic stika v določenem križišču, bo podana funkcija zadovoljna že s potmi, ki poljubno zavijajo z ene ulice na drugo. Če jo hočemo uporabiti za iskanje poti v skladu z novim odlokom, moramo torej graf predelati tako, da v križiščih sploh ne bo tistih ulic, na katere ne smemo zavijati. Toda v obstoječem grafu je to, na katere ulice je v določenem križišču dovoljeno zaviti, odvisno od tega, iz katere smeri smo v križišče prišli. Torej je zdaj pametno iz vsakega križišča, v katerem se stika k ulic, narediti k ločenih križišč, v katerem bo v vsakega vodila po ena ulica, iz nje pa le tiste, na katere smemo zaviti, če pridemo v to križišče po tej ulici.

Naj bo V množica križišč prvotnega grafa, E pa množica ulic. Naj bo $N(u)$ množica križišč, s katerimi je prek svojih ulic neposredno povezano križišče u. Množico križišč novega grafa definirajmo takole:

$$V' := \{u_v : u \in V, v \in N(u)\}.$$

Novo križišče u_v ponazarja križišče u, v katerega smo se pripeljali iz smeri v. Nasledniki tega križišča naj bodo w_u za vse tiste $w \in N(u)$, za katere zavoj iz smeri $v \to u$ v smer $u \to w$ ni zavoj v levo. Pomembno je, da je novi graf usmerjen; z drugimi besedami, po vsaki povezavi gremo lahko le v eno smer.[67] Dolžina povezave med u_v in w_u naj bo enaka kot dolžina povezave med u in w v prvotnem grafu.

Zdaj lahko poiščemo najkrajšo pot od u do v v prvotnem grafu tako, da poiščemo v novem grafu najkrajše poti od vseh u_w do vseh v_x za vse $w \in N(u)$, $x \in N(v)$ ter vzamemo najkrajšo od vseh teh poti (tu smo seveda predpostavili,

[67] Povezava od u_v do w_u namreč pomeni, da lahko v prvotnem grafu, če pridemo v u iz v, nadaljujemo pot v w. Če pa bi šli po povezavi v nasprotni smeri, od w_u do u_v, bi se čisto neupravičeno delali, kot da lahko v prvotnem grafu, če v w pridemo iz u, nadaljujemo pot nazaj v u in smo nato na istem, kot če bi v u prišli iz v.

da je vseeno, v kakšni smeri začnemo voziti od križišča u in iz katere smeri pridemo na koncu v križišče v).

N: 347

R1998.3.2

Edine količine, ki se pri izračunu šifre lahko spreminjajo, so vrednosti programskega števca in obeh registrov. To pomeni, da je program za izračun odgovora v vsakem trenutku v enem od 2^{32} stanj. Takoj, ko se eno od stanj ponovi, lahko zaključimo, da se je program ujel v neskončno zanko. Oziroma: če program uspešno izračuna odgovor, ga izračuna prej kot v 2^{32} korakih.

Procesor Merlin izvrši 2^{32} ukazov v dobrih 7,52 s. To pomeni, da zadostuje, če po vsakem vprašanju počakamo 8 s. Če po tem času ne dobimo odgovora, se je program pri danem vprašanju ujel v neskončno zanko. To moramo ponoviti za vsa možna vprašanja, kar pomeni, da lahko opravimo test v slabih šestih dnevih.

N: 347

R1998.3.3

Oglejmo si najprej, kako ročno preiskati tabelo iz naloge. Po tabeli se bomo pomikali z dvema indeksoma, „levi“ bo šel od leve proti desni, „desni“ mu bo prihajal nasproti z druge strani. V začetku kaže levi na 2, desni na 95. Vsota je 97, kar je premalo, zato jo povečamo tako, da pomaknemo levi indeks za eno mesto naprej. $7 + 95$ je že preveč — vsota se zmanjša, če pomaknemo za eno mesto desni indeks. $7 + 90$ je spet premalo, zato premaknemo levi indeks in pridemo do prve rešitve, $10 + 90$. Nadaljujemo tako, da premaknemo oba indeksa. $14 + 75$ je premalo, prestavimo levega. $24 + 75$ je (za las) premalo in ko spet premaknemo levi indeks, dobimo $31 + 75$. Ker je to več kot 100, prestavimo desnega in dobimo drugo rešitev, $31 + 69$. Igro nadaljujemo, dokler se indeksa ne srečata.

Koliko seštevanj potrebujemo pri takem postopku reševanja? Razdalja med indeksoma je v začetku $n-1$, v vsakem koraku se zmanjša za 1, kadar odkrijemo par, pa celo za 2. Seštevanj je natančno $n - 1 -$ (*število odkritih parov*).

```
type Tabela = array [1..n] of integer;

procedure IzpisiPare(a: Tabela);
var Levi, Desni: integer;
    Vsota: real;
begin
  Levi := 1; Desni := n;
  while Levi < Desni do begin
    Vsota := a[Levi] + a[Desni];
    if Vsota = 100 then WriteLn(a[Levi], ' ', a[Desni]);
    if Vsota <= 100 then Levi := Levi + 1;
    if Vsota >= 100 then Desni := Desni - 1
  end; {while}
end; {IzpisiPare}
```

Lahko se tudi čisto natančno prepričamo, da naš postopek res ne spregleda nobenega para. Na začetku vsake ponovitve zanke **while** namreč velja naslednji pogoj (takemu pogoju pravimo običajno *zančna invarianta*): program je izpisal od parov z vsoto 100 že vse tiste (a_i, a_j), ki imajo $i < l$ ali pa $j > d$. (Pisali bomo l namesto Levi in d namesto Desni.) Res, na začetku ta pogoj velja, saj je $l = 1$, $d = n$ in takih parov sploh ni. Recimo, da velja na začetku neke opazovane ponovitve glavne zanke. Če je $a_l + a_d \leq 100$, se bo l povečal za 1 in omenjeni pogoj bo po novem pokrival tudi pare z $i = l$, $j \leq d$, ki jih prej ni; toda ti pari, razen za $j = d$, imajo vsoto gotovo manjšo od $a_l + a_d$ (saj je $j < d$, v tabeli pa so sama različna števila) in zato tudi manjšo od 100; par $i = l$, $j = d$ pa smo tako ali tako posebej pregledali in ga izpisali, če je imel vsoto enako 100. Nato se bo, če je $a_l + a_d \geq 100$, d zmanjšal za 1 in omenjeni pogoj bo po novem pokrival tudi pare z $j = d$, $i \geq l$, ki jih doslej ni; toda ti pari, razen za $i = l$ (ki ga pregledamo posebej), imajo vsoto večjo od $a_l + a_d$ in zato tudi večjo od 100. Vidimo torej, da s povečevanjem l-ja in zmanjševanjem d-ja gotovo ne bomo spregledali nobenega para z vsoto 100.

R1998.3.4

Reševanja se lahko lotimo rekurzivno. Preberimo prvo črko iz drugega niza (zapisa vozlišč po nivojih); ta mora predstavljati koren, saj je to edino vozlišče na prvem nivoju. Poiščimo to črko v prvem nizu (zapisu vozlišč po načelu „levo poddrevo, koren, desno poddrevo“); vse, kar je levo od nje, se mora torej nanašati na levo poddrevo, vse, kar je desno od nje, pa na desno poddrevo. Zdaj torej vemo, katere črke so v levem poddrevesu; koren levega poddrevesa je potem tista, ki se prva med njimi pojavlja v zapisu po nivojih. Podobno lahko izvemo za koren desnega poddrevesa in ko imamo enkrat koren nekega poddrevesa, že tudi vemo, kaj je v njegovem levem pod-poddrevesu, kaj pa v desnem; itd. Izhodni niz želene oblike bomo dobili, če po obeh rekurzivnih klicih (za levo in desno poddrevo) izpišemo še črko iz korena.

N: 348

Mimogrede, sprehodu po drevesu, v katerem najprej obiščemo levo poddrevo, nato koren, nazadnje pa še desno poddrevo, pravijo v angleščini *in-order traversal*; če koren obiščemo najprej, je to *pre-order*, če nazadnje, pa *post-order traversal*. V slovenščini jim včasih pravijo *vmesni*, *premi* in *obratni* obhod.

```
function RekonstruirajDrevo(VmesniObhod, NivojskiObhod: string): string;
var ObratniObhod: string;

  procedure Poddrevo(Prvi, Zadnji: integer);
  var C, Koren: char; iKorena, jKorena, i, j: integer;
  begin
    { Pregledati moramo poddrevo z vozlišči VmesniObhod[i]
      za i = Prvi, Prvi + 1, ..., Zadnji − 1, Zadnji. }
    if Zadnji < Prvi then exit; { Prazno poddrevo. }
```

```
    for i := Prvi to Zadnji do begin
      C := VmesniObhod[i];
      { Kje v nivojskem obhodu se pojavlja trenutno vozlišče C? }
      j := 1; while NivojskiObhod[j] <> C do j := j + 1;
      { Zapomniti si hočemo tisto vozlišče trenutnega poddrevesa, ki se v nivojskem
        obhodu pojavlja najbolj zgodaj — to namreč pomeni, da leži najvišje v
        drevesu, torej je to koren našega poddrevesa. }
      if (i = Prvi) or (j < jKorena) then
        begin Koren := C; iKorena := i; jKorena := j end;
    end; {for}
    { Zdaj imamo koren in torej tudi vemo, katera vozlišča so v levem
      in katera v desnem poddrevesu. Njuna opisa bomo z rekurzivnima
      klicema dodali v niz ObratniObhod, na koncu pa mu bomo pritaknili
      še črko iz korena trenutnega drevesa. }
    Poddrevo(Prvi, iKorena - 1);
    Poddrevo(iKorena + 1, Zadnji);
    ObratniObhod := ObratniObhod + Koren;
  end; {Poddrevo};

begin {RekonstruirajDrevo}
  ObratniObhod := ''; Poddrevo(1, Length(VmesniObhod));
  RekonstruirajDrevo := ObratniObhod;
end; {RekonstruirajDrevo}
```

23. državno tekmovanje v znanju računalništva (1999)

NALOGE ZA PRVO SKUPINO

1999.1.1

R: 376

Podjetje Import Eskort te je najelo za svetovalca za rešitev njihovega problema letnice 2000. V svojih strojno berljivih bazah imajo zapisane datume transakcij tako, da je znakovni zapis za koledarsko leto dolg dva zloga (byta). Letnica 1987 je na primer predstavljena z znakoma `87`.

Predlagaj postopek, kako lahko veljavnost zapisa datuma podaljšaš vsaj do leta 2030, ali pa **utemelji**, zakaj je njihov problem žal nerešljiv. Pri tem ne smeš porabiti nič več prostora. Naj vas spomnimo, da gre v en zlog (byte) število med 0 in 255 (vključno s tema dvema).

1999.1.2

R: 376

Iz vesolja pričakuješ signal nezemeljske civilizacije. Utemeljeno lahko pričakuješ, da bo sporočilo poslano v jeziku, ki uporablja angleško abecedo 26 malih črk in zadošča naslednjim pravilom:

- Sporočilo je sestavljeno zgolj iz zaporedja znakov `a`, `b`, `c`, `d`, ..., `z`, ki sestavljajo besede, in iz presledkov med njimi.
- Zaporedne črke se vselej razlikujejo (ni podvojenih črk).
- Samoglasniki `a`, `e`, `i`, `o`, `u` se ne smejo stikati.
- Znak `x` je lahko le na koncu besede.

Napiši podprogram Pravo, ki ugotovi, ali sporočilo Sporocilo ustreza tem pogojem.

Klic funkcije naj bo takle:

```
function Pravo(var Sporocilo: array [1..1000000] of char): boolean;
```

1999.1.3

R: 377

V besedilu iščemo vrstice, ki vsebujejo vsaj eno zvezdico. Izpišemo vsako vrstico z zvezdico, poleg nje pa tudi njej sledečo vrstico, ne glede na to, ali tudi sama vsebuje zvezdico ali ne.

Napiši program, ki bo izvedel opisano opravilo. Predpostaviš lahko, da so vse vrstice krajše od 200 znakov.

R: 378 **1999.1.4** V državi Utopiji so ugotovili, da denar kvari ljudi, zato vsako leto izvedejo prerazporeditev bogastva. Vsako leto določijo, kolikšno je največje sprejemljivo premoženje. Za vsakega državljana popišejo, koliko premoženja ima; tistim, katerih premoženje presega največji dovoljeni znesek, vzamejo toliko, da mu ostane le še ta največji dovoljeni znesek. Letos bo največje sprejemljivo premoženje določeno kot 150 % povprečnega premoženja. Tako dobljeni denar razdelijo med najrevnejše državljane in to tako, da ima na koncu čim večje število najrevnejših državljanov enako in čim večje premoženje. Pri tem se vrstni red prebivalcev po bogastvu ne sme spremeniti: če je imel A prej vsaj toliko kot B, ima tudi po prerazporeditvi vsaj toliko kot B. **Opiši postopek**, ki bi to dosegel. Predpostavi, da je mogoče denar drobiti na poljubno majhne enote.

Primer: pet prebivalcev z začetnim premoženjem (30, 50, 120, 240, 260); vsota je 700; povprečje je 140; največje še sprejemljivo premoženje je 210; zadnjima dvema poberemo 30 oz. 50; dobimo 80; damo prvima dvema, končno stanje je (80, 80, 120, 210, 210). Nekatere nesprejemljive prerazporeditve so (90, 70, 120, 210, 210) (ker bi se spremenil vrstni red — prvi na seznamu ima zdaj več kot drugi), (40, 110, 120, 210, 210) (vrstni red se sicer ohrani, vendar je najrevnejši tukaj en sam, pri najboljši prerazporeditvi pa imata najrevnejša dva enako premoženje), (60, 60, 160, 210, 210) (najrevnejša dva imata sicer zdaj enako premoženje, vendar bi lahko dobila več).

NALOGE ZA DRUGO SKUPINO

R: 378 **1999.2.1** Dano je n-mestno število v desetiškem zapisu ($n \geq 2$). Na njegovih števkah lahko izvajaš tri osnovne računske operacije: seštevanje, odštevanje in deljenje. Pri deljenju se količnik zaokroži navzdol; pri seštevanju pa se upošteva le ostanek rezultata po deljenju z 10 (na primer: $6 + 7 = 3$). Deljenje z 0 seveda ni dovoljeno, prav tako pa ni dovoljeno odštevanje, če bi bila razlika negativna. Tako je rezultat vsake takšne operacije spet neko n-mestno število v desetiškem zapisu. Za operanda lahko vzameš dve različni števki ali pa dvakrat isto; rezultat nato zapišeš čez eno od števk, lahko tudi čez kakšno od tistih, ki si ju uporabil kot operanda.

Opiši postopek, ki ugotovi, če je mogoče dano n-mestno število z nekim zaporedjem teh operacij preoblikovati v dano drugo n-mestno število; za primere, ko je to mogoče, tudi opiši ustrezno zaporedje operacij (število teh operacij ni pomembno; nič hudega, če jih je veliko). Če problema ne znaš rešiti v splošnem, poskusi za kakšno podmnožico ciljnih števil (npr. taka, ki imajo po več enakih števk, ali pa taka, kjer je ena od števk enaka 1, ali pa soda, ali pa enaka neki potenci števila 2, ipd.). Splošnejše rešitve dobijo več točk; postopek naj bo čim bolj sistematičen.

Primer: število 4567 hočemo spremeniti v 1234. Primerno zaporedje ope-

racij bi bilo $7 - 4 = 3$ ($\rightarrow$ 4563), $6 - 3 = 3$ ($\rightarrow$ 4533), $5 - 4 = 1$ ($\rightarrow$ 1533), $5 - 3 = 2$ ($\rightarrow$ 1233), $1 + 3 = 4$ ($\rightarrow$ 1234). Drugo primerno zaporedje je tudi $4/4 = 1$ ($\rightarrow$ 1567), $1+1 = 2$ ($\rightarrow$ 1267), $1+2 = 3$ ($\rightarrow$ 1237), $2+2 = 4$ ($\rightarrow$ 1234). Še ena možnost je $6+6 = 2$ ($\rightarrow$ 4267), $2+2 = 4$ ($\rightarrow$ 4264), $6/4 = 1$ ($\rightarrow$ 1264), $4 - 1 = 3$ ($\rightarrow$ 1234).

1999.2.2

R: 380

V matriki $m \times n$ imajo elementi matrike lahko le vrednosti 0 ali 1. **Napiši funkcijo** Najvecji, ki vrne velikost stranice največjega kvadrata, ki povsem leži na poljih z enicami. Notranjost kvadrata mora biti zapolnjena z enicami.

Primer:

0	1	0	1	0	1
0	1	1	1	1	0
0	1	1	1	0	1
0	1	1	1	1	1
1	0	1	0	1	0
0	1	1	1	1	1
1	0	0	1	1	1

Pri zgornji matriki ima največji tak kvadrat stranico dolžine 3.

1999.2.3

R: 382

Matija je dobil službo pri projektu 2MS2MM (dva milijona Slovencev, dva milijona megabitov). Podjetje bo zagotavljalo hitro omrežno povezljivost za vse Slovence, zaradi prikaza porabe pa bo za vsakega Slovenca moralo voditi podatke o prometu. Podatek je shranjen v 32-bitnem številu in Matija dobi vsakih pet minut tabelo z dvema milijonoma vrednosti, ki jih mora spraviti na disk.

Zahteve so naslednje:

(a) podatki morajo biti zapisani kar najhitreje,

(b) shraniti se mora zadnjih 1000 vrednosti vsakega števca,

(c) če med zapisovanjem zmanjka napajanja, se morajo zavreči vsi podatki iz danega vzorca,

(d) na zahtevo mora Matija vrniti zadnjih 1000 vrednosti zahtevanega števca, skupaj s časi, ko so bile zajete.

Napiši podprogram Shrani(Stevci: **var array** [1..2000000] **of** integer), ki vrednosti števcev shrani, in podprogram Izpisi(Stevec: integer), ki izpiše zadnjih 1000

vrednosti zahtevanega števca, skupaj s časom, ko je bila vsaka vrednost shranjena, v formatu:

$$\begin{array}{l} \mathit{čas}_1\ \mathit{vrednost}_1 \\ \mathit{čas}_2\ \mathit{vrednost}_2 \\ \ldots\ldots\ldots\ldots \end{array}$$

Podprogram Izpisi naj izpis začne pri najstarejši vrednosti in nadaljuje proti najnovejši. Če je shranjenih vrednosti manj kot 1000, naj funkcija izpiše vse shranjene vrednosti. Funkcija Cas: integer ti vrne trenutni čas. Funkcija Skoci(Datoteka; i: integer) postavi točko branja in pisanja v datoteki na zapis, določen z danim celim številom. Funkcija Izplakni(Datoteka) zagotovi, da so po vrnitvi iz podprograma vsi podatki, ki smo jih doslej zapisali v datoteko, tudi v resnici vpisani na disk (torej izprazni vmesne pomnilnike). Če podprogramu Write v nekem klicu podaš 512 ali manj bytov podatkov, lahko predpostaviš, da bodo ti podatki zapisani na disk atomarno (torej če bo sploh kaj od njih zapisanega na disk, bodo zapisani vsi v celoti; če pa bo pred tem prišlo do prekinitve, ne bo zapisano nič od njih).

Opiši tudi strukturo podatkov na disku.

R: 384

1999.2.4

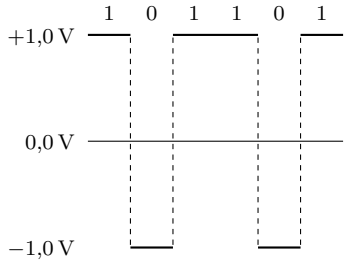
Kadar prek nekega zapisa digitalnih podatkov na magnetnem disku ali traku zapišemo novega, se prejšnja informacija le močno oslabi in jo pretežno prekrije nova, vendar lahko z natančnim branjem pridobimo dovolj informacije, da lahko rekonstruiramo poleg novega tudi prej prisotni zapis. Z dovolj natančnim odčitavanjem lahko rekonstruiramo tudi še starejši (že dvakrat prekriti) zapis ali včasih še več.

Posamezni zaporedni biti so zapisani kot namagnetenje majhnega področja diska v eno od dveh možnih smeri. Pri prehodu čitalne glave prek takega področja se v glavi pojavi električna napetost, ki je pozitivna (približno +1 V), če je na tem mestu zapisani bit 1, in negativna (približno −1 V), če je zapisani bit 0. Zaradi predhodnih zapisov in drugih fizikalnih vzrokov lahko odčitana napetost nekoliko odstopa od idealne vrednosti +1 V ali −1 V.

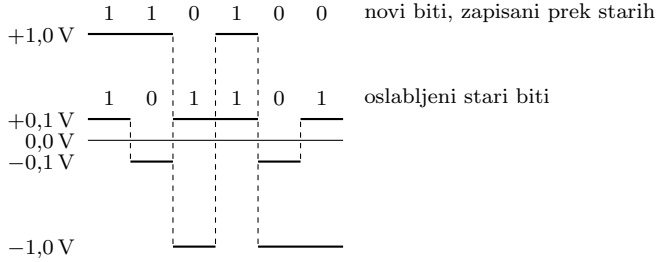
Predpostavimo, da pri pisanju novih podatkov predhodna vrednost na tem mestu oslabi na desetino in tej oslabljeni vrednosti se prišteje nova vrednost. Nova vrednost, ki jo bomo lahko na tem mestu odčitali, je torej $U_{novi} = U_{stari}/10 + U_{bit}$. Primer: če prek enice (+1,0 V) zapišemo ničlo (−1,0 V), bo odčitana napetost na tem mestu $+1{,}0\,\mathrm{V}/10 + (-1{,}0\,\mathrm{V}) = -0{,}9\,\mathrm{V}$. Na novem praznem disku je bilo namagnetenje diska zanemarljivo (blizu 0).

V domači delavnici smo s pomočjo digitalnega osciloskopa, priklopljenega na magnetno glavo, uspeli natančno izmeriti napetosti 256 zaporednih bitov. Te odčitke smo zapisali v datoteko, v vsako vrstico po eno realno število blizu +1 ali −1, ki ustreza enemu bitu. **Napiši program**, ki bo to datoteko prebral in rekonstruiral zadnji, predzadnji in predpredzadnji zapis ter izpisal vsa tri zaporedja bitov.

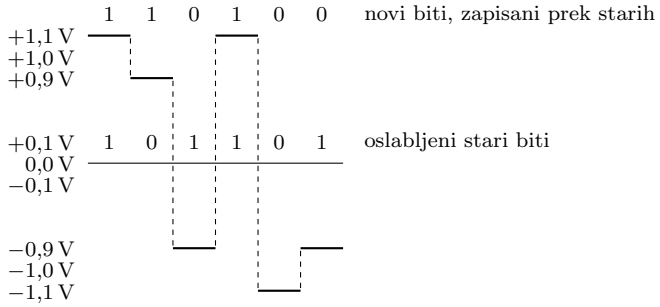
V ilustracijo: najstarejši zaporedni biti 101101 takoj zatem, ko so bili zapisani na prazen disk:



Čez prejšnji zapis zapišemo nove bite 110100. Stari zapis oslabi na desetino in k njemu se prišteje novi zapis. Za lažjo predstavo narišimo v isti diagram obe napetosti:



Rezultat je vsota obeh napetosti iz prejšnje slike:



NALOGE ZA TRETJO SKUPINO

R: 385

1999.3.1

Programer Primož se je znašel v težavnem položaju. Pred leti je napisal podprogram za iskanje v urejeni tabeli, ki ga mora zdaj prilagoditi spremenjenemu problemu.

Podprogram, iz katerega izhaja, PoisciElement (zapisan je spodaj), sprejme štiri parametre: iskani element (ki je kar celo število), tabelo, v kateri išče (tabela mora biti urejena in mora vsebovati vsaj en element) ter indeks prvega ter zadnjega elementa v tabeli. Med tema indeksoma podprogram z bisekcijo poišče element in njegov indeks vrne v parametru Indeks. Rezultat podprograma je v tem primeru true. Če iskanega elementa ni v tabeli, podprogram v parametru Indeks vrne indeks, v katerega mora biti element vstavljen, da bo tabela še vedno urejena. Rezultat je v tem primeru seveda false. Podprogram PoisciElement ima le eno napakico: če je v tabeli več elementov z enako vrednostjo, vrne indeks enega izmed njih, vnaprej pa ne znamo uganiti, ali bo vrnil indeks prvega takega elementa, zadnjega ali katerega od vmesnih. Če mu na primer pošljemo tabelo s petimi enakimi elementi in mejama iskanja 1 in 5, bo podprogram vrnil indeks 3.

Pomagaj Primožu **predelati podprogram** tako, da bo vedno vrnil indeks najbolj levega elementa (najmanjši indeks). V našem izrojenem primeru torej pričakujemo, da bo podprogram vrnil indeks 1. Če iskanega elementa ni v tabeli, se mora predelani podprogram obnašati enako kakor stari. Seveda si želi, da bo novi podprogram približno enako hiter kot stari.

Originalni podprogram, zapisan v pascalu.

```
const N = 1024;
type TabelaT = array [1..N] of integer;

function PoisciElement(Iskani: integer; var Tabela: TabelaT;
                       Prvi, Zadnji: integer; var Indeks: integer): boolean;
var Spodnji, Zgornji, Vmes, Element: integer;
begin
  Spodnji := Prvi; Zgornji := Zadnji;
  while Repeat
    Vmes := (Spodnji + Zgornji) div 2;
    Element := Tabela[Vmes];
    if Iskani = Element then begin
      PoisciElement := true; Indeks := Vmes; exit;
    end
    else if Iskani < Element
      then Zgornji := Vmes - 1
      else Spodnji := Vmes + 1;
  until Spodnji > Zgornji;
  if Iskani > Element
```

```
      then Indeks := Vmes + 1
      else Indeks := Vmes;
   PoisciElement := false;
end; {PoisciElement}
```

Originalni podprogram, zapisan v C++.

```
bool PoisciElement(int iskani, int tabela[], int prvi, int zadnji, int &indeks)
{
   int spodnji = prvi, zgornji = zadnji, vmes, element;
   while (spodnji <= zgornji)
   {
      vmes = (spodnji + zgornji) / 2;
      element = tabela[vmes];
      if (iskani == element) { indeks = vmes; return true; }
      else if (iskani < element) zgornji = vmes - 1;
      else spodnji = vmes + 1;
   }
   if (iskani > element) indeks = vmes + 1; else indeks = vmes;
   return false;
}
```

1999.3.2

R: 386

Véliki vezir ima težave s svojimi ministri. Neprestano sklepajo nove in nove medsebojne zveze, tako da ubogi vezir ne ve več, kje se ga drži glava. Zato se je odločil, da bo izpisal vse možne koalicije.[68] Ker pa v programiranju ni pretirano močan, te prosi za pomoč. Potrebuje enostaven program, ki bo prebral število ministrov in izpisal vse možne koalicije na standardni izhod. Da bo naloga bolj enostavna, naj se ministri imenujejo kar `A`, `B`, `C`, ... Če imamo na primer štiri ministre, naj program izpiše nekaj takega:

```
ABCD
ABC,D
ABD,C
AB,CD
AB,C,D
ACD,B
AC,BD
AC,B,D
AD,BC
A,BCD
A,BC,D
AD,B,C
A,BD,C
A,B,CD
A,B,C,D
```

[68] Tako pravi besedilo naloge iz leta 1999. V resnici naloga sprašuje po vseh možnih razdelitvah ministrov na eno ali več skupin; beseda „koalicija“ običajno pomeni le eno takšno skupino.

Namig: vsaj dve elegantni poti vodita do rešitve — z rekurzijo in z uporabo pomožnih datotek. In nikar se ne ubadajte preveč z obliko izpisa. Da se le vidi, kateri ministri držijo skupaj, pa bo čisto v redu.

R: 389

1999.3.3

Tekmovanje, ki je potekalo po sistemu „vsak proti vsakemu“, je bilo predčasno končano, zato so znani izidi le nekaterih tekem, ne pa vseh. Ker je bilo na koncu kljub vsemu potrebno določiti vrstni red tekmovalcev, so določili naslednje pravilo: tekmovalec A je vsaj tako dober kot tekmovalec B, če je premagal tekmovalca B v medsebojnem dvoboju ali pa je premagal kakšnega tekmovalca C, ki je vsaj tako dober kot tekmovalec B. (Če za tekmovalca A in B velja, da je tekmovalec A vsaj tako dober kot B in obratno, tedaj veljata za enako dobra.)

Sestavi algoritem, ki na podlagi le podmnožice odigranih tekem sestavi tak vrstni red tekmovalcev, v katerem je vsak tekmovalec uvrščen pred vsemi takimi tekmovalci, za katere velja, da je sam vsaj tako dober kot oni, oni pa niso vsaj tako dobri kot on. Bodi pozoren na to, da je lahko veljavnih vrstnih redov tekmovalcev več, algoritem pa naj poišče enega izmed njih.

Primer 1: Če sta na voljo rezultata:

tekmovalec 1 je premagal tekmovalca 2 in
tekmovalec 3 je premagal tekmovalca 4,

tedaj je veljaven katerikoli od naslednjih vrstnih redov:

tekmovalec 1, tekmovalec 2, tekmovalec 3, tekmovalec 4
tekmovalec 1, tekmovalec 3, tekmovalec 2, tekmovalec 4
tekmovalec 1, tekmovalec 3, tekmovalec 4, tekmovalec 2
tekmovalec 3, tekmovalec 4, tekmovalec 1, tekmovalec 2
tekmovalec 3, tekmovalec 1, tekmovalec 4, tekmovalec 2
tekmovalec 3, tekmovalec 1, tekmovalec 2, tekmovalec 4

Pomembno je torej le, da je tekmovalec 1 vedno pred tekmovalcem 2 in tekmovalec 3 vedno pred tekmovalcem 4.

Primer 2: Če so na voljo rezultati:

tekmovalec 1 je premagal tekmovalca 2,
tekmovalec 2 je premagal tekmovalca 3,
tekmovalec 3 je premagal tekmovalca 4,
tekmovalec 4 je premagal tekmovalca 2 in
tekmovalec 5 je premagal tekmovalca 6,

tedaj je veljaven katerikoli vrstni red, v katerem je tekmovalec 5 pred tekmovalcem 6 in tekmovalec 1 pred tekmovalci 2, 3 in 4. Tekmovalci 2, 3 in 4 veljajo za enako dobre. Primerjati jih je mogoče s tekmovalcem 1, ne pa tudi s tekmovalcema 5 in 6.

1999.3.4

R: 392

V podjetju *Pasivna orodja* se ukvarjajo s ponudbo brezplačne storitve elektronske pošte. Ker je ponudba zelo ugodna, imajo zelo veliko prometa. Zaradi tega so prisiljeni deliti računalniške kapacitete na več računalnikov. Na hitri lokalni mreži imajo N enakih računalnikov, ki vsi zaganjajo enak program. Eden izmed njih je glavni računalnik, vsi ostali pa so podporni. Naloga glavnega računalnika, ki edini dobiva zahteve z Interneta, je razdelitev le-teh na podporne računalnike. V primeru, da glavni računalnik odpove, ostanejo podporni računalniki brez dela. Sistemski inženir Matjaž trenutno rešuje ta problem, a ne najde rešitve. Pomagaj mu!

Napiši algoritem, ki zazna izpad glavnega računalnika in sočasno izbere novega. Na mreži isti trenutek ne sme obstajati več kot en glavni računalnik, čas brez glavnega računalnika pa želimo kar se da skrajšati. Vsak računalnik ima svojo enolično številko (npr. številka omrežne kartice).

Uporabi spodaj navedeno okostje programa. Kodo programa vpiši v podprogram, ki je klican enkrat na sekundo. Vsa sporočila, poslana v klicu tega podprograma, zagotovo pridejo do cilja do naslednjega klica podprograma in se pri tem ne izgubljajo.

```
var
  { globalne spremenljivke }

{ zunanje procedure }

{ vrne številko računalnika (je večja od 0), na katerem teče naš program }
function VrniStRacunalnika: integer; external;

{ nastavi stanje računalnika }
procedure PostaviGlavniRac; external;
procedure PostaviPodporniRac; external;

{ pošlje številko Num vsem računalnikom v omrežju (tudi nazaj pošiljatelju) }
procedure PosljiVsem(Num: integer); external;

{ funkcija za sprejemanje podatkov iz mreže }
function Prejmi(var Num: integer): boolean; external;
{ Števila, ki so prišla po mreži do našega računalnika, se zbirajo v vrsti v nekem
  pomožnem pomnilniku. Ta funkcija vzame iz vrste tisto število, ki je najdlje v
  njej, in ga shrani v Num ter vrne true; če pa je bila vrsta prazna, vrne false,
  vrednosti Num pa ne spreminja. }

procedure Zacni;
begin
  { inicializacija, klicana pred prvim klicem podprograma VsakoSekundo }
end; {Zacni}

procedure VsakoSekundo;
begin
  { tu vpišite svojo kodo }
end; {VsakoSekundo}
```

REŠITVE NALOG ZA PRVO SKUPINO

N: 367

R1999.1.1

Da ob prelomu stoletja ne bo zmede, je vsekakor koristno, če lahko letnico predstavimo v celoti, ne pa le njenih zadnjih dveh števk. Vsak byte je dolg 8 bitov in lahko hrani eno od $2^8 = 256$ različnih vrednosti; z dvema bytoma lahko torej predstavimo $256 \times 256 = 65536$ različnih vrednosti, tako da bi lahko, če bi letnice hranili kot 16-bitna dvojiška števila, predstavili datume še za nadaljnjih več deset tisočletij.

Še ena možnost, ki bi tudi delovala kar precej časa, je, da za vsako števko uporabimo 4 bite (saj omogočajo štirje biti 16 različnih vrednosti, mi pa jih potrebujemo le 10) in tako v vsak byte shranimo dve števki. Dva byta tako zadostujeta za štiri števke, torej za vsa leta do vključno 9999.

Če pa hočemo na vsak način ostati pri predstavitvi, ki uporablja le ASCII znake za števke od 0 do 9, bi lahko letnice predstavljali enako kot doslej (torej z dvema števkama, ki povesta desetice in enice), le da bi uvedli dogovor, po katerem števila od `00` do `30` predstavljajo letnice od 2000 do 2030, števila od `31` do `99` pa letnice od 1931 do 1999. Slabost te rešitve je, da ne moremo predstaviti letnic pred 1931 ali po 2030.

N: 367

R1999.1.2

Spodnji podprogram si v spremenljivki `c` zapomni trenutni znak in v `s` še podatek o tem, ali je to samoglasnik; v `pc` in `ps` hrani ta dva podatka za prejšnji znak. Tako ni težko preverjati raznih pogojev. Pogoj, da sme biti `x` le na koncu besede, lahko preverimo pri naslednjem znaku; če je `x` čisto zadnji znak v nizu, je s tem tudi na koncu besede in ga ni treba preverjati še posebej.

```
function Pravo(Sporocilo: var array [1..1000000] of char): boolean;
var i: integer; c, pc: char; s, ps: boolean;
begin
  ps := false; pc := '.'; Pravo := false;
  for i := 1 to 1000000 do begin
    c := Sporocilo[i]; s := c in ['a', 'e', 'i', 'o', 'u'];
    if not (c in ['a'..'z', ' ']) then exit; { nedovoljen znak }
    if c = pc then exit; { podvojen znak }
    if s and ps then exit; { dva samoglasnika zaporedoma }
    if (pc = 'x') and (c <> ' ') then exit; { x ni na koncu besede }
    pc := c; ps := s;
  end; {for}
  Pravo := true;
end; {Pravo}
```

R1999.1.3

N: 367

Spodnji program bere vrstico za vrstico, pri vsaki pa preveri, če slučajno vsebuje kakšno zvezdico. To si zapomni v spremenljivki Najden, v spremenljivki IzpisiNaslednjo pa si zapomni, če je našel zvezdico v prejšnji in mora zato izpisati tudi trenutno vrstico. Nato vrstico, če je to potrebno, izpiše, vrednost Najden pa prenese v IzpisiNaslednjo, da jo bo imel pri roki ob delu z naslednjo vrstico.

```
program Zvezdice(Input, Output);
const
  VrsticaM = 200;
var
  Vrstica: packed array [1..VrsticaM] of char;
  j: integer;
  Najdena: boolean;           { vrstica vsebuje zvezdico }
  IzpisiNaslednjo: boolean;   { izpisati bo treba tudi naslednjo vrstico }
begin
  IzpisiNaslednjo := false;
  while not Eof(Input) do begin
    ReadLn(Vrstica);
    j := 1; Najdena := false;
    while not Najdena and (j <= VrsticaM) do
      if Vrstica[j] = '*' then Najdena := true else j := j + 1;
    if Najdena or IzpisiNaslednjo then WriteLn(Vrstica);
    IzpisiNaslednjo := Najdena;
  end; {while}
end. {Zvezdice}
```

Kot še mnoge druge naloge, ki se tičejo predvsem dela z nizi ali besedili, lahko tudi to nalogo rešimo veliko krajše, če uporabimo kakšen drug programski jezik, na primer perl:

```
perl -ne '$z = /\*/; print if $z || $pz; $pz = $z'
```

S stikalom `-e` smo interpreterju perla naročili, da je program naveden kar kot naslednji parameter v ukazni vrstici (zato je v narekovajih), s stikalom `-n` pa smo mu naročili, naj dani program ponavlja v zanki, po enkrat za vsako vrstico vhodne datoteke. Obe stikali lahko združimo v `-ne`. Program deluje tako kot zgornja pascalska rešitev: v spremenljivko `$z` si zapiše, če trenutna vrstica vsebuje zvezdico (to preveri z regularnim izrazom `\*` — poševnica pred zvezdico je potrebna zato, ker ima zvezdica v regularnih izrazih drugače poseben pomen — dovoli nič ali več ponovitev izraza, ki stoji pred zvezdico); v spremenljivki `$pz` hrani podatek o tem, ali je bila zvezdica v prejšnji vrstici; če je res kaj od tega dvojega, trenutno vrstico izpiše; nato pa vrednost `$z` prenese v `$pz`, da jo bo uporabil pri naslednji vrstici.

N: 368

R1999.1.4

Ves pobrani denar dajmo na kup; najrevnejšemu dajmo toliko, da bo imel potem enako kot drugi najrevnejši. Če to ne gre (torej če denarja na kupu ni dovolj), mu dajmo vse, kar je še ostalo, in končajmo. Nato dajmo prvima dvema najrevnejšima (ki imata zdaj enako) toliko, da bosta imela enako kot tretji najrevnejši. Če to ne gre, dajmo vsakemu pol od tega, kar nam je še ostalo, in končajmo. Nato dajmo trem najrevnejšim toliko, da bodo imeli enako kot četrti; če bi zmanjkalo kupa, dajmo vsakemu tretjino preostanka in končajmo. Tako nadaljujemo, dokler nam kupa ne zmanjka.

Seveda denarja v resnici ni treba deliti takole po kapljicah; lahko tudi z enim pregledom tabele premoženja prebivalcev določimo, koliko bodo imeli po prerazporeditvi najrevnejši ljudje, in nato vsakemu od njih damo toliko denarja, kolikor mu še manjka do te končne vrednosti.

```
const StPreb = ...;
type Tabela = array [1..StPreb] of real;

{ Tabela mora biti urejena naraščajoče.
  Ta podprogram predpostavlja, da smo bogatašem že pobrali toliko,
  za kolikor presegajo največje dovoljeno premoženje, skupna pobrana
  vsota denarja pa je v parametru Kup. }
procedure Razdeli(var Premozenje: Tabela; Kup: real);
var StPrej: integer; { število prejemnikov }
    Vsota: real; { vsota prvotnih premoženj vseh prejemnikov 1..StPrej }
    Novo: real; { novo premoženje najrevnejših po prerazporeditvi }
    i: integer; Zmanjka: boolean;
begin
  StPrej := 0; Vsota := 0; Zmanjka := false;
  while (StPrej < StPreb) and not Zmanjka do
    { Invarianta: kup je dovolj velik, da damo lahko najrevnejšim StPrej − 1
      ljudem toliko, da bodo imeli enako kot oseba StPrej. }
    if Vsota + Kup <= Premozenje[StPrej + 1] * StPrej then Zmanjka := true;
    else begin StPrej := StPrej + 1; Vsota := Vsota + Premozenje[StPrej] end;
  if StPrej > 0 then begin
    Novo := (Vsota + Kup) / StPrej;
    for i := 1 to StPrej do Premozenje[i] := Novo;
  end; {if}
end; {Razdeli}
```

REŠITVE NALOG ZA DRUGO SKUPINO

N: 368

R1999.2.1

Če so vse števke vhodnega števila enake 0, ne moremo iz njega dobiti nobenega drugega števila. Če pa je kakšna števka različna od 0, jo delimo samo s sabo, da dobimo 1; nato jo odštevajmo od vseh drugih, dokler ne dobimo $00\ldots010\ldots0$; prištejmo jo na prvem mestu

in odštejmo na starem, da dobimo $1000\ldots00$. Nato jo prištevajmo na vseh mestih razen na prvih dveh, dokler ne dobimo tam želenih končnih vrednosti; dogovorimo se, da bomo tiste števke odslej pustili pri miru in ostal nam je le še problem, ali lahko iz para $(1,0)$ dobimo vse druge pare (a,b). Ker delamo v desetiškem zapisu in je takih parov le sto, se lahko o tem, da lahko res pridemo do vseh, prepričamo tudi ročno. Lahko pa se o tem prepričamo tudi z naslednjim induktivnim razmislekom, ki bi veljal tudi, če bi delali v kakšnem drugem številskem sestavu, ne le v desetiškem.

Vidimo, da lahko iz $(1,0)$ s prištevanjem in odštevanjem dobimo $(1,1)$, $(0,1)$, $(0,0)$. Recimo, da že znamo priti do vseh parov (a,b), kjer sta a in b manjša ali enaka nekemu k (na začetku naj bo $k=1$). Iz $(1,k)$ s prištevanjem pridemo do $(1,k+1)$ in z odštevanjem do $(0,k+1)$; podobno iz $(k,1)$ do $(k+1,1)$ in $(k+1,0)$; iz slednjega s prištevanjem do $(k+1,k+1)$. Za števila oblike $(a,k+1)$, kjer je a med vključno 1 in k: opazimo, da je potem tudi $k+1-a$ med vključno 1 in k, zato do $(a,k+1-a)$ po predpostavki že znamo priti; iz njega pa s prištevanjem tudi do $(a,k+1)$. Podobno pokažemo za $(k+1,a)$. Tako torej vidimo, da lahko pridemo do vsakega ciljnega števila.

Zanimivo bi bilo razmisliti tudi o primeru, ko imamo eno samo števko (torej $n=1$). Če ta ni ničla (iz katere ne moremo dobiti ničesar drugega), lahko z odštevanjem vedno dobimo 0, z deljenjem vedno 1, seštevanje pa je zdaj v bistvu podvajanje. Zato iz 1 dobimo 2, 4, 8, 6 (ker je $16 \bmod 10 = 6$; iz slednje pa pridemo spet v 2, tako da od tu naprej ne dobimo nič novega). Iz a bi dobili $2a$, ki pa je sod, tako da od tu naprej tudi ne dobimo nič novega. Vidimo torej, da lahko iz neničelnega a dobimo števila $a, 0, 1, 2, 4, 6, 8$, ostalih pa nikakor. Ni pa vedno tako hudo, saj lahko na primer v enajstiškem sistemu dobimo vse neničelne števke: $1 \to 2 \to 4 \to 8 \to 16 \equiv 5 \to 10 \to 20 \equiv 9 \to 18 \equiv 7 \to 14 \equiv 3 \to 6 \to 12 \equiv 1$.

Recimo, da delamo v b-iškem sestavu. Pri katerih b se bo dalo iz 1 s podvajanjem priti do vseh ostalih neničelnih števk? Za začetek opazimo, da b ne sme biti sod (razen $b=2$, ko je stvar trivialna): katerokoli števko podvojimo, je rezultat sod, če pa je večji ali enak b, se od pravega rezultata (ki je sod) odšteje b (ki je tudi sod) in dobimo spet sodo vrednost. Zato pri sodem b s podvajanjem ne moremo priti do nobene lihe števke. Omejimo se zdaj na lihe b. Zahteva, da lahko iz 1 s podvajanjem dobimo vse ostale neničelne števke, je pravzaprav enakovredna zahtevi, naj imajo števila $1, 2, 4, 8, \ldots, 2^{b-2}$ same različne ostanke po modulu b (ker jih je $b-1$, bodo to ravno vse števke $1, 2, \ldots, b-2, b-1$; ostanek 0 ni mogoč, saj potence števila 2 ne morejo biti večkratniki b-ja, ko pa je ta vendar lih). Lahko si pomagamo z Eulerjevim izrekom iz teorije števil; ta pravi, da če sta si a in b tuja, je $a^{\phi(b)} \bmod b = 1$. Pri tem je $\phi(b)$ število b-ju tujih števil iz množice $\{1, \ldots, b-1\}$. Ker je b lih, mu je $a=2$ tuj in lahko uporabimo ta izrek; $\phi(b)$ že po definiciji ne more biti večji od $b-1$, če pa bi bil manjši, bi to pomenilo, da se nam pri podvajanju

števk ponovi števka 1 že po $\phi(b) < b-1$ korakih, ne pa šele po $b-1$ korakih. Torej mora biti $\phi(b) = b-1$; toda to pomeni, da so vsa števila od 1 do $b-1$ tuja b-ju; z drugimi besedami, b mora biti praštevilo. Izkaže se, da je ta pogoj potreben, ne pa še tudi zadosten. Primerna so le praštevila oblike $8m+3$ in $8m+5$, pa še to ne vsa. Med števili do 100 so primerna samo naslednja: 2, 3, 5, 11, 13, 19, 29, 37, 53, 59, 61, 67, 83. Emil Artin je postavil znano domnevo, ki med drugim pravi, da je primernih b-jev neskončno mnogo, vendar to še ni čisto dokazano.[69]

N: 369

R1999.2.2

Najenostavneje bi lahko poiskali največji kvadrat enic tako, da bi se postavili v vsako polje matrike in se vprašali, kolikšen je največji kvadrat z (recimo) spodnjim desnim kotom v tem polju.

```
const m = ...; n = ...;
type Matrika = array [1..m, 1..n] of integer;

function Najvecji(var a: Matrika): integer;
var i, j, k, s, sm: integer; ok: boolean;
begin
  sm := 0; { sm je stranica največjega doslej najdenega kvadrata }
  for i := 1 to m do for j := 1 to n do begin
    { Poiskali bomo največji kvadrat enic s spodnjim desnim
      kotom na polju a[i, j]. Njegova stranica bo s. }
    s := 0; ok := true;
    while (s < i) and (s < j) and ok do begin
      { Vemo, da obstaja primeren kvadrat s stranico s.
        Ali obstaja tudi tak s stranico s + 1? }
      k := 1; s := s + 1;
      while (k <= s) and ok do begin
        if a[i - s + 1, j - k + 1] = 0 then ok := false;
        if a[i - k + 1, j - s + 1] = 0 then ok := false;
        k := k + 1;
      end; {while k}
    end; {while s}
    { Če se je zanka ustavila, ker smo pri poskusu povečevanja kvadrata odkrili ničlo,
      je bilo zadnje povečanje s-ja neupravičeno. }
    if not ok then s := s - 1;
    { Če je to nov največji kvadrat, si ga zapomnimo. }
    if s > sm then sm := s;
  end; {for i, j}
  Najvecji := sm;
end; {Najvecji};
```

[69] Glej MathWorld *s. vv.* "Primitive Root", "Euler's Totient Theorem", "Artin's Conjecture"; *The On-Line Encyclopedia of Integer Sequences*, A001122; in nit z naslovom "2 is a primitive root" v *sci.math* 26.–28. januarja 2001.

Vendar pa je ta rešitev precej neučinkovita. Če je matrika polna samih enic, bomo pri vsakem polju (i, j) odkrili kvadrat s stranico $\min\{i, j\}$ in pri tem pregledali vsa polja tega kvadrata. Časovna zahtevnost take rešitve je zato $O(mn(\min\{m, n\})^2)$.

Program si lahko prihrani veliko dela, če si sproti pomaga z rezultati tega, kar je doslej že naredil. Označimo s $s(i, j)$ stranico največjega kvadrata enic z desnim spodnjim kotom v polju (i, j). Ko razmišljamo o kvadratih z desnim spodnjim kotom v polju (i, j), že poznamo $s(i-1, j)$, $s(i, j-1)$ in $s(i-1, j-1)$. No, če je v polju $a(i, j)$ naše vhodne matrike ničla, že tako ali tako vemo, da bo $s(i, j) = 0$. Pa recimo zdaj, da je $a(i, j) = 1$ in je potemtakem $s(i, j) = S \geq 1$. Znotraj tega kvadrata samih enic (s stranico S in desnim spodnjim kotom v polju (i, j)) ležijo tudi kvadrati samih enic s stranico $S-1$ in desnimi spodnjimi koti v poljih $(i-1, j)$, $(i-1, j-1)$ in $(i, j-1)$; torej imamo pogoje: $s(i-1, j) \geq S-1$, $s(i-1, j-1) \geq S-1$ in $s(i, j-1) \geq S-1$. Po drugi strani, čim nek S ustreza tem trem pogojem (poleg tega pa je še $a(i, j) = 1$), takoj sledi, da so v kvadratu s stranico S in desnim spodnjim kotom v polju (i, j) same enice. Tako vidimo, da pri $a(i, j) = 1$ velja $s(i, j) = 1 + \max\{s(i-1, j)s(i-1, j-1), s(i, j-1)\}$. Lepo pri opisanem razmisleku je tudi to, da nam tabele s ne bo treba poznati v celoti, pač pa moramo, ko razmišljamo o $s(i, j)$, poznati le $s(i, j')$ za $j' < j$ ter $s(i-1, j')$ za $j' \geq j-1$, ostalo pa lahko sproti pozabljamo.

```
function Najvecji(a: Matrika): integer;
var s: array [0..n] of integer; i, j, sm, sp, sn: integer;
begin
  { Tabela s se na začetku nanaša na i = 0, kar leži
    zunaj tabele in si zato mislimo tam same ničle. }
  sm := 0; for j := 0 to n do s[j] := 0;
  for i := 1 to m do begin
    sp := 0;
    for j := 1 to n do begin
      { V polju s[j − 1] je zdaj vrednost s(i, j − 1);
        v polju s[j] je vrednost s(i − 1, j);
        vrednost s(i − 1, j − 1) pa imamo začasno v sp.
        Izračunajmo s(i, j) in jo začasno shranimo v sn. }
      if a[i, j] = 0 then sn := 0
      else sn := 1 + Min(s[j], s[j − 1], sp);
      if sn > sm then sm := sn; { Nov največji kvadrat. }
      { V naslednji iteraciji bomo potrebovali vrednost s(i − 1, j), ki je trenutno
        še v s[j]; zapomnimo si jo v sp, da bomo lahko zdaj v s[j] vpisali
        pravkar izračunano vrednost s(i, j), ki jo trenutno hranimo v sn. }
      sp := s[j]; s[j] := sn;
    end; {for j}
  end; {for i}
  Najvecji := sn;
```

end; {*Najvecji*}

Zdaj imamo z vsakim poljem (i, j) le konstantno mnogo dela, tako da je časovna zahtevnost celotnega postopka le $O(mn)$. Asimptotično pravzaprav bolje sploh ne bi moglo biti, saj potrebujemo toliko časa že samo za branje matrike a.

Mimogrede, zanimiva različica te naloge bi bila taka, pri kateri bi iskali največji pravokotnik samih enic (največji v tem smislu, da ima največjo ploščino). Glej npr. 4. nalogo na 1. izbirnem tekmovanju univerze v Portu za ACM SWERC, 2. maja 2001 (#836 na `online-judge.uva.es`); in David Vandevoorde, *The maximal rectangle subproblem*, Dr. Dobb's Journal, april 1998.

N: 369

R1999.2.3

Zaradi zahteve, naj se podatki zapišejo kar najhitreje, smo se odločili, da zapišemo vseh dva milijona števcev na disk kar v enem bloku, na začetku bloka pa naj bo še čas, ko so bili ti števci zajeti. Po tisoč pisanjih si bo tako v datoteki sledilo tisoč takih blokov; ob naslednjem (tisočprvem) pisanju bomo morali nekako doseči, da bo računalnik na prvi blok pozabil, saj smo rekli, da hočemo hraniti le zadnjih tisoč vrednosti vsakega števca. Še najlažje je, če takrat z novimi vrednostmi kar povozimo najstarejši blok, torej ravno tiste stare vrednosti, ki bi jih tako ali tako radi zavrgli. Pri izpisovanju vrednosti števcev bomo morali vedeti, kje se začnejo in kje končajo, zato bomo v ta namen na začetku datoteke, pred vsemi podatki o števcih, hranili še dve celi števili, ki povesta, kateri blok je najstarejši in koliko je vseh skupaj.

Zahtevo, naj se ob prekinitvi napajanja med pisanjem zadnja meritev zavrže, lahko upoštevamo tako, da v datoteki v resnici pustimo prostora za 1001 blok, tako da je med najnovejšim in najstarejšim dovolj prostora še za en blok. Najprej zapišimo nove vrednosti v ta dodatni blok, nato pa, če je bilo blokov prej manj kot tisoč, povečajmo podatek o številu blokov (na začetku datoteke); če pa jih je bilo že prej tisoč, povečajmo podatek o indeksu najstarejšega bloka (to nam zagotovi, da bomo tistega najstarejšega zdaj pozabili oz. ga bomo ob naslednjem pisanju povozili z novimi vrednostmi). Tako, če pride do prekinitve napajanja med pisanjem novih vrednosti (ali pa po pisanju teh in pred popravljanjem podatkov na začetku datoteke), nismo ničesar izgubili, saj smo pisali v neuporabljeni pomožni blok. Po pisanju v pomožni blok in pred popravljanjem glave datoteke kličimo za vsak primer še Izplakni,[70] da ne bi slučajno operacijskemu sistemu kasneje prišlo na misel najprej zapisati na

[70] Mnogi prevajalniki pascala ponujajo podprogram Flush, ki na prvi pogled daje vtis, kot da dela nekako to, kar bi mi tule želeli od funkcije Izplakni. Vendar pa je Flush običajno zamišljen le za datoteke tipa text, pri katerih standardna knjižnica (enota System) praviloma vzdržuje nek majhen medpomnilnik za odloženo pisanje podatkov. Flush naj bi podatke, ki so še v tem medpomnilniku in čakajo, da bodo zapisani, predal operacijskemu sistemu; vendar pa slednji praviloma tudi sam skrbi za odloženo zapisovanje podatkov na disk, tako

disk naše popravke glave datoteke, nato pa bi zmanjkalo elektrike, še preden bi se na disk zapisali tudi podatki v pomožnem bloku.

```
const N = 2000000; M = 1000;
type StevciT = array [1..N] of integer;

procedure Shrani(var Stevci: StevciT);
var F: file of integer;
    Prvi, Koliko, Kam, i: integer;
begin
  Assign(F, 'shramba'); Reset(F);
  Read(F, Prvi, Koliko);
  Kam := (Prvi + Koliko) mod (M + 1);
  Skoci(F, 2 + Kam * (N + 1));
  Write(F, Cas);
  for i := 1 to N do Write(F, Stevci[i]);
  Izplakni(F);
  if Koliko >= M then begin
    Skoci(F, 0); Write(F, (Prvi + 1) mod (M + 1));
  end else begin
    Skoci(F, 1); Write(F, Koliko + 1);
  end; {if}
  Izplakni(F); Close(F);
end; {Shrani}

procedure Izpisi(Stevec: integer);
var F: file of integer;
    OdKod, Koliko, T, X: integer;
begin
  Assign(F, 'shramba'); Reset(F);
  Read(F, OdKod, Koliko);
  while Koliko > 0 do begin
    Skoci(F, 2 + OdKod * (N + 1));
    Read(F, T);
    Skoci(F, 2 + OdKod * (N + 1) + Stevec);
    Read(F, X);
```

---

da ni nujno, da so naši podatki po klicu Flush res že zapisani na disk.

Implementacija podprograma Izplakni bo v splošnem odvisna od prevajalnika in operacijskega sistema. Spodnja različica je primerna, če je naš prevajalnik FreePascal, operacijski sistem pa kakšen iz družine Windows:

```
uses Dos, Windows; { FileRec je iz Dos, FlushFileBuffers pa iz Windows. }
procedure Izplakni(var f: file);
  begin FlushFileBuffers(FileRec(f).Handle) end;
```

Spremenljivke tipa **file** so tu namreč v resnici strukture tipa FileRec. Polje Handle je številka odprte datoteke, ki jo moramo uporabljati pri klicih funkcij operacijskega sistema. FlushFileBuffers je standardna funkcija iz vmesnika (APIja) Win32.

```
      WriteLn(T, ' ', X);
      Koliko := Koliko − 1; OdKod := (OdKod + 1) mod (M + 1);
    end; {while}
    Close(F);
end; {Izpisi}

procedure Inicializacija;
var F: file of integer;
begin
    Assign(F, 'shramba'); Rewrite(F);
    Write(F, 0, 0); Close(F);
end; {Inicializacija}
```

N: 370

R1999.2.4

Ker so stare vrednosti desetkratno oslabljene, ne morejo vplivati na predznak meritve; ta nam torej pove, katera je bila zadnja shranjena vrednost, nato pa lahko to vrednost odštejemo in preostanek pomnožimo z deset, pa smo (približno, če zanemarimo morebitne majhne motnje na traku) rekonstruirali stanje pred zadnjim pisanjem na ta del traku.

```
program Smetarjenje(Input, Output);
var Meritev: real;
    j, Starost: integer;
begin
    for j := 1 to 256 do begin
      ReadLn(Meritev);
      Write('Meritev: ', Meritev:6:3, ' ');
      for Starost := 0 to 2 do begin { postopek lahko večkrat ponovimo }
        { ugotovimo nazadnje zapisani bit — ta prevladuje nad ostalimi }
        if Meritev > 0
          then begin Write('1'); Meritev := Meritev − 1 end
          else begin Write('0'); Meritev := Meritev + 1 end;
        { odšteli smo že najbolj svežo vrednost, ojačajmo preostanek }
        Meritev := 10 * Meritev;
        { V praksi se po nekaj korakih približamo pragu motenj in šuma,
          zato starejših podatkov ne moremo več zanesljivo rekonstruirati. }
      end; {for Starost}
      WriteLn;
    end; {for j}
end. {Smetarjenje}
```

REŠITVE NALOG ZA TRETJO SKUPINO

R1999.3.1

N: 372

Predelani podprogram se od prvotnega v bistvu razlikuje le v majhni podrobnosti: namesto da bi iskali podani element, se pretvarjamo, kot da iščemo le malo manjšo vrednost. Če torej najdemo vrednost, ki je manjša od iskane, se odločimo, da je premajhna in iščemo dalje (tako kot v originalnem podprogramu). Če pa najdemo vrednost, ki je večja ali enaka, se naredimo, da je prevelika in prav tako iščemo dalje. Podprogram se zato vedno ustavi takrat, ko spodnja meja iskanja postane večja od zgornje. V težavo zaidemo le, ker mora podprogram vrniti true ali false v odvisnosti od tega, ali je iskana vrednost prisotna v tabeli. Zato uvedemo novo logično spremenljivko Nasel, ki jo postavimo na true, kadar zaznamo element z iskano vrednostjo.

Rešitev v pascalu:

```
function PoisciNajmanjsiElement(Iskani: integer; var Tabela: TTabela;
                                Prvi, Zadnji: integer; var Indeks: integer): boolean;
var
  Spodnji, Zgornji, Vmes: integer;
  Element: integer;
  Nasel: boolean;
begin
  Nasel := false;
  Spodnji := Prvi;
  Zgornji := Zadnji;
  repeat
    Vmes := (Spodnji + Zgornji) div 2;
    Element := Tabela[Vmes];
    if Iskani = Element then Nasel := true;
    if Iskani <= Element
      then Zgornji := Vmes - 1
      else Spodnji := Vmes + 1;
  until Spodnji > Zgornji;
  if Iskani > Element
    then Indeks := Vmes + 1
    else Indeks := Vmes;
  PoisciNajmanjsiElement := Nasel;
end; {PoisciNajmanjsiElement}
```

Rešitev v C++:

```
bool PoisciNajmanjsiElement(int iskani, int tabela[], int prvi, int zadnji, int &indeks)
{
  int spodnji = prvi, zgornji = zadnji, vmes, element;
```

```
  bool nasel = false;
  while (spodnji <= zgornji)
  {
    vmes = (spodnji + zgornji) / 2;
    element = tabela[vmes];
    if (iskani == element) nasel = true;
    if (iskani <= element) zgornji = vmes - 1; else spodnji = vmes + 1;
  }
  if (iskani > element) indeks = vmes + 1; else indeks = vmes;
  return nasel;
}
```

N: 373

R1999.3.2

Do rešitve nas pripelje indukcija: „Kaj se zgodi, če dodamo še enega ministra?“ Z enim ministrom je problem enostavno rešljiv — tvori lahko le eno koalicijo, sam s sabo. Denimo sedaj, da smo rešili problem za N ministrov, in dodajmo še enega. Rešitve problema N ministrov si ogledamo eno za drugo. Pri vsaki je N ministrov razdeljenih v nekaj koalicij (od 1 do N). Novi minister se lahko pridruži katerikoli od koalicij, lahko pa ostane sam. Če je na primer N enak 3 in si ogledujemo razdelitev `A BC`, lahko iz nje nastanejo tri nove razdelitve: `AD BC`, `A BCD` in `A BC D`. To razmišljanje nas neposredno pripelje do dveh rešitev — rekurzivne in z uporabo datotek. Obe uporabljata indukcijo in posamično dodajanje ministrov.

Rekurzivna rešitev:

```
program MinistriA;

const N = 11; { Največ ministrov }

  function IzpisiKoalicije(Stevilo: integer): integer;
  type Koal = set of 1..N;
       Iter = record
                Koliko: integer;
                Koalicije: array [1..N] of Koal;
              end;
  var a: Iter; Stevec: integer;

    { Izpiše vsa taka razbitja ministrov na koalicije, ki jih je moč dobiti,
      če v razbitje a dodamo vse ministre od Naslednji do Stevilo. }
    procedure RekurzivnoIzpisi(var a: Iter; Naslednji: integer);

      procedure Izpisi(var a: Iter);

        procedure IzpisiEno(k: Koal);
        var i: integer;
        begin
          for i := 1 to Stevilo do
            if i in k then Write(Chr(i + Ord('A') - 1));
```

```
      end; {IzpisiEno}

    var i: integer;
    begin {Izpisi}
      for i := 1 to a.Koliko do begin
        if i > 1 then Write(',');
        IzpisiEno(a.Koalicije[i]);
      end; {for}
      WriteLn; Stevec := Stevec + 1;
    end; {Izpisi}

  var i: integer;
  begin {RekurzivnoIzpisi}
    a.Koalicije[a.Koliko + 1] := [];
    for i := 1 to a.Koliko + 1 do begin
      { Poskusimo dodati naslednjega ministra v koalicijo i.
        i = a.Koliko + 1 pomeni, da tvori sam svojo (novo) koalicijo. }
      a.Koalicije[i] := a.Koalicije[i] + [Naslednji];
      if i = a.Koliko + 1 then a.Koliko := a.Koliko + 1;
      { Če je še kaj ministrov, izvedimo rekurzivni klic; sicer izpišimo razbitje. }
      if Naslednji = Stevilo
        then Izpisi(a)
        else RekurzivnoIzpisi(a, Naslednji + 1);
      { Izbrišimo ministra iz i-te koalicije, da ga bomo lahko dali še v }
      a.Koalicije[i] := a.Koalicije[i] - [Naslednji];      { kakšno drugo. }
    end; {for}
    { V zadnji iteraciji smo ustvarili novo koalicijo, pa jo zdaj pobrišimo. }
    a.Koliko := a.Koliko - 1;
  end; {RekurzivnoIzpisi}

begin {IzpisiKoalicije}
  Stevec := 0; a.Koliko := 0;
  RekurzivnoIzpisi(a, 1);
  IzpisiKoalicije := Stevec;
end; {IzpisiKoalicije}

var Stevilo: integer;
    StMin: integer;
begin {MinistriA}
  Write('Vnesi število ministrov (od 1 do ', N, '): ');
  ReadLn(StMin);
  if (StMin < 1) or (StMin > N) then WriteLn('Od 1 do ', N, ', sem rekel!')
  else begin
    Stevilo := IzpisiKoalicije(StMin);
    WriteLn('Vseh možnih koalicij je ', Stevilo, '.');
  end;
end. {MinistriA}
```

Oglejmo si še rešitev z uporabo datotek. Za razliko od prejšnjega programa, ki koalicije izpiše na zaslon, jih ta inačica pusti kar v datoteki, kar je verjetno bolj uporabno.

```
program MinistriB;

const N = 11; { Največ ministrov }

  procedure IzracunajKoalicije(Ministrov: integer);
  var f: array [1..2] of text;
      fi, fo: integer;
      i, j: integer;
      s, t: string;
      ch: char;
      Stevec: longint;
  begin
    Assign(f[1], '1'); Rewrite(f[1]); WriteLn(f[1], 'A'); Reset(f[1]); fi := 1;
    Assign(f[2], '2'); Rewrite(f[2]); fo := 2;
    for i := 2 to Ministrov do begin
      { V datoteki fi so vsa razbitja prvih i − 1 ministrov na koalicije.
        V datoteko fo bomo izpisali vsa razbitja prvih i ministrov. }
      Stevec := 0; ch := Chr(i + Ord('A') − 1);
      while not Eof(f[fi]) do begin
        { Preberimo naslednje možno razbitje prvih i − 1 ministrov. Presledek, ki ga
          bomo dodali na konec niza, bo deloval kot stražar za spodnji stavek if. }
        ReadLn(f[fi], s); s := s + ' ';
        { Poskusimo dodati novega ministra v vsako od koalicij. }
        for j := 1 to Length(s) do begin
          if s[j] = ' ' then begin
            t := s; Insert(ch, t, j);
            WriteLn(f[fo], Copy(t, 1, Length(t) − 1));
            Stevec := Stevec + 1;
          end; {if}
        end; {for j}
        WriteLn(f[fo], s, ch);    { Lahko pa ima novi minister sam svojo koalicijo. }
        Stevec := Stevec + 1;
      end; {while}
      if i = Ministrov then WriteLn(f[fo], Stevec, ' koalicij');
      { Zamenjamo vhodno in izhodno datoteko. }
      Rewrite(f[fi]); fi := 3 − fi;
      Reset(f[fo]); fo := 3 − fo;
    end; {for i}
    Close(f[1]); Close(f[2]); Erase(f[fo]);
    WriteLn('Koalicije so shranjene v datoteki ', fi);
  end; {IzracunajKoalicije}

var StMin: integer;
```

```
begin
  Write('Vnesi število ministrov (od 1 do ', N, '): ');
  ReadLn(StMin);
  if (StMin < 1) or (StMin > N)
    then WriteLn('Od 1 do ', N, ' sem rekel!')
    else IzracunajKoalicije(StMin);
end. {MinistriB}
```

Pa še rešitev v pythonu:

```
def koalicije(n):
  if n == 0: yield []; return
  c = chr(ord('A') + n - 1)
  for p in koalicije(n - 1):
    yield p + [c]
    for i in range(len(p)):
      yield p[:i] + [p[i] + c] + p[i + 1:]

# Primer uporabe:
import sys
for p in koalicije(int(sys.stdin.readline())):
  print p
```

Mimogrede, števila, ki nam povedo, na koliko načinov se lahko n ministrov razdeli v k nepraznih skupin, se imenujejo Stirlingova števila druge vrste in jih označujejo na različne načine, npr. $S(n,k)$, $s_n^{(k)}$ in $\left\{{n \atop k}\right\}$. Zgoraj opisani razmislek nam je pokazal, da bi jih lahko računali po formulah $\left\{{n \atop k}\right\} = \left\{{n-1 \atop k-1}\right\} + k\left\{{n-1 \atop k}\right\}$ za $n > 0$ in $\left\{{0 \atop 0}\right\} = 1$, $\left\{{0 \atop k}\right\} = 0$ za $k \neq 0$. Vsote $B_n = \sum_{k=1}^{n} \left\{{n \atop k}\right\}$, ki nam povedo skupno število vseh razbitij n ministrov, pa se imenujejo Bellova števila.[71] Vrednosti B_n za $n = 1, \ldots, 11$ so: 1, 2, 5, 15, 52, 203, 877, 4 140, 21 147, 115 975, 678 570.

R1999.3.3

Edine omejitve v zvezi z medsebojnim vrstnim redom tekmovalcev v naši razvrstitvi nastopijo v primerih, ko je a vsaj tako dober kot b, slednji pa ni vsaj tako dober kot a; takrat imamo omejitev, da mora biti a v razvrstitvi pred b. N: 374

Rekli smo, da sta dva tekmovalca enako dobra natanko tedaj, ko je vsak od njiju vsaj tako dober kot drugi. Očitno je ta lastnost tranzitivna: če je a enako dober kot b, slednji pa enako dober kot c, sta tudi a in c enako dobra. Torej je smiselno govoriti o celih skupinah enako dobrih tekmovalcev (te skupine naj bodo tudi „maksimalne“ — v tem smislu, da če je v skupini nek tekmovalec a, so v njej tudi *vsi* ostali tekmovalci, ki so enako dobri kot a). V taki skupini

[71] Glej MathWorld *s. vv.* “Stirling Number of the Second Kind”, “Bell Number”; *The On-Line Encyclopedia of Integer Sequences*, A000110.

je vsak enako dober kot vsi ostali in je zato vseeno, v kakšnem medsebojnem vrstnem redu zapišemo tekmovalce iz take skupine.

Definirajmo zdaj, da je neka skupina A vsaj tako dobra kot neka druga skupina B natanko tedaj, ko je vsaj eden od tekmovalcev iz A vsaj tako dober kot vsaj eden od tekmovalcev iz B. Pri skupinah se ne more zgoditi, da bi bili dve različni skupini enako dobri, kajti to bi pomenilo, da je neki $a \in A$ vsaj tako dober kot neki $b \in B$, neki $b' \in B$ pa vsaj tako dober kot neki $a' \in A$; ker sta a in a' oba iz A, sta oba enako dobra; torej je a' vsaj tako dober kot a in zato tudi vsaj tako dober kot b; in podobno sta b in b' enako dobra, ker sta oba iz B; zato pa je b vsaj tako dober kot b', in ker je slednji vsaj tako dober kot a', je tudi b vsaj tako dober kot a'. Iz tega sledi, da sta si b in a' enako dobra, zaradi prej ugotovljene tranzitivnosti pa tudi, da so si vsi tekmovalci iz A in vsi iz B enako dobri. Toda to je nemogoče, ker potem A in B ne bi mogli biti dve različni skupini enako dobrih tekmovalcev.

Zagotovo obstaja neka skupina, za katero velja, da ni nobena vsaj tako dobra kot ta. Kajti če bi za vsako obstajala neka druga vsaj tako dobra, bi recimo bila A_2 vsaj tako dobra kot A_1, pa A_3 vsaj tako dobra kot A_2 in tako naprej; prej ali slej bi se nam skupine začele ponavljati in bi to pomenilo, da je neka A_i vsaj tako dobra kot A_j, ta pa vsaj tako dobra kot A_i; toda to bi pomenilo, da sta enako dobri, kar pa smo že v prejšnjem odstavku spoznali za nemogoče.

Naj bo torej A neka skupina, kateri ni nobena druga vsaj tako dobra. Torej ni nobenih omejitev, ki bi zahtevale, da mora biti kateri iz tekmovalcev te skupine v vrstnem redu za nekim tekmovalcem kakšne druge skupine. Zato lahko kar takoj postavimo na začetek vrstnega reda vse tekmovalce te skupine (njihov medsebojni vrstni red je lahko poljuben, saj so si vsi enako dobri). V nadaljevanju našega razmišljanja nam tekmovalci iz skupine A ne bodo povzročali nikakršnih težav več — kot smo pravkar videli, omejitev, da bi moral biti kdo drug pred njimi, sploh ni; lahko pa obstajajo omejitve, da mora biti kdo drug za njimi, kar pa bo gotovo izpolnjeno, saj smo jih postavili na začetek vrstnega reda. Torej se lahko v nadaljevanju obnašamo, kot da tekmovalcev iz skupine A sploh nikoli ni bilo.

Zdaj zagotovo obstaja neka skupina B, kateri ni nobena druga vsaj tako dobra (lahko da je za B to veljalo še prej, lahko pa je bila prej A vsaj tako dobra kot B, ampak zdaj smo A pač v mislih zavrgli). Enak razmislek kot prej tudi zdaj pokaže, da lahko B takoj postavimo v vrstni red, saj nobena od preostalih skupin ne more zahtevati, da bi bila v njem pred tekmovalci skupine B. Zdaj lahko tudi na B pozabimo. Potem poiščemo naslednjo primerno skupino (tako, ki ji ni nobena vsaj tako dobra) in tako nadaljujemo, dokler ne obdelamo vseh skupin.

Naš algoritem je torej tak:

1 Poišči skupine enako dobrih tekmovalcev.

2 Ponavljaj, dokler je ostalo še kaj skupin:
3 Naj bo A poljubna taka skupina, za katero velja, da ni nobena druga vsaj tako dobra kot A.
4 Izpiši vse tekmovalce skupine A v poljubnem vrstnem redu.
5 Zbriši skupino A.

Pravzaprav bi se spodobilo, če bi te korake opisali malo podrobneje. Posvetimo se točki (1), s katero je še največ dela; o ostalih lahko bralec razmisli sam. Skupine enako dobrih tekmovalcev lahko poiščemo takole. Na začetku za vsakega tekmovalca a vemo, kateri so ga premagali v neposrednih tekmah; naj bo $P(a)$ množica vseh teh. Označimo s $P^\star(a)$ množico vseh, ki so vsaj tako dobri kot a. Na začetku lahko postavimo $P^\star(a) := P(a)$, nato pa za vsakega tekmovalca b, ki ga dodamo v $P^\star(a)$, dodamo v $P^\star(a)$ še vse tekmovalce iz $P(b)$, kajti če je b vsaj tako dober kot a, tisti iz $P(b)$ pa so b-ja nekoč premagali, so tudi oni vsaj tako dobri kot a. Ko tako za vsakega tekmovalca a poznamo $P^\star(a)$, gremo lahko za vsakega a in za vse $b \in P^\star(a)$ pogledat, če je slučajno tudi $a \in P^\star(b)$, in če je res tako, vemo, da sta a in b enako dobra.

Algoritem *SkupineEnakoDobrih:*
Vhod: množica tekmovalcev T
za vsakega $a \in T$ še množica $P(a)$ tistih, ki so ga premagali.
Izhod: *ŠtSkupin* pove, koliko skupin je;
skupina[a] pove, v kateri skupini je tekmovalec a.

```
 1  for each a ∈ T do
 2      skupina[a] := 0; P*(a) := {}; Q := {a};
 3      while Q ≠ {} do
 4          naj bo b poljuben element Q;
 5          Q := Q − {b};
 6          for each c ∈ P(b) do if c ∉ P*(a) then
 7              P*(a) := P*(a) ∪ {c}; Q := Q ∪ {c};
 8  ŠtSkupin := 0
 9  for each a ∈ T do if skupina[a] = 0 then
10      ŠtSkupin := ŠtSkupin + 1; skupina[a] := ŠtSkupin;
11      for each b ∈ P*(a) do
12          if a ∈ P*(b) then skupina[b] := ŠtSkupin;
```

V teoriji grafov bi rekli, da smo iz rezultatov tekmovanja naredili usmerjen graf, v katerem vsakega tekmovalca predstavlja neka točka, usmerjena povezava od a do b pa je prisotna natanko tedaj, ko je a premagal b-ja. To, da je a vsaj tako dober kot b, pomeni, da je točka b v grafu dosegljiva iz točke a; skupinam enako dobrih tekmovalcev ustrezajo krepko povezane komponente grafa. Potem naredimo nov graf, v katerem je po ena točka za vsako krepko povezano komponento in povezava od A do B natanko tedaj, če obstaja v

prvotnem grafu kakšna povezava od kakšnega $a \in A$ do kakšnega $b \in B$. Ta novi graf je zanesljivo acikličen in če ga topološko uredimo, dobimo vrstni red, v katerem je treba izpisati povezane komponente prvotnega grafa.

Gornji algoritem za odkrivanje skupin enako dobrih tekmovalcev (torej: za iskanje krepko povezanih komponent) ni najboljši možni, vendar je zelo preprost. Običajni algoritem za iskanje krepko povezanih komponent je učinkovitejši, vendar je bolj zapleten in ga tu ne bomo opisovali. Najde se ga v mnogih knjigah o algoritmih (npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 23.5 v prvi izdaji, 22.5 v drugi).

N: 375

R1999.3.4

Glavni računalnik naj pošlje vsako sekundo svojo številko vsem računalnikom v omrežju. Ostali računalniki poslušajo in če v neki sekundi ne prejmejo iz omrežja nobene številke, si to razlagajo kot izpad glavnega računalnika. V tem primeru vsak računalnik, ki to opazi, razpošlje vsem svojo številko; za glavnega obvelja tisti, ki je imel najmanjšo številko. Vsak podporni računalnik poskuša najprej prebrati prispele podatke iz omrežja; če glavni računalnik obstaja in deluje normalno, bo prišla le njegova številka in je stvar opravljena; če je glavni računalnik izpadel in je že več pomožnih to opazilo ter poslalo svoje številke, lahko določimo za glavnega tistega z najmanjšo številko; sicer pa pošljemo svojo številko v omrežje in bomo v naslednji sekundi preverili, ali smo postali glavni računalnik ali ne (kajti možno je, da je v istem času poslal svojo številko še kdo drug in mogoče bo on postal glavni, ne pa mi).

Upoštevati moramo, da lahko sporočilo potuje do drugih računalnikov skoraj celo sekundo. Poleg tega ne bi bilo realistično pričakovati, da se podprogram VsakoSekundo kliče na vseh računalnikih naenkrat. Recimo, da pošlje računalnik 1 sporočilo računalniku 2 ob času t in da sporočilo pride do računalnika 2 ob času $t + 1 - \varepsilon$; in recimo, da se na računalniku 2 izvede VsakoSekundo ob času $t + 1 - 2\varepsilon$, ko sporočilo do njega še ni prišlo; računalnik 2 bo torej sporočilo videl šele ob času $t + 2 - 2\varepsilon$, skoraj dve sekundi po tistem, ko je bilo odposlano.

Če torej ob nekem klicu VsakoSekundo opazimo, da od strežnika v zadnji sekundi nismo dobili nobenega sporočila, to še ne pomeni, da je strežnik izpadel iz omrežja. Čisto mogoče je, da je z njim vse v redu, le da smo njegovo predzadnje sporočilo prevzeli že ob prejšnjem klicu VsakoSekundo, njegovo zadnje pa se je malo zakasnilo in nas še ni doseglo, zato ob trenutnem klicu VsakoSekundo pač ne vidimo od glavnega računalnika nobenega sporočila. Zato lahko o izpadu glavnega računalnika zanesljivo govorimo šele, če v dveh zaporednih klicih VsakoSekundo opazimo, da nas ne čaka nobeno sporočilo. Računalnik naj se tudi ne razglasi za glavnega, dokler ni zmagal pri glasovanju v dveh zaporednih sekundah — po prvi sekundi namreč še obstaja možnost, da je poslal svojo številko še kak računalnik z manjšo številko od naše, ki še ni videl našega sporočila, mi pa še ne njegovega. Zato, če bi se naš računalnik takoj razglasil

za glavnega, bi se lahko zgodilo, da bi se tisti drugi kasneje tudi, pa bi imeli potem v sistemu dva glavna računalnika, ki bi vpila drug mimo drugega.

```
var SemGlavni: boolean; { Trenutno stanje računalnika. }
    MinStPrejSek,       { Najmanjša številka, ki smo jo prejeli v prejšnji sekundi. }
    StSek: integer;     { Zacni postavi StSek na 0, VsakoSekundo jo poveča za 1. }

procedure Zacni;
begin
  SemGlavni := false;
  MinStPrejSek := -1;
  StSek := 0;
  PostaviPodporniRac;
end; {Zacni}

procedure VsakoSekundo;
var St, MinSt, Glavni: integer;
begin
  StSek := StSek + 1;
  if SemGlavni then begin
    PosljiVsem(VrniStRacunalnika);
  end
  else begin
    { Katera je najmanjša številka, prejeta v zadnji sekundi? }
    MinSt := -1;
    while Prejmi(St) do if (MinSt = -1) or (St < MinSt) then MinSt := St;
    { Katera je najmanjša številka, prejeta v zadnjih dveh sekundah? }
    Glavni := MinStPrejSek;
    if (Glavni = -1) or ((MinSt > 0) and (MinSt < Glavni)) then Glavni := MinSt;
    { Poglejmo zdaj, kako je z glasovanjem. }
    if Glavni = -1 then begin
      { Glavnega računalnika ni; predlagajmo sebe za glavnega. Če smo se ravnokar
        zbudili (StSek = 1), tega raje ne storimo, saj se lahko po naključju
        zgodi, da v tisti ≤ 1 sekundi, kolikor dolgo smo budni, ne dobimo
        nobenega sporočila, čeprav v omrežju v resnici obstaja glavni računalnik.
        V tem primeru bi delali zgolj zmedo, če bi zdaj začeli vpiti svojo številko. }
      if StSek >= 2 then PosljiVsem(VrniStRacunalnika);
    end
    else begin
      { Glasovanje je v teku, glavni bo tisti z najmanjšo številko. Če smo to mi,
        se spodobi, da pošljemo takoj svojo številko, drugače bomo to storili šele
        čez eno sekundo in se bodo drugi medtem še pregovarjali. Preden pa se
        dokončno razglasimo za glavnega, bi radi, da bi bila naša številka najmanjša
        dve sekundi zaporedoma. To je potrebno zaradi možnosti, da se naša
        sporočila na poti zadržijo eno sekundo in smo lahko zato šele po dveh
```

```
        sekundah prepričani, da ni še kje kak drug računalnik, ki za našo zmago
        ne ve in še kar vpije svojo številko. }
      if VrniStRacunalnika = Glavni then begin
        if MinStPrejSek = Glavni then
          begin SemGlavni := true; PostaviGlavniRac; end;
        PosljiVsem(VrniStRacunalnika);
      end else begin
        SemGlavni := false;
        PostaviPodpornoRac;
      end; {if}
    end; {if}
    MinStPrejSek := MinSt;
  end; {if}
end; {VsakoSekundo}
```

Morebitna slabost te rešitve je, da se ne trudi preprečiti istočasnega obstoja več glavnih računalnikov. Do tega bi lahko prišlo, če bi bilo nekaj povezav v omrežju prekinjenih in bi omrežje zaradi tega razpadlo na več nepovezanih delov. Računalniki v tistih delih omrežja, iz katerih se dosedanjega glavnega računalnika ne vidi, bi mislili, da je glavni računalnik izpadel, tako da bi zdaj vsak del omrežja izglasoval svoj glavni računalnik. To je težko preprečiti, saj računalniki, ki glavnega ne vidijo več, ne vedo, ali je ta prenehal delovati ali pa je prišlo le do prekinitve kakšne omrežne povezave. Lahko pa preprečimo vsaj to, da po ponovni vzpostavitvi povezav ostane v omrežju več glavnih računalnikov, ki vpijejo drug mimo drugega. Začetek podprograma VsakoSekundo lahko popravimo tako, da tudi glavni računalnik posluša, kaj pošiljajo drugi računalniki, in če sliši nižjo številko od svoje, razglasi sebe za podpornega:

```
if SemGlavni then begin
  MinSt := −1;
  while Prejmi(St) do if (MinSt = −1) or (St < MinSt) then MinSt := St;
  if (MinSt > −1) and (MinSt < VrniStRacunalnika) then
    { Eden od drugih glavnih računalnikov ima manjšo številko
      kot mi, zato postanimo podporni računalnik. }
    begin SemGlavni := false; PostaviPodporniRac end
  else { Naš računalnik lahko ostane glavni. }
    PosljiVsem(VrniStRacunalnika);
end
```

24. državno tekmovanje v znanju računalništva (2000)

NALOGE ZA PRVO SKUPINO

2000.1.1

Starejši si, kot misliš! Koliko sekund je minilo med začetkom (0 h) dneva, v katerem si se rodil, in začetkom današnjega dneva? Na prvi pogled se zdi, da jih je $24 \cdot 60 \cdot 60 \cdot$ *število dni* med obema datumoma. To je skoraj res, ušteli smo se le za kakšen ducat sekund.

R: 403

Ozadje naloge: Sekunda (enota za čas) je bila včasih definirana s pomočjo trajanja dneva.Vedno natančnejša astronomska opazovanja so pokazala, da Zemljino vrtenje ni enakomerno, saj nanj vplivajo različne sile, na primer plimovanje, in tudi, da se Zemlja suče vse počasneje; dnevi torej niso vsi enako dolgi. Tako „nenatančna“ definicija sekunde ni več zadoščala znanosti in tehniki, zato so leta 1967 definirali sekundo s pomočjo nihanja cezijevega atoma, ki je mnogo enakomernejše od vrtenja Zemlje. Času, ki ga dobimo s štetjem tako definiranih sekund, pravimo mednarodni atomski čas (s francosko kratico TAI).

Tudi čas UTC, ki ga kažejo naše običajne ure, teče enako hitro kot TAI; vsi uporabljamo isto (novo, atomsko) definicijo trajanja sekunde. Ker pa smo vajeni, da je sonce v najvišji legi točno ob dvanajstih (zanemarimo časovne cone, denimo, da smo v Londonu), se vsako leto sproti izmeri trenutna hitrost in zaostajanje vrtenja Zemlje in po potrebi (približno enkrat letno, a ne vsako leto) vrine v UTC ena dodatna (prestopna) sekunda. Tako je imela na primer zadnja minuta v decembru 1998 po dogovoru eno sekundo več kot ostale minute, torej 61 sekund. Z drugimi besedami: 1. januarja 1999 ob 0 h smo naše ure premaknili za eno sekundo nazaj.

Naloga: Tabela datumov, ko so bile vrinjene prestopne sekunde, je na voljo v datoteki. V vsaki vrstici sta dva podatka: datum (leto-mesec-dan) tik po tem, ko je bila ob polnoči vrinjena prestopna sekunda, poleg njega pa je skupno število prestopnih sekund (torej razlika TAI − UTC), vrinjenih do tega datuma, in ki velja do nadaljnjega (do naslednjega datuma):

```
1972-07-01 11
1973-01-01 12
1974-01-01 13
1975-01-01 14
1976-01-01 15
1977-01-01 16
1978-01-01 17
1979-01-01 18
1980-01-01 19
```

```
1981-07-01 20
1982-07-01 21
1983-07-01 22
1985-07-01 23
1988-01-01 24
1990-01-01 25
1991-01-01 26
1992-07-01 27
1993-07-01 28
1994-07-01 29
1996-01-01 30
1997-07-01 31
1999-01-01 32
```

Napiši program, ki bo prebral rojstni datum (kot niz 10 znakov), potem današnji datum, in s pomočjo tabele v datoteki prestopnih sekund izračunal in izpisal število sekund, ki je minilo med obema datumoma.

Da se ti ne bo treba ukvarjati s poznavanjem koledarja in preračunavanjem datumov v nizih, si lahko pomagaš s podprogramsko funkcijo MJD, ki iz datuma, podanega kot parameter (10-znakovni niz), izračuna zaporedno številko tega dneva od nekega dogovorjenega fiksnega začetka štetja. Tako se poenostavi tudi primerjanje datumov.

```
type DatumT = packed array [1..10] of char;
function MJD(Datum: DatumT): integer; external;
```

R: 404

2000.1.2

Operacijski sistem skrbi za dodeljevanje pomnilnika procesom, ki tečejo na njem. Zaželeno je, da ima ta postopek takšne lastnosti, da je dodeljevanje in sproščanje pomnilnika hitro, njegova razdrobljenost majhna, knjigovodski podatki sistema o kosih pomnilnika pa hitro dosegljivi in neobsežni.

Nekaterim takšnim lastnostnim ustreza metoda dodeljevanja, pri kateri sistem daje pomnilnik na voljo v kosih, velikih po 2^n bajtov: 1 bajt, 2 bajta, 4 bajte, 8 bajtov... **Napiši funkcijski podprogram**, ki bo za dano število m zaprošenih bajtov vrnil velikost dodeljenega pomnilnika kot število, zaokroženo na prvo potenco števila 2, ki ni manjša od m. Pri $m = 0$ pa naj vrne kar 0, saj tisti, ki je zaprosil za 0 bajtov, pomnilnika očitno v resnici sploh ne potrebuje. Pojasni prednosti in slabosti svojega funkcijskega podprograma.

Tabela za nekaj m prikazuje vrednost te funkcije:

m	0	1	2	3	4	5	6	7	8	9	10	...
$f(m)$	0	1	2	4	4	8	8	8	8	16	16	...

R: 411

2000.1.3

Napiši program, ki prebere ocene za 10 predmetov in izpiše, ali je uspeh negativen ali pozitiven; če je uspeh pozitiven, naj izpiše tudi povprečno oceno. Za vsak predmet je podana ena ocena,

ki je celo število med 1 in 5. Uspeh je negativen, če je vsaj en predmet ocenjen z 1, sicer pa pozitiven.

2000.1.4

R: 412

Janez ima na svojem računalniku podatkovno zbirko s celotnim seznamom svojih CDjev. Za vsak CD ima podatke o izvajalcu, naslov CDja in seznam vseh skladb na CDju. Zdaj želi svoj seznam, ki je že urejen po izvajalcu in imenu albuma, izpisati na lepši način, tako da se pri izpisu ne bodo po nepotrebnem ponavljala imena izvajalcev in naslovi albumov.

Na primer, namesto:

```
Blesavi bend, 3 lahki komadi, Moja prva ljubezen
Blesavi bend, 3 lahki komadi, Moja druga ljubezen
Blesavi bend, 3 lahki komadi, Moja zadnja ljubezen
Blesavi bend, Singl, Tristo kosmatih
Nori fantje, Najvecji neuspehi, Spet si sla k drugemu
```

naj **program** izpiše:

```
Blesavi bend  3 lahki komadi     Moja prva ljubezen
                                 Moja druga ljubezen
                                 Moja zadnja ljubezen
              Singl              Tristo kosmatih
Nori fantje   Najvecji neuspehi  Spet si sla k drugemu
```

Pri tem naj program pazi, da je med imenom izvajalca in imenom albuma ter med imenom albuma in naslovom skladbe vedno vsaj za dva presledka prostora, stolpci pa naj bodo vsi levo poravnani. Vrstica je lahko poljubno dolga, naj pa ne bo daljša, kot je potrebno.

Na voljo imaš dva podprograma:

ZacniSeznam povzroči, da se seznam podatkov bere od začetka.

Komad(**var** Izvajalec, Album, Skladba: string): boolean vrne false, če ni več podatkov, sicer vrne true, v parametrih pa vrne polne podatke o naslednji skladbi v vrstnem redu.

NALOGE ZA DRUGO SKUPINO

2000.2.1

R: 413

Nekatera opravila na računalniku so take narave, da jih je treba ponavljati ob določenih časih. Tako na primer lahko želimo, da vsako uro ob polni uri zapiska zvonček; da se vsako minuto požene nek program za merjenje obremenjenosti računalnika; da se varnostno arhiviranje datotek požene vsak dan natanko dvakrat: točno opoldne in

ponoči ob 15 minut čez drugo uro po lokalnem času; Omejimo se le na opravila, ki niso vezana na datum in za katera zadošča točnost ene minute.

Za opis časov, ob katerih se mora zgoditi neko opravilo, nam tako zadoščata dve polji: podatek za uro (0..23) in podatek za minuto (0..59). Za periodična opravila (*vsako* minuto oziroma *vsako* uro) se dogovorimo, da vrednost -1 pomeni „vsako“.

Tako lahko za primer zapišemo prej našteta opravila:

ura	minuta	opravilo
-1	0	zvonček
-1	-1	vsakominutna meritev
12	0	backup
2	15	backup

Glede na to, da je ura opravila podana kot lokalni čas, moramo biti previdni dvakrat na leto: ko se spomladi lokalni čas prestavi za eno uro naprej na poletni čas, in jeseni, ko skočimo za eno uro nazaj ponovno na normalni čas. Premik je vedno izveden ob polni uri za polno uro in na štetje minut ne vpliva.

Da v računalniku ni prevelike zmede, ki bi jo povzročile nenadne prestavitve ure, njegova notranja ura teče enakomerno in meri standardni čas (UTC ali po starem GMT). Krajevni čas v Sloveniji dobimo tako, da času UTC prištejemo odmik v urah, ki je pozimi $+1$ in poleti $+2$.

Po sprejetem dogovoru se uveljavitev ali razveljavitev poletnega časa vedno izvede na določen dan ponoči ob 1 h po UTC. Spomladi ob tej uri se torej urni odmik v Sloveniji spremeni iz $+1$ na $+2$, jeseni pa nazaj na $+1$.

Tudi v takem dnevu, ki ima le 23 oziroma 25 ur, se morajo opravila izvajati po načelu „najmanjšega presenečenja“: na opravila, ki se izvajajo vsako uro, prestavitev lokalnega časa ne sme vplivati; nočno arhiviranje, za katero je predpisana določena ura, pa se mora opraviti natanko enkrat tisti dan: ko se lokalna ura prvič ta dan ujame z zahtevano uro (jeseni), ali, če te ure v tem dnevu ni (pomladi), ob času ki bi veljal, če se lokalna ura ne bi še prestavila.

Napiši podprogram:

```
procedure PolnaMinuta(UraUTC, Minuta, Odmik: integer);
```

za katerega operacijski sistem jamči, da bo pognan ob vsaki polni minuti. Kot parametre dobi standardni čas (UraUTC med 0 in 23 ter Minuta med 0 in 59) ter Odmik, to je celo število ur, ki jih je treba prišteti k UraUTC, da dobimo lokalni čas. V Sloveniji je odmik pozimi vedno $+1$ in poleti $+2$. Primer: spomladi v zaporednih minutah okrog uvedbe poletnega časa bo naš podprogram klican z naslednjimi trojicami parametrov: $(0, 58, +1)$, $(0, 59, +1)$, $(1, 0, +2)$, $(1, 1, +2)$, $(1, 2, +2)$, lokalna ura pa v teh trenutkih kaže: 1:58, 1:59, 3:00, 3:01, 3:03.

Program naj vsakokrat prebere datoteko z opravili (kje se nahaja, ni predpisano, izberi po želji; v vsaki vrstici sta po dve števili, preostanek vrstice je naziv opravila) in naj pokliče:

procedure Opravi(Opravilo: NizT); **external**;

z nazivom opravila za vsako tako opravilo, ki se mora v tej minuti začeti izvajati. Podprogram Opravi se vrne takoj in ne čaka, da se bo opravilo tudi zaključilo. Definiraš lahko svoje globalne spremenljivke in predpostaviš, da je njihova začetna vrednost 0.

2000.2.2

R: 414

Na ravnini so podani pari točk, ki predstavljajo leva spodnja in desna zgornja oglišča pravokotnikov s stranicami, vzporednimi s koordinatnima osema. **Napiši podprogram**, ki poišče presek vseh pravokotnikov in izpiše koordinati spodnjega levega oglišča in zgornjega desnega oglišča tako določenega pravokotnika.

2000.2.3

R: 414

Na svoj ročni računalnik (dlančnik, palmtop) želiš redno prenašati nekatere spletne vsebine (novice, kino spored, ipd.), ki so že objavljene na različnih strežnikih po Sloveniji. Ker pa imajo ročni računalniki nekatere omejitve, zaradi katerih originalne HTML strani niso povsem primerne za direktni prikaz (majhen zaslon, nekateri imajo tudi črnobel zaslon), moraš napisati program, ki bo originalne HTML strani „oskubil“ tako, da bodo primerne za pregledovanje na ročnem računalniku.

Zapis HTML uporablja posebne oznake za označevanje delov teksta in strukture, vse pa se začnejo z znakom „<“ in končajo z „>“, vmes pa je ime elementa in njegovi atributi. Primer:

```
<HTML>
  <HEAD>
    <TITLE>Primer HTML strani</TITLE>
  </HEAD>
  <BODY BGCOLOR="white">
    <H1>Naslov: primer HTML strani</H1>
    Privzet font, <FONT FACE="Helvetica" SIZE="1" COLOR="blue">
    spremenjen font</FONT>.
  </BODY>
</HTML>
```

Pri predelavi HTML strani želimo doseči dvoje: odstraniti želimo vse odvečne atribute, tako recimo želimo namesto `<BODY BGCOLOR="white">` imeti v rezultatu samo `<BODY>`. Nato pa se želimo znebiti še odvečnih (imenujmo jih kar „prepovedanih“) HTML oznak, ki nimajo vpliva na prikaz na ročnem računalniku ali pa ga celo kvarijo. V zgornjem primeru se želimo znebiti oznake `<FONT FACE=...>` in seveda tudi `</FONT>` (če bi se samo prve, bi izhodni HTML zapis vseboval napako: konec HTML elementa, ki nima svojega začetka). Če programu torej navedemo, da se želimo znebiti le oznak `FONT`, mora biti izhod programa pri zgornjem primeru naslednji:

```
<HTML>
  <HEAD>
    <TITLE>Primer HTML strani</TITLE>
  </HEAD>
  <BODY>
    <H1>Naslov: primer HTML strani</H1>
    Privzet font,
    spremenjen font.
  </BODY>
</HTML>
```

Vsebina med odstranjenima oznakama `FONT` je seveda ostala nespremenjena.

Na voljo imaš seznam znakovnih nizov, ki predstavljajo „prepovedane“ oznake, oziroma tiste, ki jih želimo popolnoma odstraniti. Seznam je dolg n elementov:

- v Pascalu:
  const n = 10;
  var Prepovedana: **array** [1..n] **of** string;
- v C-ju:
  #define SIZE 256
  #define N 10
  char Prepovedana[N][SIZE];

Oznake HTML so sestavljene iz alfanumeričnih znakov (črke in številke).

Napiši proceduro ali program, ki odstranjuje HTML atribute iz oznak in v celoti tiste HTML oznake, ki so navedene v seznamu Prepovedana. Bral boš s standardnega vhoda (Input v pascalu, stdin v C) in rezultat izpisal na standardni izhod (Output v pascalu, stdout v C).

R: 415

2000.2.4

Slovenska nogometna reprezentanca je bila v pretekli sezoni zelo uspešna. Dobila je celo nagrado svetovne nogometne zveze za največji napredek v sezoni. Nas pa zanima, ali je slovenska reprezentanca naredila tudi največji skok na lestvici reprezentanc. **Napiši algoritem**, ki bo ugotovil, katera reprezentanca je naredila največji skok na lestvici.

Podatki o vrstem redu reprezentanc se nahajajo v dveh datotekah. V datoteki `stanje98.txt` so reprezentance naštete v vrstnem redu, kakršen je bil ob koncu leta 1998, v datoteki `stanje99.txt` pa so reprezentance naštete v vrstnem redu s konca leta 1999. Vsaka reprezentanca je v svoji vrstici, ta pa vsebuje ime reprezentance in njeno identifikacijsko številko.

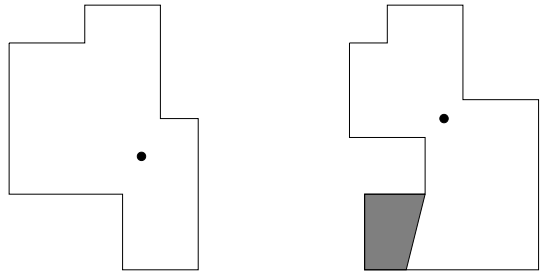
NALOGE ZA TRETJO SKUPINO

2000.3.1

R: 416

Imaš prijatelja, ki je glavni varnostnik v veliki trgovini. Ena od njegovih nalog je, da mora zagotoviti stalen video nadzor v vseh nadstropjih trgovine. Trgovina ima omejen proračun, zato želi uporabiti samo eno kamero za vsako nadstropje. Kamere lahko gledajo v vse smeri.

Prvi problem je določiti mesto, kamor bi lahko postavili kamero za posamezno nadstropje. Edina zahteva je, da mora biti s tega mesta vidno celotno nadstropje. Na spodnji sliki je levo nadstropje take oblike, da ga je možno opazovati samo z eno kamero, medtem ko v desnem primeru nikakor ne moremo postaviti kamere tako, da bi videla celoten prostor.



Preden bodo poskusili postaviti kamere, hoče tvoj prijatelj vedeti, ali je v nekem nadstropju sploh mogoče najti primerno mesto, kamor bi lahko postavili kamero. Tukaj pride na vrsto tvoja naloga. Podan imaš načrt nadstopja, ugotoviti pa moraš, ali je sploh možno postaviti kamero tako, da bi videla vsak del nadstropja.

Napiši funkcijo, ki bo ob danih vhodnih podatkih vrnila `true`, če je kamero možno postaviti tako, da bo videla celoten prostor in `false`, če tega ni možno narediti.

Stene so vedno vzporedne koordinatnima osema. Vhodni podatek je seznam točk (x, y), ki med seboj povezane v smeri urinega kazalca, določajo stene nadstropja.[72]

2000.3.2

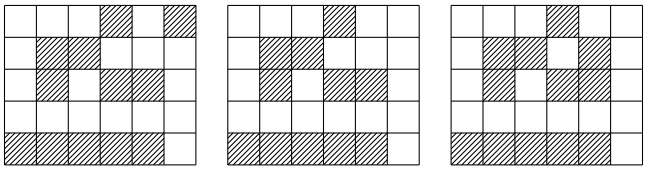
R: 418

Dan je labirint, ki temelji na pravokotni karirasti mreži (vsako polje je lahko prosto ali pa zazidano). Po labirintu se lahko premikamo, seveda le po prostih poljih; z enega gremo lahko na drugo polje le v primeru, če imata skupno eno od stranic. Sprehod po labirintu, pri katerem nobenega polja ne obiščemo po večkrat, imenujemo *pot*. Dolžino poti definiramo kot število polj, ki jih pri tej poti obiščemo, vključno z začetnim

[72] To je naloga D z ACMovega jugozahodnoevropskega študentskega tekmovanja v programiranju (SWERC 1997, Ulm, 23. nov. 1997); #588 v zbirki na `online-judge.uva.es`.

in končnim poljem. O labirintu vemo, da je zgrajen tako, da za vsak par polj obstaja natanko ena pot med tema dvema poljema.

Primer: od treh labirintov na sliki ustreza temu pogoju le skrajni levi labirint.



Radi bi ugotovili, kako dolga je najdaljša pot, ki jo lahko opravimo v tem labirintu. **Opiši** (čim hitrejši) **postopek**, s katerim bi to ugotovil. Lahko si misliš, da je labirint podan z matriko:

```
type
  Polje = (Prosto, Zid);
const
  Visina = ...;
  Sirina = ...;
var
  Labirint: array [1..Visina, 1..Sirina] of Polje;
```

Primer: za skrajni levi labirint na zgornji sliki je rešitev 15 (tako dolga je npr. pot med najbolj desnima prostima poljema v prvi vrstici).[73]

R: 422

2000.3.3

Varčni napredni kmetovalec Janez je opazil, da mu vsako leto otroci pojedo vse češnje v njegovem sadovnjaku, in se odločil, da sadovnjak ogradi z bodečo žico. Ker pa je varčen, želi postaviti natanko toliko ograje, da bo z njo obdal vsa drevesa. Janez je premeril svoj vrt in vsakemu drevesu določil koordinate njegove lege. S svojim novim računalnikom želi izračunati dolžino ograje, da bo zajela ravno vsa drevesa, a se mu je zataknilo pri postopku, s katerim želi ugotoviti, katerih dreves se bo ograja dotikala.

Janezu prišepni **postopek**, ki bo iz dvojic drevesnih koordinat poiskal tiste koordinate, ki se jih ograja dotika. Te koordinate naj postopek izpiše.

R: 426

2000.3.4

Prvo, na kar pomislimo ob besedi „Internet“, je takorekoč neomejen dostop do brezštevilne množice podatkov. Druga misel pa je: „Zakaj je moja povezava tako počasna?“ Tako za prvo kot za drugo so v veliki meri „krivi“ usmerjevalniki (*router*), preko katerih dostopamo do svetovnega medmrežja. V tej nalogi si bomo ogledali, kako (naj bi) usmerjevalniki reševali težave s počasnostjo povezav.

[73] To je naloga E z ACMovega srednjeevropskega študentskega tekmovanja v programiranju (CERC 1999, Praga, 12.–13. nov. 1999).

Zelo poenostavljeno, a smiselno za to nalogo, si lahko usmerjevalnik predstavljamo kot škatlo z n vhodnimi cevmi (imenovanimi tudi „vhodni tokovi“, *streams*) in eno izhodno cevjo. Sam promet po ceveh sestoji iz paketkov različnih dolžin, kjer dolžina paketka predstavlja *količino podatkov*. V nekem trenutku se lahko zgodi, da je količina prihajajočih podatkov večja, kot jih lahko takrat zapusti usmerjevalnik in zato jih usmerjevalnik začasno skladišči (*buffering*). Za potrebe te naloge bomo predpostavili, da usmerjevalnik *vse* prispele podatke tudi slej ko prej odpošlje.

Vrstni red, kako naj bodo paketki odposlani iz usmerjevalnika, je pomemben in je stvar usmerjevalnika — vrstni red odhajajočih podatkov določa *politiko razvrščanja* (*scheduling policy*). Idealna politika je, da tok podatkov iz vsake vhodne cevi dobi v določenem časovnem obdobju enako količino izhodne cevi — se pravi, da je količina odposlanih podatkov (*ne* število paketkov!) približno enaka za vsako vhodno neprazno cev. (V resničnih usmerjevalnikih običajno dodatno utežimo posamezno vhodno cev — tista cev, ki prihaja od vira, kateri je pripravljen več plačati, dobi večji delež izhodne cevi.) Kako učinkovita je ta politika in kako pravična je, določa *kakovost postrežbe* (*quality of service, QoS*).

Opišite in naredite (implementirajte) podatkovno strukturo s pripadajočimi operacijami, ki bo zagotavljala čim bolj pravično odpošiljanje prispelih paketkov. Vaša rešitev naj ima operacije: Pripravi(str), ki postavi strukturo str v začetno stanje, Vstavi(str, pkt, cev), ki bo poklicana, ko se bo na cevi cev pojavil paketek pkt, in Naslednji(str, pkt), ki bo poklicana, ko bo izhodna cev pripravljena za oddajo naslednjega paketka. Slednja operacija je funkcija in vrne true, če je paketek na voljo; če pa ga ni, vrne false. Paketek, ki je na voljo za pošiljanje, dobi v parametru pkt. Predpostavite lahko, da bo funkcija Naslednji samodejno poklicana, ko se bo v strukturi pojavil prvi paketek.

Predpostavite lahko tudi, da bo vedno veljalo $0 \leq \mathsf{cev} < n$.

Pri svoji rešitvi imate na voljo funkcijo Velikost(pkt), ki vrne velikost paketa pkt. Zopet, bolj učinkovita in bolj pravična bo vaša rešitev, več točk boste dobili.

REŠITVE NALOG ZA PRVO SKUPINO

R2000.1.1

N: 395

Z branjem datumov prestopnih sekund lahko ugotovimo, koliko prestopnih sekund je bilo vrinjenih do posameznega datuma. To je število sekund v tisti vrstici datoteke, ki ima najkasnejši datum, manjši ali enak iskanemu datumu.

Med rojstnim in današnjim dnem je torej poleg ustreznega števila dni (kar dobimo kot razliko vrednosti, ki ju za ta dva datuma vrne funkcija MJD) minilo še nekaj sekund — natančneje, tiste prestopne sekunde, ki so se nabrale v dnevih od rojstnega do pred današnjim.

```
program Starost(Input, Output, Prestopne);
type
  DatumT = packed array [1..10] of char;
var
  Prestopne: text;
  Rojstvo, Danes, Kdaj: DatumT;
  Starost, PrestopnihSek: integer;
  PrestopnihSekObRojstvu, PrestopnihSekDanes: integer;

  function MJD(Datum: DatumT): integer; external;

begin
  PrestopnihSekObRojstvu := 0; PrestopnihSekDanes := 0;
  ReadLn(Rojstvo); ReadLn(Danes);
  while not Eof(Prestopne) do begin
    ReadLn(Prestopne, Kdaj, PrestopnihSek);
    if MJD(Kdaj) <= MJD(Rojstvo) then
      PrestopnihSekObRojstvu := PrestopnihSek;
    if MJD(Kdaj) <= MJD(Danes) then
      PrestopnihSekDanes := PrestopnihSek;
  end; {while}
  Starost := (MJD(Danes) - MJD(Rojstvo)) * 24 * 60 * 60 +
             (PrestopnihSekDanes - PrestopnihSekObRojstvu);
  WriteLn('Od rojstnega do današnjega datuma (ob polnoči) je minilo ',
          Starost, ' sekund.');
end. {Starost}
```

Mimogrede, MJD pomeni *modified Julian day* in je definiran kot MJD := JD − 2400000,5. JD (*Julian day*) šteje čas v dnevih od poldneva, 1. januarja 4713 pr. n. š., MJD pa zato od polnoči, 17. novembra 1858. Glej npr. `http://tycho.usno.navy.mil/mjd.html` in `http://en.wikipedia.org/wiki/Julian_day`.

N: 396

R2000.1.2

Nalogo lahko rešimo na veliko različnih načinov. Eden od najpreprostejših je, da v zanki pregledujemo vse večje potence števila 2, dokler ne naletimo na prvo tako, ki je vsaj tolikšna kot število, ki smo ga dobili kot parameter.

```
function Zaokrozi1(x: integer): integer;
var y: integer;
begin
  if x > 0 then y := 1 else y := 0;
  while y < x do y := y * 2;
  Zaokrozi1 := y;
end; {Zaokrozi1}
```

Če za vhodni parameter x velja $2^{k-1} < x \leq 2^k$, se bo zanka izvedla k-krat.

Če zapišemo x v dvojiškem sestavu in ugasnemo vse prižgane bite razen najvišjega, dobimo največjo potenco števila 2, ki je manjša ali enaka x. Recimo tej vrednosti $g(x)$. Na primer: iz $29 = 11101_2$ dobimo $10000_2 = 16$. Nas pa zanima najmanjša potenca števila 2, ki je večja ali enaka x; recimo tej vrednosti $f(x)$. Če je x potenca števila 2, je $g(x) = x = f(x)$, sicer pa je $f(x) = 2g(x)$; tako torej, če izračunamo $g(x)$, ne bo težko dobiti tudi $f(x)$. Lahko pa $f(x)$ izračunamo tudi tako, da upoštevamo, da je $f(x) = 2g(x-1)$, ker je največja potenca števila 2, manjša ali enaka $x-1$, zanesljivo manjša od x in jo moramo zato pomnožiti z 2, da dobimo najmanjšo potenco števila 2, večjo ali enako x.

Oglejmo si zdaj, kako se lahko lotimo ugašanja prižganih bitov. Število x je v računalniku verjetno predstavljeno v dvojiški obliki; na najnižjih mestih je mogoče nekaj ničel, prej ali slej pa nastopi prva enica (če x ni kar enak 0). Če x zmanjšamo za 1, se tiste ničle spremenijo v enice, enica pred njimi pa v ničlo:

$$\begin{aligned} x &= \ldots 1000 \ldots 00, \\ x-1 &= \ldots 0111 \ldots 11, \\ \text{torej } x \textbf{ and } (x-1) &= \ldots 0000 \ldots 00. \end{aligned}$$

Operacija x := x **and** (x − 1) bi torej ugasnila najnižjo enico v številu x. Tako lahko ugašujemo enice eno za drugo; tik preden postane x enak 0, je prižgana le še najvišja enica iz prvotne vrednosti — takrat imamo torej vrednost 2^k, če je bil $2^k \le x < 2^{k+1}$. Zdaj torej število ponovitev zanke ni nujno enako k, ampak je enako številu enic v dvojiškem zapisu vrednosti x, to pa je vedno manj ali enako k, razen pri $x = 2^{k+1} - 1$, ki ima $k+1$ enic.

```
function Zaokrozi2(x: integer): integer;
var xPrvotni, y: integer;
begin
  xPrvotni := x; if x <= 0 then y := 0;
  while x > 0 do begin
    y := x;
    x := x and (x − 1);
  end; {while}
  if xPrvotni > y then y := y * 2;
  Zaokrozi2 := y;
end; {Zaokrozi2}
```

Ko se zanka konča, vsebuje y vrednost, ki jo je imel x, tik preden smo v njem ugasnili še zadnjo (najvišjo) enico. Torej, če je bila prvotna vrednost x na intervalu $2^k \le x < 2^{k+1}$, bo y po koncu zanke enak 2^k. Ravno to pa tudi potrebujemo.

Še boljši postopek dobimo, če takoj za stavek xPrvotni := x dodamo še x := x **and not** (x **div** 2) (tako dopolnjeni različici podprograma Zaokrozi2 recimo Zao-

krozi2a). To namreč pusti od vsake skupine zaporednih enic goreti le najvišjo:

$$\begin{aligned} x &= \ldots 00\,11111\,00000\,11111\,00000\,11111 \\ x \textbf{ div } 2 &= \ldots 00\,01111\,10000\,01111\,10000\,01111 \\ \textbf{not } (x \textbf{ div } 2) &= \ldots 11\,10000\,01111\,10000\,01111\,10000 \\ x \textbf{ and not } (x \textbf{ div } 2) &= \ldots 00\,10000\,00000\,10000\,00000\,10000. \end{aligned}$$

Tako lahko precej zmanjšamo število prižganih bitov, preden se začne izvajati zanka **while**. Ker mora priti za vsako enico, razen na najnižjem bitu, še vsaj ena ničla, je lahko prižganih največ $\lfloor k/2 \rfloor + 1$ bitov.

Koliko smo s temi izboljšavami pridobili v primerjavi s prvotno rešitvijo? Lahko poskusimo prešteti, koliko iteracij zanke **while** se vsega skupaj izvede, če poženemo zgornje podprograme na vseh številih x iz neke množice. Z nekaj telovadbe pridemo do rezultatov iz spodnje tabele. Vidimo lahko, da porabi Zaokrozi2a približno pol manj iteracij kot Zaokrozi2, ta pa pol manj kot Zaokrozi1.

	Zaokrozi1	Zaokrozi2	Zaokrozi2a
Skupno število iteracij na vseh n-bitnih številih $(2^{n-1}, \ldots, 2^n - 1)$			
	$n2^{n-1} - 1$	$(n+1)2^{n-2}$	$(n+2)2^{n-3}$ (izjema: 1 pri $n = 1$)
OEIS	A001787	A001792	A045623
Skupno število iteracij na vseh številih z največ n biti $(1, \ldots, 2^n - 1)$			
	$(n-1)(2^n - 1)$	$n2^{n-1}$	$(n+1)2^{n-2}$
OEIS	A059672	A001787	A001792

Namesto z ugašanjem bitov lahko rešimo nalogo tudi s prižiganjem. Recimo, da za začetno vrednost x velja $2^k \le x < 2^{k+1}$. Najvišji prižgani bit v dvojiškem zapisu števila x je torej bit k. Če bi prižgali še vse nižje ležeče bite, bi dobili vrednost $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1 = 2g(x) - 1$. Tako ni težko priti do $g(x)$, od tam pa do $f(x)$ z enakim razmislekom kot zgoraj. Za učinkovito prižiganje bitov si pomagajmo z zamikanjem: če je v x že prižganih r najvišjih bitov (torej od $k - r + 1$ do k), bodo v x **shr** r prižgani biti od $k - 2r + 1$ do $k - r$; če števili x in x **shr** r združimo z operatorjem **or**, bodo prižgani vsi biti od $k - 2r + 1$ do k, torej že $2r$ najvišjih bitov. Število zagotovo prižganih bitov se torej v vsakem koraku podvoji, zato bo treba to izvesti največ petkrat, če delamo z 32-bitnimi celimi števili.

```
function Zaokrozi3(x: integer): integer;
begin
  x := x − 1;
  x := x or (x shr 1);
  x := x or (x shr 2);
  x := x or (x shr 4);
  x := x or (x shr 8);
  x := x or (x shr 16);
```

```
  Zaokrozi3 := x + 1;
end; {Zaokrozi3}
```

Če pa ne bi vedeli, koliko bitov imajo lahko števila tipa integer, bi lahko naredili zanko in se ustavili, čim bi bili v x prižgani že vsi biti. To prepoznamo tako, da primerjamo x in $x+1$; če so v x prižgani vsi biti, so v $x+1$ vsi ti biti ugasnjeni, naslednji bit pa je prižgan. Primer:

$$\begin{aligned} x &= 111111 \\ x+1 &= 1000000 \\ x \text{ and } (x+1) &= 0. \end{aligned}$$

```
function Zaokrozi3a(x: integer): integer;
var r: integer;
begin
  x := x - 1; r := 1;
  while x and (x + 1) <> 0 do begin
    x := x or (x shr r);
    r := r shl 1;
  end; {while}
  Zaokrozi3a := x + 1;
end; {Zaokrozi3a}
```

Lepo pri tej različici rešitve je tudi to, da izvede le toliko iteracij, kot je potrebno — na primer, če je $x = 85 = 1010101_2$, bomo že po prvi iteraciji dobili 1111111_2 in se takoj ustavili. V splošnem se izkaže, da potrebujemo $1+\lfloor \log_2 k \rfloor$ iteracij, če je k dolžina najdaljše strnjene skupine ničel v dvojiškem zapisu števila x. Povprečna vrednost k po vseh n-bitnih številih (torej $2^{n-1} \le x < 2^n$) je približno $\log_2 n - 0{,}67$, povprečno število iteracij po vseh n-bitnih številih pa narašča še malo počasneje kot $O(\log \log n)$. Na primer: pri 30-bitnih številih porabi Zaokrozi2a povprečno 8 iteracij, Zaokrozi3a pa 2,67; pri 64-bitnih številih porabi Zaokrozi2a povprečno 16,5 iteracij, Zaokrozi3a pa le 3.[74]

Nalogo lahko rešimo tudi z uporabo števil s plavajočo vejico. Če je $2^{k-1} < x \le 2^k$, bo za dvojiški logaritem veljalo $k - 1 < \log_2 x \le k$, torej $k = \lceil \log_2 x \rceil$. Ko to izračunamo, moramo vrniti vrednost 2^k. Ker običajno nimamo na razpolago posebne funkcije za računanje dvojiškega logaritma, bomo uporabili naravne algoritme in formulo $\log_2 x = (\ln x)/(\ln 2)$.

```
function Zaokrozi4(x: integer): integer;
var y: integer;
```

[74] Pri 2048-bitnih številih porabi Zaokrozi2a povprečno 512,5 iteracij, Zaokrozi3a pa le 3,998! Žal pa v tem primeru naša analiza tako ali tako ni več pretirano realistična, ker samo šteje iteracije glavne zanke in zanemarja dejstvo, da pri delu z zelo velikimi n-bitnimi števili mnoge operacije (npr. **or** in **and**) ne trajajo konstantno mnogo časa, pač pa $O(n)$ časa. Zato postopek, kot je Zaokrozi3a, takrat ne bi bil posebej primeren.

```
begin
  if x <= 0 then Zaokrozi4 := 0
  else begin
    { V pascalu žal nimamo funkcije za zaokrožanje navzgor,
      imamo pa Trunc, ki zaokroža proti 0. }
    y := 1 shl Trunc(Ln(x) / Ln(2));
    if y < x then y := 2 * y;
    Zaokrozi4 := y;
  end; {if}
end; {Zaokrozi4}
```

Slabost tega postopka je, da je računanje naravnega logaritma pogosto precej počasno in je zato celoten postopek počasnejši od prej omenjenih rešitev.

Recimo, da v našem narečju pascala obstaja tip double, ki hrani 64-bitna števila s plavajočo vejico, predstavljena po standardu IEEE 754. Takšno število je sestavljeno iz treh delov: mantise (spodnjih 52 bitov), eksponenta e (naslednjih 11 bitov) in predznaka s (najvišji bit). Če odmislimo posebne primere, kot so neskončnosti, vrednost NaN in denormalizirana števila, je vrednost takega števila (recimo ji x) naslednja: pred 52-bitno mantiso v mislih postavimo enico in „decimalno“ vejico ter dobljeno število preberimo kot neko (mogoče ne-celo) število y, zapisano v dvojiškem sistemu. Nato ga še pomnožimo z 2^{e-1023}; če je predznak $s = 1$, ga pomnožimo še z -1 (pri naši nalogi bomo delali le z nenegativnimi števili in lahko na predznak pozabimo). Ker je $1 \le y < 2$, je $2^{e-1023} \le x < 2^{e-1022}$, torej je $\lfloor \log_2 x \rfloor = e - 1023$. Tako nam logaritma ne bo treba računati s funkcijo Ln, ampak lahko kar izluščimo vrednost e iz števila tipa double. Predpostavili bomo še, da imamo na voljo 64-bitni celoštevilski tip int64. Na spremenljivko tipa double lahko pogledamo, kot da je tipa int64, in jo zamaknemo za 52 bitov v desno („**shr** 52“) ter izključimo vse bite razen spodnjih 11 („**and** 2047“); kar ostane, je ravno iskani eksponent.

```
function Zaokrozi4a(x: integer): integer;
var xx: double; y: integer;
begin
  if x <= 0 then Zaokrozi4a := 0
  else begin
    xx := x;
    y := 1 shl (((int64(xx) shr 52) and 2047) - 1023);
    if y < x then y := 2 * y;
    Zaokrozi4a := y;
  end; {if}
end; {Zaokrozi4a}
```

Tip double smo uporabili zato, ker ima dovolj veliko mantiso, da lahko brez napake hrani poljubno 32-bitno celo število. Če nas zanimajo le manjša števila, bi bil dovolj dober že tip single, ki ima 23-bitno mantiso.

Nalogo lahko rešimo tudi z bisekcijo. Našli bi radi vrednost k, za katero je $2^k \le x < 2^{k+1}$. Med iskanjem vzdržujmo dve števili, k_L in k_D, tako da je $2^{k_L} \le x < 2^{k_D}$; v vsaki iteraciji zanke bomo eno od teh dveh števil spremenili in to tako, da bo ta pogoj še naprej veljal, razlika $k_D - k_L$ pa se bo razpolovila (od tod ime „bisekcija“ — razdeljevanje na dvoje). Tako že v nekaj korakih pridemo do $k_D - k_L = 1$; natančneje povedano, če delamo z največ n-bitnimi števili, postavimo na začetku $k_L = 0$, $k_D = n$ in ker se razdalja med njima v vsaki iteraciji prepolovi, bomo izvedli največ $\lceil \lg n \rceil$ iteracij. Pri 32-bitnih številih pomeni to pet iteracij, pri 64-bitnih pa šest.

```
function Zaokrozi5(x: integer): integer;
var k1, k2, k: integer;
begin
  if x <= 0 then begin Zaokrozi6a := 0; exitend;
  k1 := 0; k2 := 32;
  while k2 - k1 > 1 do begin
    { Na tem mestu velja: 2^k1 ≤ x < 2^k2. }
    k := (k1 + k2) div 2;
    if x < 1 shl k then k2 := k else k1 := k;
  end; {while}
  { Na tem mestu velja: 2^k1 ≤ x < 2^(k1+1). }
  k := 1 shl k1;
  if x > k then k := k * 2;
  Zaokrozi5 := k;
end; {Zaokrozi5}
```

Zanko lahko tudi „razvijemo“ v skupino drevesasto gnezdenih pogojnih stavkov. Spodaj je različica, ki predpostavlja, da je vhodni parameter x neko osembitno število, torej $x \le 256$; če bi hoteli podpreti vsa števila do 2^{16}, bi bil podprogram dvakrat daljši, za vsa 32-bitna števila pa bi bil štirikrat daljši. Lepo pri tej rešitvi je, da se ni več treba ukvarjati s spremenljivkami k, k1 in k2, zato bo podprogram malo hitrejši.

```
function Zaokrozi5a(x: integer): integer;
begin
  if x <= 16 then
    if x <= 4 then
      if x <= 2 then
        if x <= 0 then Zaokrozi5a := 0
        else Zaokrozi5a := x
      else Zaokrozi5a := 4
    else
      if x <= 8 then Zaokrozi5a := 8
      else Zaokrozi5a := 16
  else
    if x <= 64 then
```

```
      if x <= 32 then Zaokrozi5a := 32
      else Zaokrozi5a := 64
    else
      if x <= 128 then Zaokrozi5a := 128
      else Zaokrozi5a := 256;
end; {Zaokrozi5a}
```

Recimo, da nas zanimajo le števila z največ n biti, torej $0 \le x < 2^n$, in da se pojavljajo vsa enako pogosto. Med temi števili je kar polovica n-bitnih, četrtina je $(n-1)$-bitnih in tako naprej; velika števila so torej pogostejša in če hočemo rešitev s čim manjšo povprečno časovno zahtevnostjo, je pametno narediti takšno, ki teče na velikih številih čim hitreje, četudi je na majhnih zato mogoče malo počasnejša. Uporabimo lahko na primer postopek, ki dela podobno kot Zaokrozi1, le da pregleduje potence števila 2 od večjih proti manjšim. Če je $x > 2^{n-1}$, mora naša funkcija vrniti 2^n; če je $x > 2^{n-2}$, mora vrniti 2^{n-1} in tako naprej.

```
function Zaokrozi6(x: integer): integer;
var r: integer;
begin
  if x <= 0 then begin Zaokrozi6 := 0; exit end;
  r := 1 shl (n - 1);
  while r >= x do r := r shr 1;
  if x = 1 then Zaokrozi8 := 1
  else Zaokrozi8 := r * 2;
end; {Zaokrozi8}
```

Če poženemo ta podprogram na vseh x od 0 do $2^n - 1$, se izvede vsega skupaj $2^n - 1$ iteracij zanke **while** — torej v povprečju manj kot ena na vsak x! Če se torej v povprečju res pojavljajo vsi x enako pogosto, bo ta rešitev zelo hitra. Največ časa pa porabi pri majhnih x, zato je manj ugodna, če jo mislimo uporabljati v razmerah, ko bodo prevladovali majhni x, možni pa bodo tudi veliki x, tako da pri inicializaciji spremenljivke r ne smemo uporabiti kakšnega majhnega n. Še hitrejšo različico Zaokrozi6 dobimo, če zanko razvijemo v zaporedje stavkov **if**. Spodaj je različica, ki deluje pravilno za števila do 256; če bi hoteli podpirati tudi večje x, pa bi morali pač dodati na začetku še nekaj takih pogojnih stavkov.

```
function Zaokrozi6a(x: integer): integer;
begin
  if x > 128 then Zaokrozi6a := 256
  else if x > 64 then Zaokrozi6a := 128
  else if x > 32 then Zaokrozi6a := 64
  else if x > 16 then Zaokrozi6a := 32
  else if x > 8 then Zaokrozi6a := 16
```

```
  else if x > 4 then Zaokrozi6a := 8
  else if x > 2 then Zaokrozi6a := 4
  else if x > 1 then Zaokrozi6a := 2
  else if x > 0 then Zaokrozi6a := 1
  else Zaokrozi6a := 0;
end; {Zaokrozi6a}
```

Težave s počasnostjo pri majhnih x bi lahko omilili tako, da bi rezultate zanje hranili kar v neki tabeli (ki bi jo inicializirali ob zagonu programa).

Primerjavo hitrosti delovanja različnih tu opisanih rešitev kaže spodnja tabela.

		Povprečni čas izvajanja [ns] za $x = 1, 2, \ldots, 2^n$			
Funkcija	Opis	$n = 20$	24	27	30
Zaokrozi1	primerja x z 1, 2, 4, 8, ...	116	136	152	167
Zaokrozi2	ugaša prižgane bite	70	80	87	95
Zaokrozi2a	ugaša skupine prižganih bitov	52	57	61	65
Zaokrozi3	prižiga bite (5 korakov)	39	39	39	39
Zaokrozi3a	prižiga bite (zanka)	35	36	37	37
Zaokrozi4	uporabi Ln	219	219	219	250
Zaokrozi4a	uporabi eksponent v tipu double	60	60	60	60
Zaokrozi5	bisekcija po eksponentih	78	78	75	73
Zaokrozi5a	bisekcija, razvita v stavke **if**	15	14	14	14
Zaokrozi6	primerja x z 2^{n-1}, 2^{n-2}, 2^{n-3}, ...	19	17	17	17
Zaokrozi6a	kot Zaokrozi6 z razvito zanko	13	12	12	12

Vsako od rešitev naloge 2000.1.2 smo pognali na vseh x od 1 do 2^n in skupni procesorjev čas delili z 2^n. Tabela prikazuje dobljene povprečne čase v nanosekundah.

R2000.1.3

Pri tej nalogi ni treba drugega kot slediti navodilom. Ko beremo ocene, jih sproti seštevamo (spremenljivka Vsota), v spremenljivki Negativen pa si zapomnimo, če smo naleteli na kakšno negativno oceno. Na koncu povprečno oceno izračunamo tako, da vsoto delimo z deset.

N: 396

```
program Spricevalo(Input, Output);
var j, Ocena, Vsota: integer;
    Negativen: boolean;
begin
  Negativen := false; Vsota := 0;
  for j := 1 to 10 do begin
    ReadLn(Ocena);
    Vsota := Vsota + Ocena;
    if Ocena = 1 then Negativen := true;
  end; {for}
```

```
  if Negativen then
    WriteLn('Uspeh je negativen.')
  else
    WriteLn('Uspeh je pozitiven. Povprečna ocena je ', Vsota/10:2:1);
end. {Spricevalo}
```

N: 397

R2000.1.4

V prvem prehodu skozi vse podatke si le zapomnimo dolžino najdaljšega imena izvajalca, naslova albuma in naslova skladbe, da bomo kasneje lahko podatke pri izpisu lepo poravnali v stolpce. V drugem prehodu beremo zapise enega za drugim in če ima nek zapis istega izvajalca in album kot prejšnji, moramo izpisati le naslov skladbe (pred tem pa dovolj presledkov); če ima istega izvajalca, a ne album, moramo izpisati album in skladbo; če pa ima tudi drugega izvajalca, izpišemo vse troje. V spremenljivkah PrejIzv in PrejAlbum si zapomnimo izvajalca in album iz prejšnjega zapisa, da ju lahko primerjamo s trenutnim.

```
program Izpisi(Input, Output);

var SirIzv, SirAlb: integer; { širine stolpcev }
    Izvajalec, Album, Skladba: string;
    PrejIzv, PrejAlbum: string;
begin
  SirIzv := 0; SirAlb := 0;
  ZacniSeznam;
  while Komad(Izvajalec, Album, Skladba) do begin
    if Length(Izvajalec) > SirIzv then SirIzv := Length(Izvajalec);
    if Length(Album) > SirAlb then SirAlb := Length(Album);
  end; {while}
  SirIzv := SirIzv + 2; SirAlb := SirAlb + 2); { po dva presledka med stolpci }
  PrejIzv := ''; { da bomo prvega izvajalca izpisali v celoti }
  ZacniSeznam;
  while Komad(Izvajalec, Album, Skladba) do begin
    if Izvajalec <> PrejIzv then begin { nov izvajalec }
      WriteLn(Izvajalec, ' ':(SirIzv - Length(Izvajalec)),
              Album, ' ':(SirAlb - Length(Album)), Skladba);
      PrejIzv := Izvajalec; PrejAlbum := Album;
    end else if Album <> PrejAlbum then begin { nov album }
      WriteLn(' ':SirIzv, Album, ' ':(SirAlb - Length(Album)), Skladba);
      PrejAlbum := Album;
    end else begin
      WriteLn(' ':SirIzv, ' ':SirAlb, Skladba);
    end; {if}
  end; {while}
end. {Izpisi}
```

REŠITVE NALOG ZA DRUGO SKUPINO

R2000.2.1

N: 397

V neki globalni spremenljivki si zapomnimo odmik (v urah) ob prejšnjem klicu. Ob vsaki polni uri (torej ko je Minuta = 0) primerjajmo prejšnji odmik s sedanjim, pa bomo zaznali primere, ko se ista ura še enkrat ponavlja ali pa je bila ena ura izpuščena (to si zapomnimo v spremenljivkah IzpustiliUro in PonavljamoUro). Opravila, ki se izvajajo na vsako uro, moramo v tem primeru izvajati kot običajno; v primeru ponavljanja ene ure tistih, ki se morajo izvesti na to uro, zdaj ne smemo izvesti še drugič; v primeru izpuščene ure moramo izvesti tudi tista opravila, ki bi se morala izvesti v izpuščeni prejšnji uri. Ob naslednji polni uri se bosta IzpustiliUro in PonavljamoUro postavili nazaj na false in program bo spet deloval po starem. Glede seznama opravil bomo predpostavili, da ga lahko beremo kar s standardnega vhoda.

```
type
  NizT = packed array [1..64] of char;
var
  PrejsnjiOdmikDefiniran: integer value 0;
  PrejsnjiOdmik: integer;
  IzpustiliUro, PonavljamoUro: boolean;

  procedure Opravi(Opravilo: NizT); external;

procedure PolnaMinuta(UraUTC, Minuta, Odmik: integer);
var
  h, m: integer;
  Opravilo: NizT;
  Stori: boolean;
begin
  if PrejsnjiOdmikDefiniran = 0 then
    begin PrejsnjiOdmik := Odmik; PrejsnjiOdmikDefiniran := 1 end;
  if Minuta = 0 then begin
    IzpustiliUro := Odmik > PrejsnjiOdmik;
    PonavljamoUro := Odmik < PrejsnjiOdmik;
    PrejsnjiOdmik := Odmik;
  end; {if}
  while not Eof(Input) do begin
    ReadLn(h, m, Opravilo);
    if (m < 0) or (m = Minuta) then begin
      if PonavljamoUro then  { prehod na zimski čas }
        Stori := (h < 0)
      else if IzpustiliUro then { prehod na poletni čas }
        Stori := (h < 0) or (h = UraUTC + Odmik) or (h = UraUTC + Odmik - 1)
      else
```

```
        Stori := (h < 0) or (h = UraUTC + Odmik);
      if Stori then Opravi(Opravilo);
    end; {if}
  end; {while}
end; {PolnaMinuta}
```

N: 399

R2000.2.2

Presek pravokotnika s spodnjim levim ogliščem (x_1, y_1) in zgornjim desnim (x_2, y_2) ter pravokotnika z ogliščema (x'_1, y'_1) in (x'_2, y'_2) ima oglišči

$$(\max\{x_1, x'_1\}, \max\{y_1, y'_1\}) \quad \text{in} \quad (\min\{x_2, x'_2\}, \min\{y_2, y'_2\}).$$

Potem lahko izračunamo presek med njim in nekim tretjim pravokotnikom in imamo zdaj presek vseh treh. Tako nadaljujemo, pri tem pa lahko še sproti preverjamo, če je presek postal prazen (če desni rob ne leži več desno od levega ali pa zgornji ne več nad spodnjim).

```
const N = ...;                  { število kvadratov }
type Kvadrat = record           { vsak kvadrat je podan z dvema točkama: }
       x1, y1, x2, y2: integer; { spodnjim levim in zgornjim desnim ogliščem }
     end;
var Kvadrati: Kvadrat[1..N];

function PresekObstaja: boolean;
var Presek: Kvadrat; i: integer;
begin
  PresekObstaja := False;
  Presek := Kvadrati[1];
  for i := 2 to N do begin
    Presek.x1 := max(Presek.x1, Kvadrati[i].x1);
    Presek.y1 := max(Presek.y1, Kvadrati[i].y1);
    Presek.x2 := min(Presek.x1, Kvadrati[i].x1);
    Presek.y2 := min(Presek.y1, Kvadrati[i].y1);
    if (Presek.x1 > Presek.x2) or (Presek.y1 > Presek.y2) then
      exit; { presek ne obstaja oz. je prazna množica }
  end; {for}
  PresekObstaja := true;
end; {PresekObstaja}
```

N: 399

R2000.2.3

Vhodno besedilo beremo znak po znak; ko naletimo na `<` (ki mu mogoče sledi še `/`), preberemo ime oznake in nato preskočimo atribute (do znaka `>`). Če je oznaka na seznamu prepovedanih, ne izpišemo nič, sicer pa samo ime oznake, brez atributov. Besedilo med oznakami izpisujemo nespremenjeno.

```
const N = 10;
var Prepovedana: array [1..N] of string;
```

```
var C: char; S: string; EndTag, Ok, Gt: boolean; i: integer;
begin
  while not Eof(Input) do begin
    Read(C);
    if C <> '<' then begin Write(C); continue end;
    { Preberimo ime oznake. }
    S := ''; EndTag := false; Gt := false;
    while not Eof(Input) do begin
      Read(C);
      if C = '/' then EndTag := true { To je oznaka za konec elementa. }
      else if not (C in ['A'..'Z', 'a'..'z', '0'..'9']) then
        begin Gt := C = '>'; break end { Konec imena oznake. }
      else S := S + C;
    end; {while}
    { Berimo do konca oznake (do znaka '>'), da preskočimo atribute.
      Mogoče smo znak '>' celo že prebrali — to pove spremenljivka Gt. }
    if not Gt then while not Eof(Input) do
      begin Read(C); if C = '>' then break end;
    { Preverimo, če je oznaka prepovedana. }
    Ok := true; i := 1;
    while (i <= N) and Ok do
      begin Ok := UpCase(S) <> UpCase(Prepovedana[i]); i := i + 1 end;
    if Ok then begin { Izpišimo oznako. }
      Write('<');
      if EndTag then Write('/');
      Write(S, '>');
    end; {if}
  end; {while}
end.
```

Funkcija UpCase naj bi vrnila niz, v katerem je vsaka mala črka zamenjana z ustrezno veliko. Pri prevajalnikih, ki take funkcije nimajo (ali pa podpira le posamične znake, ne pa nizov), bi si pač lahko pomagali s svojim podprogramom.

R2000.2.4

N: 400

Predpostavili bomo, da je koda reprezentance neko majhno število, ki ga lahko uporabimo kot indeks v tabelo (drugače bi si lahko pomagali z razpršeno tabelo), in da se v razvrstitvi leta 1999 ne pojavlja nobena taka reprezentanca, ki se ne bi že leta 1998. Imeli bomo tabelo imen in položajev v lanski razvrstitvi. Ko beremo letošnjo razvrstitev, prek te tabele poiščemo lanski položaj, izračunamo premik na lestvici in sproti vzdržujemo podatek o tem, katera od doslej prebranih je naredila največji skok (Najboljsa) in kolikšen je ta skok bil (NajvecjiSkok). Na koncu le še izpišemo rezultat.

```
Uvrstitev := 0;
```

```
while not Eof(Stanje98) do begin
  Uvrstitev := Uvrstitev + 1;
  PreberiPodatek(Stanje98, ImeReprezentance, KodaReprezentance);
  Ime[KodaReprezentance] := ImeReprezentance;
  Skok[KodaReprezentance] := Uvrstitev;
end; {while}
Uvrstitev := 0;
NajvecjiSkok := -1;
while not Eof(Stanje99) do begin
  Uvrstitev := Uvrstitev + 1;
  PreberiPodatek(Stanje99, ImeReprezentance, KodaReprezentance);
  Skok[KodaReprezentance] := Skok[KodaReprezentance] - Uvrstitev;
  if Skok[KodaReprezentance] > NajvecjiSkok then begin
    NajvecjiSkok := Skok[KodaReprezentance];
    Najboljsa := KodaReprezentance;
  end; {if}
end; {while}
WriteLn('Najboljša je reprezentanca ', Ime[Najboljsa],
        ', ki je naredila skok za ', Skok[Najboljsa], ' mest.');
```

REŠITVE NALOG ZA TRETJO SKUPINO

N: 401

R2000.3.1

Opazujmo orientacijo sten. Dve steni (obe vodoravni ali navpični) morata imeti orientacijo v smeri urinega kazalca; torej, če je neka navpična stena levo od neke druge, mora leva kazati navzgor, desna pa navzdol; podobno velja tudi za vodoravne stene. Ker so točke podane v smeri urinega kazalca, smeri stene ni težko določiti.

nerešljiv rešljiv

Spodnji podprogram zaradi lažjega dostopa do tabele predpostavlja, da je v px[0] in py[0] še ena kopija koordinat px[n] in py[n]. Ker je vsaka druga stena navpična, vse vmes pa vodoravne, je stena, ki se začne pri točki $i \bmod 2$, gotovo vzporedna tisti, ki se začne pri i, tako da lahko gre j od i **mod** 2 naprej s korakom 2.

```
const n = ...;
var px, py: array [0..n] of integer;

function LahkoPostavimoKamero: boolean;
var i, j: integer;
begin
  LahkoPostavimoKamero := false;
```

```
  for i := 0 to n − 1 do begin
    j := i mod 2; while j < n do begin
      if px[i] = px[i + 1] then begin  { navpično }
        if (px[i] < px[j]) and (py[i] > py[i + 1]) and (py[j] < py[j + 1]) then exit;
      end else                         { vodoravno }
        if (py[i] > py[j]) and (px[i] > px[i + 1]) and (px[j] < px[j + 1]) then exit;
      j := j + 2;
    end; {while}
  end; {for}
  LahkoPostavimoKamero := true;
end; {LahkoPostavimoKamero}
```

Ta rešitev je časovno precej zahtevna, saj bi pri n stenah kar $(n^2/2)$-krat primerjala orientacijo dveh sten (vsako vodoravno z vsemi $n/2$ vodoravnimi in vsako navpično z vsemi $n/2$ navpičnimi). Obstaja pa tudi učinkovitejši postopek. Označimo položaj kamere s koordinatama (x_k, y_k). Vsaka stena prestavlja neko omejitev glede tega, kje sme biti kamera, da bo to steno videla. Če obstaja neka vodoravna stena od (x_1, y) do (x_2, y) in je $x_1 < x_2$, mora biti kamera pod to steno, da jo bo videla z notranje strani; imamo torej pogoj $y_k < y$, ki je obenem potreben in zadosten. Podobno, če je $x_1 > x_2$, dobimo pogoj $y_k > y$. Za navpične stene od (x, y_1) do (x, y_2) pa pri $y_1 < y_2$ dobimo pogoj $x_k > x$ in pri $y_1 > y_2$ pogoj $x_k < x$. Kamero lahko postavimo na poljubno točko (x_k, y_k), ki ustreza vsem dobljenim pogojem; moramo torej le preveriti, ali sploh obstaja kakšna taka točka.

```
const n = ...;
var px, py: array [1..n] of integer;

function LahkoPostavimoKamero: boolean;
var i, j, xMin, yMin, xMax, yMax: integer;
begin
  { Na začetku vzemimo, da je sprejemljiv vsak položaj kamere
    v očrtanem pravokotniku (bounding box) cele sobe. }
  xMin := px[1]; xMax := px[1]; yMin := py[1]; yMax := py[1];
  for i := 2 to n do begin
    if px[i] < xMin then xMin := px[i];
    if px[i] > xMax then xMax := px[i];
    if py[i] < yMin then yMin := py[i];
    if py[i] > yMax then yMax := py[i];
  end; {for}
  { Upoštevajmo omejitve, ki jih nalagajo posamezne stene. }
  j := n;
  for i := 1 to n do begin          { Oglejmo si steno od j do i. }
    if px[i] = px[j] then begin     { navpična }
      if (py[i] > py[j]) and (px[i] > xMin) then xMin := px[i];
      if (py[i] < py[j]) and (px[i] < xMax) then xMax := px[i];
```

```
      end else begin                    { vodoravna }
        if (px[i] < px[j]) and (py[i] > yMin) then yMin := py[i];
        if (px[i] > px[j]) and (py[i] < yMax) then yMax := py[i];
      end;
      j := i;
    end; {for}
    { Sprejemljivi položaji kamere so vsi, ki ustrezajo pogojema
      xMin < xKamere < xMax in yMin < yKamere < yMax.
      Torej mora biti xMin < xMax in yMin < yMax. }
    LahkoPostavimoKamero := (xMin < xMax) and (yMin < yMax);
end; {LahkoPostavimoKamero}
```

N: 401

R2000.3.2

Mislimo si graf, kjer vsakemu prostemu polju labirinta pripada po ena točka in sta dve točki povezani natanko tedaj, ko imata njuni polji skupno stranico. Ker je v labirintu med vsakim parom polj natanko ena pot, je ta graf pravzaprav drevo. Eno od polj (vseeno je, katero) si izberimo za koren drevesa. Čim imamo koren, lahko v drevo vpeljemo relacije tipa starši–potomci: od korena do poljubne druge točke u obstaja natanko ena pot in u-jev oče je zadnja točka pred u na tej poti. Ko imamo v drevesu takšno strukturo, lahko govorimo tudi o poddrevesih, ta pa nam omogočajo, kot bomo videli, precej elegantno določiti najdaljšo pot v drevesu.

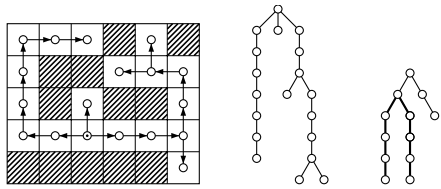
Za koren lahko vzamemo na primer kar prvo prosto polje v matriki Labirint. Preostanek drevesa lahko potem preiščemo z iskanjem v globino. To je eden od znanih postopkov za sistematično pregledovanje grafov. Na začetku si mislimo, da je vsaka točka še nepregledana. Poženimo naslednji podprogram in mu kot parameter podajmo koren drevesa:

```
procedure IskanjeVGlobino(a: Tocka);
begin
  Pregledana[a] := true;
  za vsako a-jevo sosedo b:
    if not Pregledana[b] then
      { Točka a je roditelj točke b. Preglejmo zdaj poddrevo, ki se prične pri b. }
      IskanjeVGlobino(b);
end; {IskanjeVGlobino}
```

Ko se torej ta postopek pokliče za neko točko a, se bo zakopal v a-jeva poddrevesa in se ne bo vrnil, dokler ne bo rekurzivno pregledal vsega, kar se v drevesu nahaja pod točko a, pa ne glede na to, kako globoko bo treba iti. Odtod tudi izraz „iskanje v globino“.

Ker naloga zagotavlja, da naši labirinti ne vsebujejo cikličnih poti, je med a-jevimi sosedami že pregledana le tista, iz katere smo prišli v a. Zato ni nujno imeti posebne tabele Pregledana, ampak si lahko zgolj zapomnimo, od kod smo prišli v trenutno točko.

Primer labirinta in drevesa, ki ga odkrijemo v njem, kažeta sliki (1*a*) in (1*b*) spodaj.



(1*a*) (1*b*) (2)

(1*a*) Primer odkrivanja drevesa v labirintu. Polje, pri katerem smo začeli, je označeno z ⊙. (1*b*) Dobljeno drevo, če bi ga narisali na bolj tradicionalen način. (2) Primer drevesa, pri katerem najdaljša pot (označena z debelimi črtami) ne poteka skozi koren drevesa.

Zdaj lahko najbolj oddaljeni točki v drevesu poiščemo tako, da začnemo pri listih drevesa in napredujemo proti korenu, pri tem pa upoštevamo naslednji rekurzivni razmislek: najdaljša pot v drevesu lahko koren drevesa obišče ali pa ne. (*i*) Če pot gre skozi koren, jo koren razdeli na dva kosa, vsak od njiju pa gre lahko le še navzdol po drevesu (saj iz korena ne moremo narediti koraka navzgor, torej gre prvi korak navzdol; koraku navzdol pa v drevesu ne more slediti korak navzgor, saj bi se tako le vrnili nazaj v točko, iz katere smo ravnokar prišli, takih poti pa naloga ne dovoli). Zato je najdaljša tovrstna pot tista, pri kateri se ta dva konca poti spustita v najgloblji poddrevesi. (*ii*) Če pa najdaljša pot ne gre skozi koren drevesa (glej npr. drevo (2) na sliki), pomeni, da v celoti leži znotraj enega od poddreves (kajti pot ne more iti iz enega poddrevesa v drugo, ne da bi šla pri tem skozi koren); torej mora biti to najdaljša pot tistega poddrevesa.

Najdaljšo pot bomo torej dobili tako, da bomo od opisanih dveh možnosti vedno vzeli tisto, ki nam da daljšo pot. Preden lahko določimo najdaljšo pot (in globino) nekega drevesa, moramo poznati najdaljše poti in globine njegovih poddreves. (Ta rekurzija se konča pri listih drevesa, torej takrat, ko trenutna točka sploh nima poddreves. Takrat je edina in najdaljša pot ta, ki obišče dani list in nič drugega.) Zato lahko računanje najdaljših poti in globin lepo vključimo v zgoraj opisani postopek iskanja v globino: primeren trenutek za izračun najdaljše poti in globine za poddrevo s korenom v točki a je takoj po tistem, ko se vrnemo iz rekurzivnih klicev za a-jeve otroke (in preden se vrnemo iz rekurzivnega klica za samo točko a); takrat imamo že pripravljene vse potrebne podatke o a-jevih poddrevesih.

```
program NajdaljsaPot;
const MaxVisina = ...; MaxSirina = ...;
      DX: array [1..4] of longint = (0, 0, −1, 1);
```

```
      DY: array [1..4] of longint = (1, −1, 0, 0);
type Polje = (Prosto, Zid);
var Labirint: array [1..MaxVisina, 1..MaxSirina] of Polje;
    Sirina, Visina: LongInt;

  { Pregleda poddrevo, ki se začne pri (X, Y); vrne globino poddrevesa in dolžino
    najdaljše poti v njem. (PX, PY) sta koordinati starša polja (X, Y) v drevesu. }
  procedure Drevo(X, Y, PX, PY: longint; var Glob, Dolz: longint);
  var Smer, XC, YC, GC, DC, Glob2: longint;
  begin
    Glob := 0; Glob2 := 0; Dolz := 0;
    for Smer := 1 to 4 do begin
      { Oglejmo si poddrevo, ki se začne pri (XC, YC). }
      XC := X + DX[Smer]; YC := Y + DY[Smer];
      if (XC = PX) and (YC = PY) then continue;
      if (XC < 1) or (XC > Sirina) or (YC < 1) or (YC > Visina) then continue;
      if Labirint[YC, XC] = Zid then continue;
      Drevo(XC, YC, X, Y, GC, DC);
      if GC > Glob then begin Glob2 := Glob; Glob := GC end
      else if GC > Glob2 then Glob2 := GC;
      if DC > Dolz then Dolz := DC;
    end; {for}
    { Zdaj imamo v Glob in Glob2 globini dveh najglobljih poddreves. }
    DC := Glob + Glob2 + 1; if DC > Dolz then Dolz := DC;
    Glob := Glob + 1; { Globina drevesa = globina najglobljega poddrevesa + 1. }
  end; {Drevo}

var X, Y, XR, YR, Glob, Dolz: longint; C: char;
begin
  { (XR, YR) bo koren drevesa. To je lahko poljubno prosto polje. }
  ReadLn(Sirina, Visina); XR := −1; YR := −1;
  for Y := 1 to Visina do begin
    for X := 1 to Sirina do begin
      Read(C);
      if C = '#' then Labirint[Y, X] := Zid
      else begin Labirint[Y, X] := Prosto; XR := X; YR := Y end;
    end; {for X}
    ReadLn;
  end; {for Y}
  Drevo(XR, YR, −1, −1, Glob, Dolz);
  WriteLn('Dolžina najdaljše poti: ', Dolz, '.');
end. {NajdaljsaPot}
```

Ta rešitev je lahko nekoliko potratna s pomnilnikom, če gredo rekurzivni klici zelo globoko. Načeloma bi lahko prosta polja tvorila eno samo dolgo kačasto pot, ki bi obsegala približno polovico vseh polj v labirintu. Rekurzivni klici podprograma Drevo se lahko tedaj gnezdijo do globine približno Sirina·Visina/2,

pri vsakem klicu pa se na sklad naložijo vsi parametri, lokalne spremenljivke in še kakšni knjigovodski podatki (npr. naslov za vrnitev iz klica). V takih primerih bi lahko prihranili precej pomnilnika, če bi rekurzivne klice simulirali sami; dovolj bi bilo, če bi za vsako celico hranili vrednosti Glob (globina poddrevesa, ki se začenja pri tej celici), Dolz (dolžina najdaljše poti v tem poddrevesu) in podatek o tem, katera od sosednjih točk je njen roditelj v drevesu (slednje bi lahko hranili kar v tabeli Labirint). Vendar pa je zdaj poraba pomnilnika za te dodatne podatke (poleg samega labirinta) vedno sorazmerna s številom vseh polj v labirintu (ker moramo pač rezervirati toliko pomnilnika za obe tabeli), medtem ko je bila prej sorazmerna z globino najglobljega drevesa (toliko je bilo namreč največ gnezdenih rekurzivnih klicev). Zato je ta prijem koristen le, če bomo imeli opravka z zelo izrojenimi drevesi. Na testnih podatkih z ACMovega tekmovanja (CERC 1999) je bil spodnji program nekaj počasnejši od gornjega (1,75 s namesto 1,2 s).

```
program NajdaljsaPot;
type PoljeSmer = (Prosto, Zid, Gor, Dol, Levo, Desno);
const MaxVisina = ...; MaxSirina = ...;
      DX: array [Gor..Desno] of longint = (0, 0, −1, 1);
      DY: array [Gor..Desno] of longint = (1, −1, 0, 0);
var Labirint: array [1..MaxVisina, 1..MaxSirina] of PoljeSmer;
    Glob, Dolz: array [1..MaxVisina, 1..MaxSirina] of longint;
    Sirina, Visina: longint;
    X, Y, XC, YC, XR, YR, Dolz1, DC, Glob1, Glob2, GC: longint;
    C: char; Smer: PoljeSmer; Nasel: boolean;
begin
    { Preberimo labirint. Za koren (XR, YR) izberimo poljubno prosto polje. }
    ReadLn(W, H); XR := −1; YR := −1;
    for Y := 1 to H do begin
     for X := 1 to W do begin
        Read(C);
        if C = '#' then Labirint[Y, X] := Zid
        else begin Labirint[Y, X] := Prosto; XR := X; YR := Y end;
     end; {for X}
     ReadLn;
    end; {for Y}
    if XR < 0 then begin WriteLn('Dolžina najdaljše poti: ', 0, '.'); exit end;
    Labirint[YR, XR] := Zid; X := XR; Y := YR; { Koren drevesa. }
    Smer := Pred(Gor);
    while true do begin
     { Poskusimo najti naslednjega otroka polja (X, Y). }
     Nasel := false;
     while Ord(Smer) < Ord(Desno) do begin
        Smer := Succ(Smer); XC := X + DX[Smer]; YC := Y + DY[Smer];
        if (XC <= 1) or (XC > W) or (YC <= 1) or (YC > H) then continue;
```

```
      if Labirint[YC, XC] <> Prosto then continue; { Zid ali pa naš roditelj. }
      { V Labirint[YC, XC] si zapomnimo, v kakšni smeri smo se premaknili. }
      X := XC; Y := YC; Labirint[Y, X] := Smer; Nasel := true; break;
    end; {while}
    if Nasel then begin Smer := Pred(Gor); continue end; { „rekurzivni klic“ }

    { Izračunajmo Glob in Dolz pri (X, Y) s pomočjo vrednosti pri otrocih. }
    Glob1 := 0; Glob2 := 0; Dolz1 := 0;
    for Smer := Gor to Desno do begin
      XC := X + DX[Smer]; YC := Y + DY[Smer];
      { Preverimo, če ni (XC, YC) slučajno naš roditelj. }
      if (X <> XR) or (Y <> YR) then if (XC = X - DX[Labirint[Y, X]])
        and (YC = Y - DY[Labirint[Y, X]]) then continue;
      if (XC <= 1) or (XC > W) or (YC <= 1) or (YC > H) then continue;
      if Labirint[YC, XC] = Zid then continue;
      GC := Glob[YC, XC]; DC := Dolz[YC, XC];
      if GC > Glob1 then begin Glob2 := Glob1; Glob1 := GC end
      else if GC > Glob2 then Glob2 := GC;
      if DC > Dolz1 then Dolz1 := DC;
    end; {for}
    DC := Glob1 + Glob2 + 1; if DC > Dolz1 then Dolz1 := DC;
    Dolz[Y, X] := Dolz1; Glob[Y, X] := Glob1 + 1;

    { Premaknimo se nazaj na starša in si zapomnimo tudi smer,
      od katere bo treba nadaljevati s pregledovanjem otrok. }
    if (X = XR) and (Y = YR) then break; { Preiskali smo že celo drevo. }
    Smer := Labirint[Y, X]; X := X - DX[Smer]; Y := Y - DY[Smer];
  end; {while}
  WriteLn('Dolžina najdaljše poti: ', Dolz[YR, XR], '.');
end. {NajdaljsaPot}
```

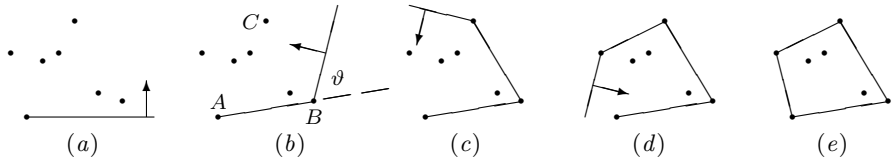
N: 402

R2000.3.3

Problem, ki ga rešujemo pri tej nalogi, je v literaturi o računski geometriji znan kot problem iskanja konveksne ovojnice (*convex hull*). Konveskna ovojnica neke množice točk je najmanjši tak konveksni lik (ali telo, če smo v več dimenzijah), ki vsebuje vse te točke. Lik je konveksen takrat, ko za vsak par točk, ki ju vsebuje, vsebuje tudi celo daljico med njima; bolj preprosto si lahko to predstavljamo tako, da si mislimo, da je lik vsepovsod izbočen (ali pa raven), nikjer pa vbočen. Pri mnogokotniku, torej liku z ravnimi stranicami, je ta pogoj enakovreden zahtevi, naj bo notranji kot pri vsakem oglišču manjši od 180°. Ograja, ki jo hočemo postaviti okoli sadovnjaka, bo morala iti ravno po robu konveksne ovojnice, da bo najkrajša.

Naloge se lahko lotimo na več načinov. Če poiščemo najnižje ležečo točko (recimo ji A), torej tisto z najmanjšo koordinato y, vemo, da bo gotovo ležala na robu ovojnice. Mislimo si, da bi na to točko privezali en konec elastične vrvice, nato pa bi šli z drugim koncem okoli sadovnjaka (glej sliko na str. 423); elastika bi se pri tem ovijala okoli dreves na robu ovojnice in na koncu, ko bi

prišli naokoli in privezali drugi konec za drevo, kjer smo začeli, bi tekla elastika ravno po robu celotne konveksne ovojnice.

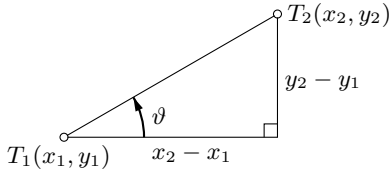


Vidimo, da bo naslednja točka na robu ovojnice (gledano v pozitivni smeri, torej nasproti smeri urinega kazalca) tista B, za katero bo kot med smerjo AB in vodoravno smerjo desno od A najmanjši. V nadaljevanju se elastika ovija okoli točke B in prva točka, ob katero zadene, je tista C, za katero je kot ϑ med staro smerjo AB in novo smerjo BC najmanjši. Potem ovijamo elastiko okoli C in tako naprej, dokler se ne vrnemo nazaj na začetek, v točko A.

```
type Tocka = record x, y: real end;
const N = ...;
var p: array [1..N + 1] of Tocka;

{ Vrne smerni kot daljice od t1 do t2 v stopinjah. }
function Theta(t1, t2: Tocka): real;
const eps = 1e−5; { Majhno pozitivno število. }
var dx, dy, Kot: real;
begin
  dx := t2.x − t1.x; dy := t2.y − t1.y;
  if Abs(dx) < eps then begin { navpičen vektor }
    if dy >= eps then Kot := Pi * 0.5 else Kot := Pi * 1.5
  end else begin
    Kot := ArcTan(dy / dx);
    if dx < 0 then Kot := Kot + Pi              { (−π/2, π/2) → (π/2, 3π/2) }
    else if dy < 0 then Kot := Kot + 2 * Pi;    { (−π/2, 0) → (3π/2, 2π) }
  end; {if}
  Theta := Kot * 180.0 / Pi; { Preračunajmo radiane v stopinje. }
end; {Theta}

procedure Ovojnica;
var i, Min, M: integer;
    MinKot, v: real;
    t: Tocka;
begin
  Min := 1;
  for i := 2 to N do
    if p[i].y < p[Min].y then Min := i;
  M := 0;
  p[N + 1] := p[Min];
```

```
  MinKot := 0.0;
  repeat
    { Na ovojnici že imamo točke p[1..M] v tem vrstnem redu,
      vemo pa tudi, da bo p[Min] naslednja. Torej jo premaknimo v p[M + 1]. }
    M := M + 1; t := p[M]; p[M] := p[Min]; p[Min] := t;
    Min := N + 1; v := MinKot; MinKot := 360.0;
    { Poglejmo zdaj, katera od preostalih točk, torej p[M..N + 1] (na mestu N + 1
      imamo kopijo prve točke z ovojnice, ker bomo morali ovojnico še speljati
      nazaj do nje), oklepa najmanjši kot s staro smerjo. Obenem vemo, da bo
      naklon nove stranice večji kot naklon prejšnje, ki je v stopinj. }
    for i := M + 1 to N + 1 do
      if Theta(p[M], p[i]) > v then
        if Theta(p[M], p[i]) < MinKot then
          begin Min := i; MinKot := Theta(p[m], p[Min]) end;
  until Min = N + 1;
end; {Ovojnica}
```

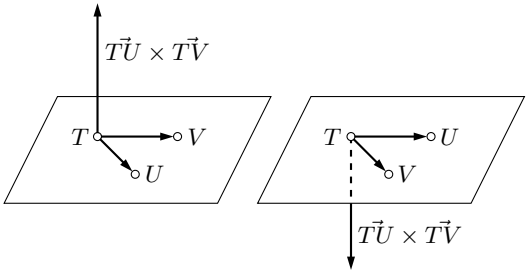
Funkcija Theta si pri računanju kotov pomaga s funkcijo arc tg (glej sliko), vendar pa mora posebej paziti na (skoraj) navpične vektorje (pri katerih bi lahko prišlo do deljenja z 0) in na primere, ko je sprememba x-koordinate negativna (ker arc tg vedno vrne vrednost z intervala $(-\pi/2, \pi/2)$). Če hočemo na koncu vračati kote z intervala $[0, 2\pi)$ namesto $(-\pi/2, 3\pi/2]$, moramo posebej paziti še na to.



Smerni kót daljice od T_1 do T_2 lahko izračunamo s pomočjo funkcije arc tg. Za pravokotni trikotnik na sliki velja $\operatorname{tg}\vartheta = \dfrac{y_2 - y_1}{x_2 - x_1}$, torej je $\vartheta = \operatorname{arc\,tg}\dfrac{y_2 - y_1}{x_2 - x_1}$.

Pogoja v notranji zanki (**for** i) bi lahko še malo poenostavili in se izognili potrebi po funkciji Theta, če bi uporabili vektorski produkt. Naj bo T prejšnja točka na ovojnici, U doslej najobetavnejša kandidatka za naslednjo točko, V pa še neka nova točka, za katero moramo zdaj preveriti, če ni mogoče še boljša od U. V bomo sprejeli kot novo najboljšo kandidatko, če je kot med smerjo prejšnje stranice in smerjo TV manjši kot med smerjo prejšnje stranice in TU; toda ta pogoj je enakovreden temu, da je TV desno od TU (oz. natančneje: če se hočemo iz smeri TU zasukati v smer TV in pri tem napraviti kot, manjši od 180°, se moramo zasukati nasproti smeri urinega kazalca). Recimo, da smer TU opisuje vektor $\vec{TU} = (x_1, y_1)$, smer TV pa $\vec{TV} = (x_2, y_2)$. Delajmo se, da sta to 3-d vektorja z z-koordinato 0 in izračunajmo njun vektorski produkt $\vec{TU} \times \vec{TV}$.

Spomnimo se, da je vektorski produkt definiran tako, da je pravokoten na $\vec{TU}$ in $\vec{TV}$ in da za njegovo smer velja pravilo desne roke: če pokažemo s palcem desne roke v smer $\vec{TU}$, s kazalcem pa v smer $\vec{TV}$, lahko pokažemo s sredincem v smer $\vec{TU} \times \vec{TV}$, v nasprotno smer pa ne (oz. težko).



Če imamo v neki ravnini podane tri točke, T, U in V, lahko iz smeri vektorskega produkta $\vec{TU} \times \vec{TV}$ ugotovimo, ali gre pri premiku iz smeri TU v smer TV za premik v levo ali v desno.

Torej, če je zasuk iz smeri TU v smer TV pozitiven, kaže vektorski produkt $\vec{TU} \times \vec{TV}$ nad ravnino, v kateri ležijo točke T, U in V, sicer pa pod njo. V našem primeru si lahko mislimo, da ležijo vse tri točke na ravnini $z = 0$, torej lahko to, ali kaže $\vec{TU} \times \vec{TV}$ nad ali pod njo, preverimo preprosto tako, da pogledamo predznak njegove z-koordinate. Iz običajne formule za vektorski produkt sledi, da je ta v našem primeru enak $(x_1, y_1, 0) \times (x_2, y_2, 0) = (0, 0, x_1 y_2 - x_2 y_1)$. Torej lahko pogoj, da mora biti smer TV desno od smeri TU, zamenjamo s pogojem $x_1 y_2 - x_2 y_1 < 0$. Tako nam ne bo treba računati s koti ali kakšnimi drugimi nezaželenimi operacijami s plavajočo vejico.

```
Min := M + 1;
for i := M + 2 to N + 1 do then
  if (p[i].x − p[M].x) * (p[Min].y − p[M].y) <
    (p[i].y − p[M].y) * (p[Min].x − p[M].x) then Min := i;
```

Pomembno je, da imamo v pogoju $<$ in ne $\leq$; drugače bi pri iskanju druge točke ovojnice (torej pri $M = 1$), ko bi i prišel do $N + 1$, računali vektorski produkt med vektorjem od p[M] do p[Min] ter vektorjem od p[M] do p[N + 1]; ker pa je $M = 1$ in je v p[N + 1] kopija začetne točke, je drugi od teh dveh vektorjev ničelni, zato bi dobili vektorski produkt 0 in pogoj $\leq$ bi bil zagotovo izpolnjen; tako bi naš program takoj speljal ovojnico nazaj v začetno točko in končal. Pogoj $<$ pa v takem primeru zanesljivo ni izpolnjen.

Slabost opisanega postopka (ki mu običajno pravijo „zavijanje darila“ ali paketa, pa tudi „Jarvisov obhod“) je, da mora preiskati vse točke (razen tistih, ki so že na robu ovojnice), da dobi naslednjo stranico ovojnice. Če ima ovojnica h stranic, je časovna zahtevnost tega postopka $O(nh)$; in v najslabšem primeru so lahko na robu ovojnice kar vse točke (velja $h = n$). Obstajajo tudi učinkovitejši postopki s časovno zahtevnostjo $O(n \lg n)$ in celo $O(n \lg h)$.[75]

[75]Literatura: kaj o računski geometriji, dovolj bo že ustrezno poglavje v Cormen *et al.*,

Preden začnemo računati konveksno ovojnico, lahko z naslednjim preprostim razmislekom zavržemo še nekaj točk, da se nam kasneje ne bo treba ukvarjati z njimi: če neki množici točk dodamo neko novo točko, ki leži znotraj konveksne ovojnice dosedanje množice, se konveksna ovojnica nič ne spremeni. Zato velja tudi obratno: če izberemo nekaj točk iz naše množice, lahko potem iz nje zavržemo vse, ki ležijo v njihovi konveksni ovojnici, pa se konveksna ovojnica cele množice ne bo zato nič spremenila. Lahko bi na primer izbrali najbolj levo, najnižjo, najbolj desno in najvišjo točko cele množice; konveksna ovojnica teh štirih je kar štirikotnik, ki jih obišče vse štiri v navedenem vrstnem redu (lahko je tudi izrojen v trikotnik ali celo daljico, če tiste štiri točke niso vse različne); zdaj ni težko za vsako od preostalih točk preveriti, če slučajno leži v tem štirikotniku, in če je to res, jo lahko zavržemo. Seveda pa ni nujno, da se bomo na ta način res znebili kakšne točke (npr. če so bile razporejene po neki krožnici).

N: 402

R2000.3.4

Če bi naloga zahtevala, da je *število* odposlanih podatkov za posamezno cev enako, bi bila pravilna rešitev preprosta vrsta (*queue*, FIFO). Ker pa naloga zahteva, da je enaka *količina* odposlanih podatkov, je prava rešitev *prioritetna vrsta* (*priority queue*). Ker lahko uporabimo poljubno rešitev prioritetne vrste iz literature, bomo v tej rešitvi predpostavili, da le-ta že obstaja in z njo tudi operacije PripraviPV(PV), VstaviPV(PV; Prioriteta; Paket) in IzlociNajmPV(PV; **var** Prioriteta; **var** Paket). Vloga prvih dveh operacij je očitna, medtem ko slednja izloči iz prioritetne vrste PV element z najmanjšo prioriteto. Pri tem predpostavljamo, da operacija vrne false, če ni v prioritetni vrsti PV nobenega elementa, in true, če je. V slednjem primeru je izločeni element paket Paket s prioriteto Prioriteta.

Prioriteta elementa (paketka) določa, kako zgodaj bo paketek zapustil usmerjevalnik; nižja ko je, prej bo odšel. Zato za vsako vhodno cev vodimo trenutno prioriteto, ki pove, koliko podatkov je doslej prišlo iz te cevi.

Takšna rešitev je pravilna, če iz vseh vhodnih cevi kar naprej prihajajo paketki. Ko pa iz neke cevi nekaj časa ni paketkov, bi bili naslednji iz nje prispeli paketki takoj odposlani, kar pa ni pravično — saj bi v tistem trenutku ta cev dobila večjo količino izhodne cevi od vseh ostalih. (Predstavljajmo si na primer, da imamo dve vhodni cevi in je en dan le ena pošiljala podatke, druga pa nič. Če hočeta naslednji dan pošiljati obe, mi pa bi gledali le količino poslanih podatkov, bi imela zdaj tista cev, ki včeraj ni poslala ničesar, danes ves dan vso izhodno pasovno širino zase, dokler ne bi poslala toliko podatkov, kolikor jih je prva cev poslala včeraj; šele potem bi začeli deliti izhodno pasovno širino enakomerno med obe cevi.)

Introduction to Algorithms (35. pogl. v prvi izdaji, 33. v drugi). Preparata in Shamos (*Computational Geometry: An Introduction*, 2nd ed., Springer, 1988) pa imata o konveksnih ovojnicah dve precej zajetni poglavji.

Zato poleg količine podatkov, prispelih iz vsake cevi, vodimo še podatek, koliko podatkov je doslej imela vsaka cev možnost poslati. Namreč, da bi bilo porazdeljevanje pravično, moramo predpostaviti, da je vsaka cev (tudi tista, ki je trenutno prazna) prejela (in zatorej tudi odposlala naprej) vsaj toliko podatkov.

Najprej definicije vseh tipov, ki jih bomo potrebovali v rešitvi:

```
const N = 100;                          { Število vhodnih cevi. }
type
  PaketekT = ...;                       { En paketek. }
  PrioritetnaVrstaT = ...;              { Prioritetna vrsta. }
  StrukturaT = record
    PV: PrioritetnaVrstaT;              { Prioritetna vrsta. }
    Prejeto: array [1..N] of integer;   { Količina prejetih podatkov po vsaki cevi. }
    Poslano: integer;                   { Koliko podatkov je doslej imela možnost
                                          poslati vsaka cev. }
  end; {StrukturaT}
```

V resničnih okoljih bi bil PaketekT definiran kot **struct** mbuf v BSD Unixih in **struct** sk_buff v Linux-ih.

In sedaj posamezne operacije. Najprej Pripravi:

```
procedure Pripravi(var Str: StrukturaT);
var Cev: integer;
begin
  PripraviPV(Str.PV);       { Pripravimo prioritetno vrsto. }
  for Cev := 1 to N do      { Iz nobene cevi ni bilo še nič prejetega... }
    Str.Prejeto[Cev] := 0;
  Str.Poslano := 0;         { ... in nič poslanega. }
end; {Pripravi}
```

Nato Vstavi, ki se pokliče, ko pride paketek:

```
procedure Vstavi(var Str: StrukturaT; Paket: PaketekT; Cev: integer);
var Prioriteta: integer;
begin
  { Če je cev doslej prejela manj podatkov, kot jih je imela možnost poslati,
    je to njen problem in zato predpostavimo, da je v resnici prejela
    vsaj Str.Poslano podatkov. }
  if Str.Prejeto[Cev] < Str.Poslano
  then Prioriteta := Str.Poslano
  else Prioriteta := Str.Prejeto[Cev];
  Prioriteta := Prioriteta + Velikost(Paket);    { Tole bo prioriteta tega paketka, }
  VstaviPV(Str.PV, Prioriteta, Paket);  { ki ga zdaj vstavimo v prioritetno vrsto. }
  { Popravimo še količino prejetih podatkov za to cev. }
  Str.Prejeto[Cev] := Prioriteta;
end; {Vstavi}
```

Na koncu še Naslednji, ki je klicana, ko izhodna cev želi poslati naslednji paketek:

```
function Naslednji(var Str: StrukturaT; Paket: PaketekT): boolean;
var Cev, Prioriteta: integer;
begin
  if not IzlociNajmPV(Str.PV, Prioriteta, Paket) then
    Naslednji := false      { Nobenega paketka ni na voljo. }
  else begin
    { Funkcija Vstavi pripiše vsakemu novemu paketu prioriteto tako, da vzame
      količino podatkov, ki jih je oz. bo tista cev že poslala naprej
      (ali pa vsaj imela možnost poslati), preden bo prišel na vrsto tudi
      trenutni paket; temu številu pa prišteje še dolžino trenutnega paketa.
      Če ima paket, ki ga zdajle odpošiljamo, prioriteto Prioriteta, pomeni,
      da je doslej vsaka vhodna cev poslala ali pa vsaj imela možnost poslati
      naprej Prioriteta podatkov. Zato vpišimo to trenutno prioriteto v polje
      Str.Poslano, ki je namenjeno shranjevanju prav tega podatka. }
    Str.Poslano := Prioriteta;      { Največ toliko podatkov je bilo }
                                    { doslej poslanih iz vsake cevi. }
    Naslednji := true;
  end; {if}
end; {Naslednji}
```

Oglejmo si primer delovanja opisane rešitve. Recimo, da imamo dve vhodni cevi in je nekaj časa pošiljala podatke samo prva cev, nato pa začnejo prihajati podatki po obeh ceveh:

Dogodek	Stanje strukture po tem dogodku		
	Prioritetna vrsta	Prejeto	Poslano
	prazna	(100, 0)	100
iz 1. cevi pride paket A dolžine 10	$(A, 110)$	(110, 0)	100
iz 2. cevi pride paket B dolžine 5	$(B, 105), (A, 110)$	(110, 105)	100
iz 2. cevi pride paket C dolžine 10	$(B, 105), (A, 110), (C, 115)$	(110, 115)	100
izhodna cev odpošlje paket B	$(A, 110), (C, 115)$	(110, 115)	105
izhodna cev odpošlje paket A	$(C, 115)$	(110, 115)	110
iz 2. cevi pride paket D dolžine 10	$(C, 115), (D, 125)$	(110, 125)	110
izhodna cev odpošlje paket C	$(D, 125)$	(110, 125)	115
iz 1. cevi pride paket E dolžine 20	$(D, 125), (E, 135)$	(135, 125)	115

Tukaj opisano rešitev najdemo v visoko zmogljivih usmerjevalnikih novejše generacije. Edina kritična točka zgornje rešitve je učinkovita izvedba prioritetne vrste. Običajno se pri slednji tudi zalomi, saj klasične rešitve „iz knjig“ zahtevajo $O(\log m)$ časa za vsako od operacij (m je tukaj število čakajočih paketkov). V večini primerov je to preveč in zato se izdelovalci poslužujejo drugih (približnih) rešitev.

25. državno tekmovanje v znanju računalništva (2001)

NALOGE ZA PRVO SKUPINO

2001.1.1 Tipkanje

Predpostavimo, da lahko vse znake, ki jih želimo natipkati, razdelimo v dve skupini: nekatere tipkamo vedno z levo roko, druge pa vedno z desno. **Napiši program**, ki prebere nek niz in izpiše dolžino najdaljšega takega podniza danega niza, ki ga je mogoče v celoti natipkati z eno samo roko. Na voljo imaš funkcijo SKateroRoko, ki zna za vsak znak povedati, v katero od navedenih dveh skupin spada:

R: 442

```
type Roka = (Leva, Desna);
function SKateroRoko(C: char): Roka; external;
```

Ali, v C-ju:

```
#define Leva 0
#define Desna 1
extern int SKateroRoko(char c);
```

Primer: če se *a* in *e* tipkata vedno z levo roko, *i*, *o* in *u* pa vedno z desno, je pri nizu *aeiou* pravilni odgovor 3 (podniz *iou* lahko v celoti natipkamo z desno roko), pri nizu *aaeaiaoaua* je odgovor 4 (podniz *aaea* v celoti z levo roko), pri *ouaeauo* je odgovor 3 (podniz *aea* v celoti z levo), pri *uaoei* je odgovor 1 (nobenih dveh zaporednih znakov ne moremo natipkati z isto roko), pri *iieeoo* pa je odgovor 2 (*ii* v celoti z desno ali pa *ee* z levo ali pa *oo* z desno).

2001.1.2 Stopniščni avtomat

Avtomat za upravljanje luči na stopnišču skrbi za to, da se ugasnjene luči ob pritisku na tipkalo takoj prižgejo, po eni minuti pa same ugasnejo. Čas do avtomatskega izklopa začnemo šteti šele od trenutka, ko tipkalo izpustimo (izključimo).

R: 443

Poleg te osnovne funkcije pozna avtomat še eno funkcijo za varčne uporabnike: če tipkalo ponovno pritisnemo, med tem ko so luči še prižgane in pred iztekom časa do avtomatskega izklopa, potem se luči ugasnejo takoj.

Napiši program za upravljanje stopniščnega avtomata. Na voljo imaš naslednje podprograme:

```
function TipkaloPritisnjeno: boolean; external;
```

Vrne true, če je (dokler je) tipkalo pritisnjeno, in false, če ni pritisnjeno.

procedure Vklopi; **external**;
Vključi luči.

procedure Izklopi; **external**;
Izključi luči.

procedure Pocakaj1ms; **external**;
Klic tega podprograma zaustavi delovanje programa za 1 ms (eno tisočinko sekunde). Klicanje tega podprograma je edino sredstvo, ki ti je na voljo za merjenje časa. Predpostavi, da je delovanje preostalih delov programa mnogo hitrejše in zato zanemarljivo v primerjavi z 1 ms. Predpostavi tudi, da je odzivni čas 1 ms (ali nekaj ms) dovolj kratek, da se stanovalcem zdi odziv avtomata trenuten, in dovolj kratek glede na zmožnost hitrega pritiskanja na tipko.

Ob začetku delovanja programa naj se luči ugasnejo.

2001.1.3 Besedilo v stolpcu

R: 444 Imamo poljubno dolgo besedilo v neproporcionalni pisavi (vsi znaki so enako široki). Med besedami je zaradi poravnavanja desnega roba besedila včasih lahko po več kot en presledek. Nobena vrstica ni daljša od 60 znakov.

Napiši program, ki bo s standardnega vhoda bral vrstice in preštel vse stavke, ki ustrezajo pogoju, da je celoten stavek napisan v isti vrstici. Predpostaviš lahko, da se besedilo konča s prazno vrstico.

Prvi stavek se prične s prvim znakom v besedilu. Vsak naslednji stavek se prične za znakom za konec stavka (to je lahko pika, klicaj ali vprašaj), ki mu sledi presledek ali konec vrstice.[76] V besedilu ni okrajšav, kjer bi bila uporabljena pika (vsaka pika, ki ji sledi presledek ali konec vrstice, pomeni konec stavka).

konec predhodnega stavka	pričetek novega stavka
`... to save her.`	`It cannot have ...`
`... My God, my God!`	`Has it come to ...`
`... dear what is it?`	`What does this ...`

Primer: v spodnjem besedilu je osem stavkov, ki so v celoti na eni sami vrstici.

```
"In God's name what does this mean?" Harker cried out.  "Dr
Seward, Dr Van Helsing, what is it? What has happened? What
is wrong? Mina, dear what is it? What does that blood mean?
My God, my God! Has it come to this!"  And, raising himself
```

[76] Zaradi enostavnosti smo zanemarili primer, ko piki sledi narekovaj (na koncu dobesednega navedka), kar v resnici tudi pomeni konec stavka, čeprav gornja definicija trdi drugače. Če bi se hoteli ukvarjati s takšnimi podrobnostmi, bi tako ali tako prišli v težave pri parih `?"` in `!"`, ki lahko pomenita konec stavka ali pa tudi ne.

```
to his knees, he beat his hands wildly together.  "Good God
help us! Help her! Oh, help her!"
   With a quick movement  he jumped from bed,  and began to
pull on his clothes,  all the man in him awake  at the need
for instant exertion. "What has happened? Tell me all about
it!"  he cried without pausing.  "Dr Van Helsing,  you love
Mina, I know.  Oh, do something to save her. It cannot have
gone too far yet. Guard her while I look for him!"
```

2001.1.4 Pitagorejske trojice

Opiši postopek, ki bo za dani celi števili a in b (a bo manjši ali enak b, oba pa bosta večja ali enaka 1) ugotovil, koliko je trojic pozitivnih celih števil (x, y, z), za katere je $x^2 + y^2 = z^2$ in je z med a in b (lahko je kateremu od njiju tudi enak). R: 446

Primer: za $a = 5$ in $b = 20$ naj bi tvoj postopek izračunal 12; tedaj namreč obstaja dvanajst trojic števil, ki ustrezajo zgoraj opisanim pogojem. To so:

$$(3,4,5),\quad (4,3,5),\quad (6,8,10),\quad (8,6,10),\quad (5,12,13),\quad (12,5,13),$$
$$(9,12,15),\quad (12,9,15),\quad (8,15,17),\quad (15,8,17),\quad (12,16,20),\quad (16,12,20).$$

NALOGE ZA DRUGO SKUPINO

2001.2.1 Iskalnik

Mojstru Pepetu so naročili, naj postavi spletni iskalnik `www.kdoriscetanajde.si`, ki naj deluje na naslednji način. Vmesnik iskalnika ima polje besed, ki se v iskanih datotekah morajo pojavljati, in polje besed, ki se v njih ne smejo nahajati (eno od teh dveh polj je lahko tudi prazno). Tako bo lahko uporabnik, ki ga zanima glasba, vendar ne prenese cviljenja Britney Spears, v prvo polje vpisal geslo „glasba“, v drugo pa „Britney Spears“. Iskalnik mu bo vrnil imena vseh datotek, ki vsebujejo besedo glasba in v njih ni omenjeno dotično dekletce. Pepetu so poslali paket plošč DVD, ki vsebujejo tekstovne datoteke. Na voljo ima počasen računalnik z zelo malo pomnilnika in zelo veliko diskovno kapaciteto. Ima tudi dovolj časa, da napiše program za iskanje in po potrebi preuredi podatke. Ko pa je iskalnik dostopen uporabnikom, mora delovati čim hitreje. Mojster Pepe je sicer sijajen programer, vendar pa nalogi ni kos. **Opiši** (z besedami) **princip delovanja iskalnika** tako, da bo s tvojo pomočjo znal napisati program. R: 458

2001.2.2 Psevdo-tetris

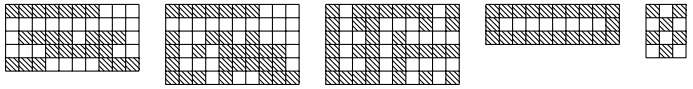
R: 461 Dana je igralna površina, ki je pravokotna karirasta mreža z Y_P vrsticami in X_P stolpci. Vsako od $X_P \times Y_P$ polj (kvadratkov) te mreže je lahko „polno“ ali pa „prazno“.

Po tej igralni površini premikamo lik, ki je kar eno samo polje (kvadrat velikosti 1×1). Na začetku igre je lik tik nad igralno površino (spodnji rob lika se dotika zgornjega roba igralne površine). Lik lahko premikamo navzdol, levo in desno (ne pa navzgor ali po diagonali; vrteti pa ga tako ali tako ne bi imelo smisla). Pri tem ne sme lik nikoli štrleti ven skozi levi ali desni rob igralne površine.

Lik se lahko seveda premika le po praznih poljih igralne površine, ne pa po polnih. Zanima nas, kako „globoko“ (se pravi: kako daleč navzdol) lahko lik pod temi pogoji premaknemo po igralni površini. Če je igralna površina ugodne oblike, se lahko zgodi celo to, da lik na koncu pade skoznjo.[77]

Globino lika merimo od zgornjega roba igralne površine do spodnjega roba lika (enota je dolžina stranice kvadratka); če je uspel pasti skozi igralno površino, si mislimo, da je končal na globini $Y_P + 1$.

Nekaj primerov (pod vsako mrežo je pravilni rezultat za to mrežo, torej podatek o tem, kako globoko lahko pride lik 1×1):



4 7 5 0 1

Opiši postopek, s katerim bi rešil ta problem. Ni treba pisati implementacije v kakšnem konkretnem programskem jeziku, lahko pa si za oporo pomagaš z naslednjimi definicijami:

```
const XP = ...; YP = ...;
type PoljeT = (Prazno, Polno);
     PovrsinaT: array [0..YP − 1, 0..XP − 1] of PoljeT;

{ Opisati moraš delovanje takšne funkcije: }
function KakoGloboko(var Povrsina: PovrsinaT): integer;
```

[77] To je poenostavljena različica naloge D z ACMovega srednjeevropskega študentskega tekmovanja v programiranju (CERC 1999, Praga, 12.–13. nov. 1999). V prvotni nalogi lik, ki ga premikamo skozi mrežo, ni nujno kvadrat 1×1, ampak je lahko večji in poljubne oblike. Pač pa je naloga zagotavljala, da je lik povezan (torej: sestavljen iz enega kosa) in da tudi ostane povezan, če mu odrežemo zgornjih nekaj vrstic.

2001.2.3 CD-predalček

Predvajalnik plošč CD je take izvedbe, da CD leži na predalčku, ki ga premika elektromotorni pogon: predalček lahko prileze ven (se odpre), da lahko zamenjamo ploščo, za predvajanje pa je treba predalček premakniti noter (ga zapreti). R: 462

Za upravljanje odpiranja in zapiranja predalčka ima uporabnik na voljo eno tipko, na kateri piše „ODPRI/ZAPRI“. Vsak nov pritisk tipke (sprememba iz nepritisnjenega stanja tipke v pritisnjeno) mora povzročiti ustrezno krmiljenje motorja: vklop v pravo smer in izklop v končni legi, tako da bo na koncu predalček prišel v izbrano končno stanje. V doseženem končnem stanju (povsem odprto ali povsem zaprto) je treba motor izklopiti.

Tipko lahko pritisnemo tudi sredi premikanja predalčka in spodobi se, da se mehanizem takoj smiselno odzove na zahtevo po spremembi smeri.

Napiši program, ki bo upravljal elektromotor za premikanje predalčka, kot smo predpisali. Na voljo so naslednji podprogrami:

function TipkaPritisnjena: boolean; **external**;
: Vrne true, če je tipka „ODPRI/ZAPRI“ pritisnjena, in false, če ni pritisnjena; pomemben dogodek je le začetek pritiska tipke, ne pa trajanje pritiska ali sprostitev tipke.

function Odprto: boolean; **external**;
: Odčita stanje zunanjega končnega stikala: vrne true, če je predalček v popolnoma izvlečeni legi, sicer pa false.

function Zaprto: boolean; **external**;
: Odčita stanje notranjega končnega stikala: vrne true, če je predalček popolnoma zaprt, sicer pa false.

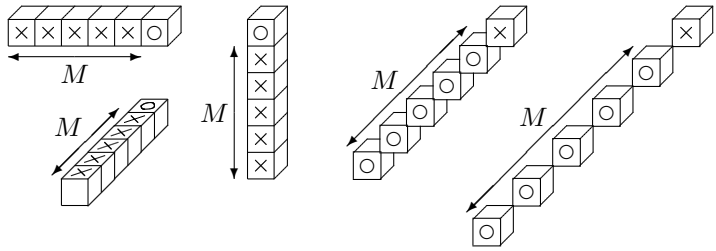
type SmerT = (Stop, Noter, Ven);
procedure Motor(IzbranaSmer: SmerT); **external**;
: S tem podprogramom lahko upravljamo elektromotor: parameter je lahko: Stop (ugasne motor), Noter (vključi motor v smer zapiranja predalčka), Ven (vključi motor v smer odpiranja predalčka).

Začetno stanje vklopa elektromotorja ni znano, prav tako ni znana začetna lega predalčka (lahko pa predpostaviš zaprt predalček, če ti to olajšuje rešitev). Obnašanje programa ob vklopu ni predpisano (npr.: priprt predalček lahko zapre). Predpostavi, da je izvajanje programa mnogo hitrejše od časov med spremembami stanja tipke.

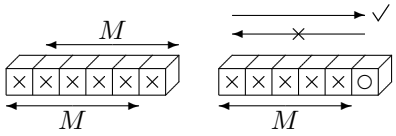
2001.2.4 3-D križci in krožci

R: 463 Imamo kocko s stranico N, ki jo sestavlja N^3 celic. V vsako celico kocke naključno vpišemo znak × ali ○.

Napiši podprogram, ki bo pri dani stranici N in nekem M, ki je večji od 0 in manjši ali enak N, ugotovil, koliko je v kocki vseh takih zaporedij M celic, ki ustrezajo pogoju, da so vse celice zaporedja označene z enakim znakom. Zaporedje celic ima lahko katerokoli smer (po širini, višini, globini, diagonali ravnine ali diagonali prostora).



Ista celica kocke se lahko nahaja v več zaporedjih (slika *a*). Vsako zaporedje mora biti prešteto le enkrat — istega zaporedja v nasprotni smeri ne štejemo (slika *b*).



Tvoj podprogram naj ustreza naslednjim deklaracijam:

```
const N = ...;
type ZnakT = (Krizec, Krozec);
     KockaT = array [0..N − 1, 0..N − 1, 0..N − 1] of ZnakT;

function Prestej(Kocka: KockaT; M: integer): integer;
```

Ali:

```
#define N ...
#define Krizec 0
#define Krozec 1
typedef int KockaT[N][N][N];

int Prestej(KockaT Kocka, int M);
```

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

[Na začetku tekmovanja smo tekmovalcem najprej razdelili naslednja navodila. Nekaj minut kasneje so dobili tudi besedilo nalog, za reševanje pa so imeli slabe tri ure časa. — *Op. ur.*]

Vsaka naloga zahteva, da **napišeš program**, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*`.in` in izpisujejo svoje rezultate v *imenaloge*`.out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil listek, na katerem je napisano uporabniško ime in geslo, s katerim se boš prijavil na računalnik. Na vsakem računalniku imaš na voljo enoto (disk) `U:`, na kateri lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku Pascal, C ali C++. Za delo lahko uporabiš `turbo` (Turbo Pascal), `tc` (Turbo C) ali `gcc`/`g++` (GNU C/C++ — command line compiler).

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `rtk.exe`, ki ga lahko uporabiš za preverjanje svojih rešitev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

```
rtk imenaloge.pas
rtk imenaloge.c
rtk imenaloge.cpp
rtk gcc -o imenaloge imenaloge.c
rtk g++ -o imenaloge imenaloge.cpp
```

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Na spletni strani boš dobil obvestilo o tem, ali je program na testne primere odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot pol minute, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z drugimi datotekami kot

z vhodno in izhodno. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov, prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Če si oddal N programov za to nalogo in je najboljši med njimi pravilno rešil M (od desetih) testnih primerov, dobiš pri tej nalogi $\max\{0, 10M-3(N-1)\}$ točk. Z drugimi besedami: vsak pravilno rešen testni primer ti prinese 10 točk, za vsako oddajo (razen prve) pri tej nalogi pa se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk.

Skupno število točk tekmovalca je vsota po vseh nalogah. Tekmovalce razvrstimo po skupnem številu točk.

Vsak tekmovalec se mora sam zase odločiti o tem, katerim nalogam bo posvetil svoj čas, v kakšnem vrstnem redu jih bo reševal in podobno. Verjetno je priporočljivo najprej reševati lažje naloge.

Poskusna naloga (ne šteje k tekmovanju) `poskus.in`, `poskus.out`

Napiši program, ki iz vhodne datoteke prebere eno število (le-to je v prvi vrstici, okoli njega ni nobenih dodatnih presledkov ipd.) in izpiše njegov desetkratnik v izhodno datoteko.

Primer vhodne datoteke:	Ustrezna izhodna datoteka:
`123`	`1230`

Primer rešitve:

```
program PoskusnaNaloga;
var T: text; i: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * i); Close(T);
end. {PoskusnaNaloga}
```

```
#include <stdio.h>
int main() {
  FILE *f = fopen("poskus.in", "rt");
  int i; fscanf(f, "%d", &i); fclose(f);
  f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * i);
  fclose(f); return 0;
}
```

```
#include <fstream.h>
int main() {
  ifstream ifs("poskus.in"); int i; ifs >> i;
  ofstream ofs("poskus.out"); ofs << 10 * i;
  return 0;
}
```

NALOGE ZA TRETJO SKUPINO

2001.3.1 Števila v ogledalu

`addrev.in`, `addrev.out`

Dogovorimo se, da bomo števila zapisovali v nasprotni smeri — najbolj leva števka naj predstavlja enice, naslednja desetice in tako naprej. Poleg tega pri pretvorbi števila v naš novi zapis tudi zbrišimo morebitne ničle, če se jih kaj pojavlja na začetku obrnjenega števila. Tako na primer iz 345 dobimo `543`, iz 12 dobimo `21`, iz 120 in 1200 pa ravno tako `21`. R: 464

Napiši program, ki bo znal seštevati števila v obrnjenem zapisu. Iz prve vrstice *vhodne datoteke* naj prebere dve pozitivni celi števili v obrnjenem zapisu (ločeni sta s presledkom) in v *izhodno datoteko* izpiše njuno vsoto v obrnjenem zapisu. Ker, kot smo videli zgoraj, ta zapis ni enoličen, naj pri branju vhodnih podatkov predpostavi, da se pri pretvorbi v obrnjeni zapis ni izgubila nobena ničla: če torej v vhodni datoteki naletiš na `21`, si to razlagaj kot število 12, ne pa kot 120 ali 1200. Ko vsoto pri izpisu pretvarjaš v obrnjeni zapis, ne pozabi, da morebitnih začetnih ničel ne smeš izpisati.[78]

Števila, s katerimi boš imel opravka (tako seštevanci kot vsote), utegnejo biti večja od 2^{16}, gotovo pa bodo manjša od 2^{31} (zato uporabljaj `longint` ali **`long`**).

Primer treh vhodnih datotek:	Pripadajoče izhodne datoteke:
`123 45`	`573`
`543 534`	`87`
`207 892`	`1`

2001.3.2 Oklepajski izrazi

`oklizr.in`, `oklizr.out`

Oklepajski izraz je niz, ki vsebuje samo oklepaje in zaklepaje (torej znaka „`(`“ in „`)`“) in so le-ti hkrati pravilno gnezdeni (torej ima vsak oklepaj tudi pripadajoči zaklepaj in obratno). Oklepajski izrazi so na primer `()`, `(()())()()`, `()((())())`, `()()`; nizi `(()`, `()((()())`, `)(()` in `(()))(()` pa ne. Tudi prazen niz velja za oklepajski izraz. R: 466

Oklepajski izrazi reda N so natanko tisti oklepajski izrazi, ki vsebujejo točno N oklepajev in N zaklepajev. V *vhodni datoteki* bo v prvi vrstici neko celo število N ($1 \le N \le 15$), tvoj program pa naj v *izhodno datoteko* izpiše vrstico, ki vsebuje število oklepajskih izrazov reda N.

[78] To je naloga A z ACMovega srednjeevropskega študentskega tekmovanja v programiranju (CERC 1998, Praga, 14. nov. 1998; #713 v zbirki na `online-judge.uva.es`).

Ker obstaja skoraj deset milijonov oklepajskih izrazov reda 15, ti priporočamo, da delaš z 32-bitnimi spremenljivkami (longint ali **long**).

Primer treh vhodnih datotek:	Ustrezne izhodne datoteke:
`2`	`2`
`4`	`14`
`3`	`5`

Vsi oklepajski izrazi reda 3 so na primer:

```
()()()    ()(())    (())()    (()())    ((()))
```

Reda 4 pa:

```
(()()())    (())(())    ()()()()
(()(()))    (())()()    ()()(())
((())())    ((()))()    ()(())()
((()()))    (()())()    ()(()())
(((())))                ()((()))
```

2001.3.3 Parlament

`parlamen.in`, `parlamen.out`

R: 469

Po dolgih letih zdrah in prepirov politični analitiki že vedo, da bo vlado po volitvah vedno oblikovala takšna koalicija strank, ki ima v parlamentu najšibkejšo možno večino. Napiši program, ki jim bo znal pri danem izidu volitev povedati, katere stranke bodo v koaliciji.

Vhodna datoteka opisuje sestavo parlamenta. Vsa števila so v prvi vrstici datoteke; ta se začne s celim številom N, ki pove, koliko strank je prišlo v parlament (zaradi petodstotnega praga za vstop v parlament je $1 \leq N \leq 20$). Sledi N pozitivnih celih števil $P_1, \ldots, P_N$, ki povedo, da ima prva stranka P_1 poslancev, druga P_2 poslancev in tako naprej. Predpostaviš lahko, da skupno število poslancev $P = P_1 + \ldots + P_N$ ne presega 10000. Tvoj program naj v *izhodno datoteko* napiše eno vrstico, v kateri bodo (s presledki ločene) številke strank (štejejo se od 1 do N), ki tvorijo najšibkejšo možno večinsko vlado: to je torej skupina strank, ki imajo skupaj čim manj poslancev, vendar pa več kot $P/2$. Če je takih najšibkejših koalicij več (torej vse enako šibke), lahko program izpiše katero koli od njih. Vrstni red, v katerem izpišeš koalicijske stranke, ni pomemben (`3 6 8` je enakovredno `8 3 6`, ipd.).

Primer šestih vhodnih datotek:	Možne pripadajoče izhodne datoteke:
`10 10 9 8 7 6 5 4 3 2 1`	`1 9 3 8 6`
`3 3333 3333 3333`	`3 2`

`3 3333 3333 3333`	`2 1`
`3 3333 3334 3333`	`1 3`
`4 25 25 25 25`	`1 2 3`
`4 25 25 25 24`	`2 3`

2001.3.4 **Kletke** `kletke.in`, `kletke.out`

Podjetje PasjiHlevi, d. d., je lastnik pravokotnega zemljišča, ki je v bistvu karirasta mreža, sestavljena iz $M \times N$ enako velikih kvadratnih celic. Med dve sosednji celici (torej taki s skupno stranico) lahko postavijo jekleno pregrado (v tem primeru neposreden prehod med tema dvema celicama ni mogoč, drugače pa je). Pojem „kletka“ pomeni skupino teh kvadratnih celic, med katerimi je mogoče prosto prehajati (če je npr. med dvema celicama iste kletke pregrada, mora obstajati neka pot naokoli po drugih celicah te kletke) in ki ji ni mogoče dodati nobene nove celice, ne da bi ta pogoj prenehal veljati (kletka je torej vse naokoli ograjena od ostalih kletk in od zunanjosti).

R: 472

Vodstvo podjetja sedaj razmišlja, kako bi organizirali kletke, da bi pridobili čimveč strank. Prosijo te, da napišeš program, ki bo za dano stanje povedal, koliko je sploh kletk, kolikšna je površina največje in kolikšna najmanjše kletke.

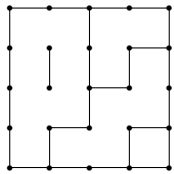
V *vhodni datoteki* sta v prvi vrstici najprej napisani dimenziji M in N (v tem vrstnem redu; velja $1 \le M \le 100$, $1 \le N \le 100$), potem pa sledi N vrstic; v vsaki vrstici je M dvomestnih števil (ločenih s presledki), ki predstavljajo stanje pregrad okoli posameznih celic. (Vrstice so podane od severa proti jugu, znotraj vsake vrstice pa so celice navedene od zahoda proti vzhodu.) Možne vrednosti teh števil so od 00 do 15, vrednost pa dobimo tako, da seštejemo 1, če na severnem robu celice *ni* pregrade, 2, če je ni na vzhodnem, 4, če je ni na južnem, in 8, če je ni na zahodnem. 0 torej pomeni, da je celica povsem zagrajena (in tvori kletko zase), 15 pa, da okoli nje ni nobene pregrade. Na zunanjem robu zemljišča so pregrade vedno zagotovo postavljene (torej tiste, ki niso med dvema celicama, pač pa med celico in zunanjim svetom). Pregrada med dvema celicama vedno velja za obe (če ima torej npr. neka celica vzhodno pregrado, je ta pregrada navedena tudi kot zahodna pregrada pri njeni vzhodni sosedi, ipd.) — „enosmernih“ pregrad ni.

V *izhodni datoteki* mora biti v prvi vrstici število kletk, v drugi površina najmanjše kletke in v tretji vrstici površina največje kletke. Površina kletke je definirana kot število celic, ki to kletko sestavljajo.

Primer vhodne in izhodne datoteke kaže slika na str. 440.

Primer vhodne datoteke:

```
4 4
06 12 06 08
05 05 01 04
07 09 06 09
01 02 09 00
```



Tu imamo štiri kletke s površinami 1, 3, 5 in 7, tako da je ustrezna izhodna datoteka takšna:

```
4
1
7
```

Primer vhodne in izhodne datoteke za nalogo 2001.3.4.

2001.3.5 Prefiksi

`prefiksi.in`, `prefiksi.out`

R: 478

Dana je množica nizov $\{S_1, \ldots, S_N\}$, ki jim recimo *vzorci*, in še nek niz S. Napiši program, ki bo za dane $S_1, \ldots, S_N$ in S ugotovil, kako dolg je najdaljši začetek (prefiks) niza S, ki se ga dá dobiti s stikanjem (sestavljanjem) vzorcev $S_1, \ldots, S_N$. Pri tem lahko vzorce stikamo v poljubnem vrstnem redu, vsakega od njih pa lahko uporabimo ničkrat, enkrat ali večkrat. Dolžina niza je definirana kot število znakov v njem.[79]

Prva vrstica *vhodne datoteke* vsebuje samo celo število N ($1 \leq N \leq 100$). Sledi ji N vrstic, ki vsebujejo vsaka po en vzorec (prva S_1, druga S_2 in tako naprej), za njimi pa je še vrstica, ki vsebuje niz S. Vsak od nizov $S_1, \ldots, S_N$ je dolg največ 100 znakov, niz S pa največ 50000 znakov.[80] Vsi ti nizi so sestavljeni le iz velikih črk angleške abecede (`A`, `B`, ..., `Z`). Program naj v *izhodno datoteko* izpiše dolžino najdaljšega prefiksa niza S, ki se ga da sestaviti s stikanjem vzorcev $S_1, \ldots, S_N$.

Primer štirih vhodnih datotek:

`3`	`1`	`4`	`4`
`ABC`	`A`	`SPQR`	`SPQR`
`BC`	`BA`	`RSQP`	`RSQP`
`CA`		`Q`	`Q`
`ABCBCCABCA`		`QR`	`QR`
		`QRSQPQRSQPR`	`QRSQPQRPSQR`

Pripadajoče izhodne datoteke:

`9`	`0`	`10`	`7`

[79] Ta naloga temelji na eni od nalog z računalniških olimpijad (IOI 1996, Veszprém, Madžarska, 25. jul.–1. avg. 1996; druga naloga drugega dne). V prvotni nalogi so dolgi vzorci največ 20 znakov, vendar pa je lahko niz S dolg do 500 000 znakov.

[80] *Opomba:* V resnici smo dolžino niza S v testnih primerih na koncu omejili na 30000 znakov, da bi zmanjšali morebitne težave pri dodeljevanju pomnilnika pri tekmovalcih, ki so uporabljali katerega od 16-bitnih prevajalnikov; vendar pa smo v besedilu naloge to pozabili navesti. (Formalno gledano s tem seveda ni nič narobe, saj so testni primeri še vseeno ustrezali specifikacijam.)

2001.3.6 Podobnost med dokumenti

`docclust.in, docclust.out`

Pogosto je koristno, če znamo za dani dve besedili nekako oceniti, kako podobni ali različni sta si. Da postanejo stvari bolj obvladljive in preprostejše, bomo pri tej nalogi besedila gledali kot „vreče“ besed (torej se ne zmenimo za vrstni red besed v besedilu, pač pa le za to, kolikokrat se posamezna beseda v njem pojavlja). Če vse možne besede, ki se pojavljajo v opazovani skupini besedil, oštevilčimo od 1 do n, lahko besedilo opišemo z vektorjem oblike $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, kjer komponenta x_i pove, kolikokrat se v tem besedilu pojavlja beseda i. Če dve besedili predstavljata vektorja $\mathbf{x}$ in $\mathbf{y}$, definirajmo podobnost med tema besediloma kot

R: 480

$$P(\mathbf{x}, \mathbf{y}) = \frac{x_1y_1 + x_2y_2 + \ldots + x_ny_n}{\sqrt{x_1^2 + \ldots + x_n^2}\sqrt{y_1^2 + \ldots + y_n^2}}.$$

Tvoja naloga je napisati program, ki bo znal v dani skupini besedil poiskati tisti dve, ki sta si najbolj podobni (se pravi: tisti, pri katerih je $P(\mathbf{x}, \mathbf{y})$ največji).

V prvi vrstici *vhodne datoteke* je zapisano število besedil, ki jih pri tem testnem primeru opazujemo (recimo mu D; to bo celo število, $2 \leq D \leq 20$). Sledi D vrstic, od katerih vsaka vsebuje po eno od teh besedil. Celo besedilo je torej v eni sami vrstici; sestavljeno je iz vsaj ene in kvečjemu 20 besed, vsaka beseda pa je dolga največ 20 znakov. Besede so sestavljene samo iz velikih črk angleške abecede, ločene so s po enim presledkom, pred prvo in za zadnjo besedo pa ni v vrstici nobenih dodatnih presledkov. Cela vrstica ne bo nikoli daljša od 250 znakov.

Tvoj program naj v *izhodno datoteko* v eno vrstico izpiše indeksa dveh najpodobnejših besedil (besedila si mislimo oštevilčena od 1 do D; najprej izpiši manjšega od obeh indeksov, nato večjega), ločena s presledkom. Testni primeri bodo sestavljeni tako, da bo najboljši vedno natanko en par (torej ne bo na prvem mestu več enako dobrih parov).

Primer dveh vhodnih datotek:

```
8
I AM AS I AM AND SO WILL I BE
BUT HOW THAT I AM NONE KNOITH TRULIE
BE YT EVILL BE YT WELL BE I BONDE BE I FRE
I AM AS I AM AND SO WILL I BE
I LEDE MY LIF INDIFFERENTELYE
I MEANE NOTHING BUT HONESTELIE
AND THOUGH FOLKIS JUDGE FULL DYVERSLYE
I AM AS I AM AND SO WILL I DYE

5
ENA DVE TRI STIRI ENA DVE TRI STIRI
STIRI PET SEST SEDEM OSEM DEVET DESET
STIRI ENAJST DVANAJST TRINAJST STIRI STIRINAJST PETNAJST SESTNAJST
ENA DVE TRI STIRI SEDEMNAJST OSEMNAJST DEVETNAJST DVAJSET ENAA DVEE TRII
STIRI ENAINDVAJSET ENA DVAINDVAJSET DVE TRIINDVAJSET TRI
```

Pripadajoči izhodni datoteki:

```
1 4

1 5
```

REŠITVE NALOG ZA PRVO SKUPINO

R2001.1.1 Tipkanje

N: 429 Z vhoda berimo znak za znakom. V spremenljivki TrenRoka si zapomnimo, s katero roko smo natipkali prejšnji znak, spremenljivka StSTrenRoko pa nam pove, koliko zadnjih znakov (vključno s prejšnjim) smo natipkalo s to roko. Če tudi novi znak natipkamo z isto roko, je treba samo povečati vrednost StSTrenRoko za 1. Če pa začnemo tipkati z drugo roko, mora začeti tudi StSTrenRoko šteti spet od 1 naprej. Ob zamenjavi roke (pa tudi na koncu vhodnih podatkov) poglejmo še, če je pravkar končano zaporedje znakov, natipkanih z isto roko, mogoče najdaljše takšno zaporedje v doslej prebranem nizu (spremenljivka MaxZEnoRoko).

```
program ZEnoRoko;

  type Roka = (Leva, Desna);
  function SKateroRoko(C: char): Roka; external;

var
  TrenRoka, NovaRoka: Roka;
  StSTrenRoko, MaxZEnoRoko: integer;
  c: char; ZeTipkamo: boolean;
begin

  ZeTipkamo := false; StSTrenRoko := 0; MaxZEnoRoko := 0;
  TrenRoka := Leva; { Samo toliko, da vrednost spremenljivke ne bo nedefinirana. }
  while not (Eof or Eoln) do begin

    { Preberimo naslednji znak. S katero roko se ga tipka?}
    Read(c); NovaRoka := SKateroRoko(c);

    if (NovaRoka = TrenRoka) and ZeTipkamo then
      { Z isto roko kot doslej natipkamo še en znak več. }
      StSTrenRoko := StSTrenRoko + 1
    else begin
      { Zamenjamo roko. Mogoče smo z dosedanjo roko dosegli nov rekord. }
      if StSTrenRoko > MaxZEnoRoko then MaxZEnoRoko := StSTrenRoko;
      StSTrenRoko := 1; TrenRoka := NovaRoka;
    end; {if}

    ZeTipkamo := true;
  end; {while}

  { Mogoče pa je rekordno zaporedje tisto na koncu niza. }
```

```
  if StSTrenRoko > MaxZEnoRoko then MaxZEnoRoko := StSTrenRoko;
  WriteLn('Dolžina najdaljšega podzaporedja: ', MaxZEnoRoko);
end. {ZEnoRoko}
```

R2001.1.2 Stopniščni avtomat

Naš program bo v zanki opazoval stanje tipkala. Ko opazimo, da je tipkalo pritisnjeno (TipkaloNovo = true), ob prejšnji meritvi (TipkaloStaro = false) pa še ni bilo, vemo, da je uporabnik pritisnil tipkalo in moramo na to odreagirati. Če je luč vključena (spremenljivka Vkljucena), jo takoj izključimo, sicer pa jo vključimo in si zapomnimo, da jo bo treba čez minuto ugasniti (PreostaliCas). Slednjo spremenljivko nato v vsaki ponovitvi zanke zmanjšamo in tako štejemo čas v milisekundah; ko pade na 0, luč ugasnemo.

N: 429

```
program StopniscniAvtomat(input,output);

  var TipkaloNovo, TipkaloStaro: boolean;   { stanje tipkala }
      Vkljuceno: boolean;                   { stanje luči }
      PreostaliCas: integer;                { čas do izklopa v milisekundah }

  function TipkaloPritisnjeno: boolean; external;
  procedure Vklopi; external;
  procedure Izklopi; external;
  procedure Pocakaj1ms; external;

begin {StopniscniAvtomat}
  Izklopi; Vkljuceno := false;
  TipkaloStaro := false; PreostaliCas := 0;
  repeat
    TipkaloNovo := TipkaloPritisnjeno;
    if TipkaloNovo and not TipkaloStaro then
    begin { začetek pritiska tipke }
      if Vkljuceno then Izklopi else Vklopi;
      Vkljuceno := not Vkljuceno;
    end; {if}
    TipkaloStaro := TipkaloNovo;
    if not Vkljuceno then PreostaliCas := 0
    else if TipkaloNovo then PreostaliCas := 60000;
    if PreostaliCas > 0 then PreostaliCas := PreostaliCas - 1
    else if Vkljuceno then begin Izklopi; Vkljuceno := false end;
    Pocakaj1ms;
  until false;
end. {StopniscniAvtomat}
```

R2001.1.3 Besedilo v stolpcu

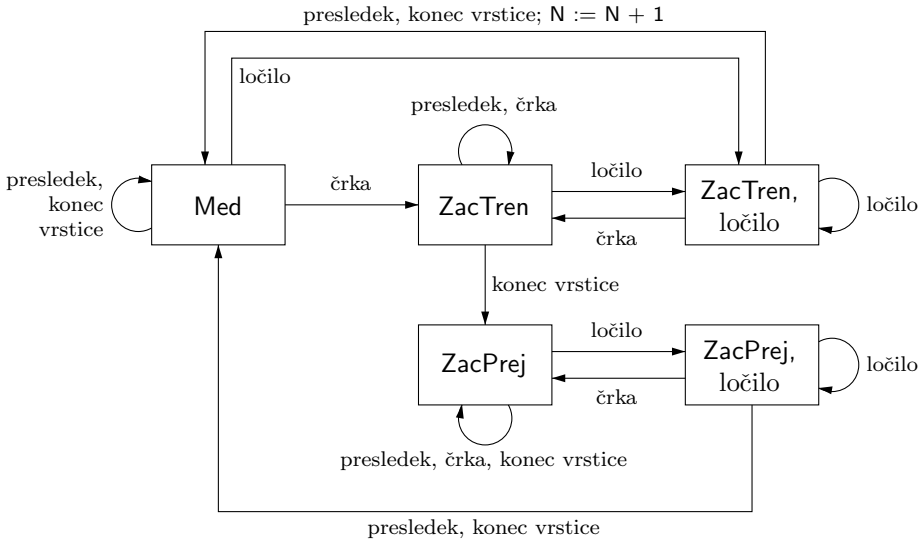
N: 430 Spodnji program pri branju besedila razlikuje med tremi možnimi stanji: lahko se nahaja v praznem prostoru (presledki) med dvema stavkoma, lahko je v stavku, ki se je začel v trenutni vrstici, ali pa v stavku, ki se je začel že v neki prejšnji vrstici. Ko naletimo na znak, ki ni presledek, se iz stanja Med premaknemo v ZacTren, ker se je s tem v trenutni vrstici začel nov stavek. Ko naletimo na ločilo na koncu stavka, po potrebi (če se trenutni stavek ni začel že v neki prejšnji vrstici) povečamo števec N za 1, nato pa se spet premaknemo v stanje Med. Na koncu vrstice se stanje ZacTren spremeni v ZacPrej, ker se z vidika naslednje vrstice trenutni stavek pač začenja v neki predhodni vrstici.

```
program StetjeStavkov;
var Stanje: (Med, ZacTren, ZacPrej);
    S: string; i, L, N: integer;
begin
  ReadLn(S); Stanje := Med; N := 0;
  while S <> '' do begin
    if Stanje = ZacTren then Stanje := ZacPrej;
    L := Length(S);
    for i := 1 to L do begin
      if (Stanje = Med) and (S[i] <> ' ') then Stanje := ZacTren;
      if (S[i] in ['.', '!', '?']) and ((i = L) or (S[i + 1] = ' ')) then begin
        if Stanje = ZacTren then N := N + 1;
        Stanje := Med;
      end; {if}
    end; {for}
    ReadLn(S);
  end; {while}
  { Če se zadnji stavek ne konča s končnim ločilom, ga doslej še nismo šteli. }
  if Stanje = ZacTren then N := N + 1;
  WriteLn(N);
end. {StetjeStavkov}
```

Program bi lahko predelali tudi tako, da bi bral datoteko znak za znakom, ne pa celo vrstico naenkrat (glej sliko na str. 445); to bi bilo lahko koristno, če vnaprej ne bi vedeli, kako dolge utegnejo biti posamezne vrstice (in bi radi, da bi program deloval zanesljivo tudi pri datotekah z zelo dolgimi vrsticami).

Kot se pri delu z nizi in besedili pogosto zgodi, lahko tudi tu nalogo rešimo krajše s pomočjo orodij operacijskega sistema unix:

```
sed 's/[.?!]/!/g; s/n//g; s/! /!n/g'' | tr n '\n' | \
sed 's/^.*!$/n /g; s/^[^n]*$/a/g' | tr -d '\n' | \
sed 's/a[an]*//g' | wc -w
```



Ilustracija k rešitvi naloge 2001.1.3.

To je primer končnega avtomata za štetje stavkov, ki so v celoti znotraj ene same vrstice. Začnemo v stanju `Med` in postavimo `N := 0`. Nato beremo vhodno besedilo znak za znakom in spreminjamo stanje, kot nam kažejo puščice. „Ločilo“ pomeni tu znak `.`, `?` ali `!`, „črka“ pa poljuben znak, ki ni presledek, ločilo ali konec vrstice. Na koncu je v spremenljivki `N` število stavkov, ki so v celoti znotraj ene same vrstice (težave so lahko le v primeru, če se zadnji stavek ne konča s končnim ločilom).

Za začetek vsa končna ločila spremenimo v klicaje (da nam kasneje ne bo treba povsod naštevati vseh treh); nato, če za končnim ločilom pride presledek, ga spremenimo v črko `n`, vse ostale `n` pa še pred tem pobrišimo. Če nato vsak `n` spremenimo v znak za konec vrstice (s programom `tr`), smo dobili datoteko, v kateri se vsak stavek konča na koncu vrstice (ni pa nujno konec vsake vrstice tudi konec stavka). Iz vsake vrstice, ki se konča s klicajem, naredimo znak `n` in presledek, iz vsake ostale vrstice pa nato poljuben drug znak, recimo `a`. Potem pobrišimo znake za konec vrstice (ukaz `tr -d`) in tako staknimo vse v eno samo dolgo vrstico. Tako iz vsakega stavka nastane „beseda“, ki se konča na `n`, pred njim pa ima še nič ali več `a`-jev, namreč po enega za vsako vrstico (razen zadnje), v kateri se ta stavek še pojavlja. Stavki, ki so v celoti znotraj ene same vrstice, torej dobijo `n` brez `a`-jev; vse ostale besede pobrišimo in nato (z `wc`) preštejmo, koliko jih je ostalo. Slabost te rešitve je med drugim ta, da po stikanju vrstic nastane datoteka z eno samo dolgo vrstico, ki ima vsaj toliko znakov, kolikor je imela prvotna datoteka vrstic. Ker `sed` svoj vhod bere vrstico za vrstico, zna biti nerodno, če bo dobljena vrstica predolga. Lahko bi torej takoj za `tr -d '\n'` vrinili še `tr ' ' '\n'` in tako poskrbeli, da bo vsaka

„beseda“ (ki predstavlja po en stavek prvotne datoteke) v svoji vrstici. Zdaj so lahko vrstice predolge le, če nam je nekdo hudobno podtaknil besedilo s patološko dolgimi stavki.

R2001.1.4 Pitagorejske trojice

N: 431 Recimo, da bi se osredotočili na trojice z neko konkretno hipotenuzo z. Če hočemo, da velja $x^2 + y^2 = z^2$ in sta x ter y pozitivni celi števili, mora biti $0 < x < z$ in $y = \sqrt{z^2 - x^2}$. Lahko gremo torej po vseh možnih x in pri vsakem preverimo, če je $\sqrt{z^2 - x^2}$ res celo število.

```
program PitagorejskeTrojice;
var n, x, z, a, b: integer; y: real;
begin
  ReadLn(a, b); n := 0;
  for z := a to b do
    for x := 1 to z - 1 do begin
      y := Sqrt(z * z - x * x);
      if y = Trunc(y) then n := n + 1;
    end; {for x}
  WriteLn(n);
end; {PitagorejskeTrojice}
```

Postopek bi lahko še malo izboljšali, če bi gledali le trojice z $x < y$. Tistih z $y < x$ je namreč prav toliko kot takih z $x < y$ in jih lahko torej upoštevamo preprosto tako, da tiste z $x < y$ štejemo dvojno. (Takih z $x = y$ pa sploh ni, saj bi to pomenilo, da je $z^2 = 2x^2$ in zato $z = x\sqrt{2}$, potem pa z in x ne bi mogla biti oba hkrati celi števili.) Čim bi opazili, da je $2x^2 \geq z^2$, bi notranjo zanko prekinili, saj bi vedeli, da bomo odtlej pri trenutnem z dobivali le še trojice z $y < x$.

Še ena drobna izboljšava (od katere v praksi ne bi bilo kakšne posebne koristi): ker je $x > 0$, je $y^2 = z^2 - x^2 < z^2$, zato $y < z$ oz. $y \leq z - 1$ (ker sta y in z cela). Zato je $x^2 = z^2 - y^2 \geq z^2 - (z-1)^2 = 2z - 1$, torej $x \geq \sqrt{2z-1}$. Torej ni treba, da začne notranja zanka pri $x = 1$, ampak bi lahko začela pri $\lceil\sqrt{2z-1}\rceil$.

Rešitev brez operacij v plavajoči vejici. Našo rešitev lahko spremenimo tudi tako, da ne bo računala s števili v plavajoči vejici — npr. če nas skrbi, da je funkcija Sqrt prepočasna ali pa se bojimo morebitnih numeričnih nenatančnosti v njej (čeprav v praksi za to najbrž ni prav velikih možnosti). Če smo se odločili za nek konkreten z, ustreza potem vsakemu x natanko en y, namreč $y = \sqrt{z^2 - x^2}$. Recimo, da bi poleg trenutnega x hranili še neko možno vrednost y. Potem, če vidimo, da je $x^2 + y^2 > z^2$, vemo, da je ta y prevelik, in ga lahko zmanjšamo. Če namesto $>$ velja enakost, smo odkrili novo trojico in lahko povečamo števec n. Nato pa, če pa velja $x^2 + y^2 \leq z^2$, je čas, da se

pomaknemo na naslednji x (torej x povečamo za 1), pa se bomo v nadaljevanju spet ubadali s tem, če je y kaj prevelik in podobno.

```
program PitagorejskeTrojice2;
var n, x, y, z, a, b: integer;
begin
  ReadLn(a, b); n := 0;
  for z := a to b do begin
    x := 2; y := z - 1;
    while x < y do begin
      if x * x + y * y > z * z then y := y - 1
      else begin
        if x * x + y * y = z * z then n := n + 2;
        x := x + 1;
      end; {if}
    end; {while}
  end; {for}
  WriteLn(n);
end. {PitagorejskeTrojice2}
```

Če je množenje zelo počasna operacija, bi lahko vrednosti x^2, y^2 in z^2 hranili tudi v samostojnih spremenljivkah in jih ob spremembah vrednosti x, y in z popravljali le s seštevanjem in odštevanjem: ko se x poveča za 1, se x^2 poveča za $2x+1$, in ko se y zmanjša za 1, se y^2 zmanjša za $2y-1$.

Dokaz pravilnosti te rešitve. O tem, da ta program res ne spregleda nobene pitagorejske trojice, se lahko prepričamo z naslednjo zančno invarianto: na začetku vsake ponovitve zanke **while** velja, da je zanka že naštela vse trojice (X, Y, z) in (Y, X, z) pozitivnih celih števil, za katere je $X^2+Y^2 = z^2$ in $X < Y$ in ($X < x$ in/ali $y < Y$).

Na začetku prve ponovitve to očitno velja, saj pogojema $X < x$ in $y < Y$ ustrezata le $X = 1$ in $Y = z$, pitagorejskih trojic oblike $1^2 + Y^2 = z^2$ ali $X^2 + z^2 = z^2$ pa ni (če zahtevamo, da so števila pozitivna) in zanka dotlej res tudi še ni nobene naštela.

Recimo zdaj, da je naša invarianta veljala na začetku neke ponovitve zanke **while**. Prepričati se hočemo, da bo veljala tudi na koncu (in s tem tudi na začetku naslednje ponovitve). Ločimo tri možnosti: (1) Če pri trenutnih x in y velja $x^2 + y^2 = z^2$, bomo to trojico zdaj našteli (se pravi: povečali n; ker ga povečamo za 2, s tem upoštevamo tudi trojico (y, x, z)), poleg tega pa s tem x-om ne more biti v paru noben drug y in s tem y-om noben drug x. Torej se invarianta ohrani tudi potem, ko x povečamo in y zmanjšamo za 1 (gornji program pravzaprav le poveča x). (2) Če je $x^2 + y^2 > z^2$, potem s trenutnim y-om prav gotovo ni v paru noben $X \geq x$, za $X < x$ pa to velja že po predpostavki, da je invarianta veljala ob začetku izvajanja zanke. Zato se invarianta ohrani, ko y zmanjšamo za 1. (3) Če je $x^2 + y^2 < z^2$, potem s

trenutnim x-om prav gotovo ni v paru noben $Y \le y$, za $Y > y$ pa velja to že po predpostavki, da je invarianta veljala ob začetku izvajanja zanke. Zato se invarianta ohrani, ko x povečamo za 1.

V trenutku, ko se konča zadnja ponovitev zanke **while** (pri nekem z), sta x in y enaka (sicer se zanka ne bi končala). Skupaj z ugotovitvijo, da na koncu vsake ponovitve velja gornja invarianta, vidimo, da je zanka dotlej že našla vse trojice (X, Y, z) in (Y, X, z) pozitivnih celih števil, za katere je $X^2 + Y^2 = z^2$ in $X < Y$ in ($X < x$ in/ali $x < Y$). Ali je možno, da smo kaj spregledali? Za $X > Y$ se nam ni treba zanimati, kajti če odkrijemo vsako trojico z $X < Y$ in jo štejemo dvojno, smo s tem šteli že tudi vse trojice z $X > Y$. Za $X = Y$ se nam tudi ni treba zanimati, ker takih trojic ni ($X^2 + X^2 = z^2$ bi pomenilo, da je $z = X\sqrt{2}$ in torej ni celo število). Pri $X < Y$ pa so edine, ki jih še nismo šteli, take, pri katerih ne velja niti $X < x$ niti $y < Y$; torej take z $X \ge x$ in $y \ge Y$; toda zdaj sta x in y enaka in bi za takšne trojice veljalo $X \ge Y$, ne pa $X < Y$. Torej smo res našteli vse iskane trojice pri trenutnem z. Ker gre z v zunanji zanki **for** po vseh celih številih od a do b (vključno s tema dvema), bomo res našli vse trojice, po katerih sprašuje naloga.

Rešitev iz teorije števil. Doslej opisani rešitvi imata s štetjem pitagorejskih trojic pri hipotenuzi z pač $O(z)$ dela, za pregled $n = b - a + 1$ možnih vrednosti z pa zato skupaj $O(nb)$ dela; do hitrejših rešitev lahko pridemo s pomočjo ugotovitev iz teorije števil. Bralec, ki se mu bo zdelo naše razmišljanje preveč matematično, lahko preostanek te rešitve brez posebne škode preskoči.

Vzemimo dve celi števili k in l; naj bo $k > l$. Potem tvorijo števila $x = k^2 - l^2$, $y = 2kl$ in $z = k^2 + l^2$ pitagorejsko trojico. Izkaže se, da če pregledamo vse pare (k, l), pri katerih je $k > l$ in sta si k in l tuja in je eden od njiju sod, bomo dobili s formulo $(k^2 - l^2, 2kl, k^2 + l^2)$ ravno vse take pitagorejske trojice, pri katerih je x lih, y sod in števila x, y in z nimajo nobenega skupnega delitelja, večjega od 1.[81] (Največji skupni delitelj števil x, y in z označimo pogosto z $\gcd(x, y, z)$. Pitagorejski trojici, za katero velja $\gcd(x, y, z) = 1$, pravimo tudi *primitivna pitagorejska trojica*.) Lepo je tudi to, da ne bomo nobene take trojice dobili po večkrat, pač pa vsako natanko pri enem paru (k, l). Če nas zanimajo trojice s hipotenuzo $\le b$, mora biti $k^2 + l^2 \le b$ in zato $k \le \sqrt{b}$. Zdaj torej ni treba drugega, kot da z dvema gnezdenima zankama pregledamo vse primerne pare (k, l) do $k = \lfloor \sqrt{b} \rfloor$.

Rekli smo, da z gornjo formulo dobimo vse take trojice (x, y, z), pri katerih je x lih, y sod in $\gcd(x, y, z) = 1$. Kako bomo dobili še ostale trojice? Če vsako primitivno trojico štejemo dvakratno, smo s tem zajeli še (y, x, z), torej take trojice, pri katerih je x sod in y lih. Primitivnih trojic, ki bi imele x in y oba soda ali pa oba liha, pa ni; če bi bila oba soda, bi bil $z^2 = x^2 + y^2$ tudi sod, torej bi bil tudi z sod, torej bi imela števila x, y in z skupni delitelj 2 in trojica ne bi bila primitivna. O tem, da x in y ne moreta biti oba liha, pa

[81] Glej npr. Jože Grasselli, *Diofantske enačbe*, 1984, § 10.

se lahko prepričamo takole: kvadrat sodega števila $2t$ je oblike $4t^2$, kvadrat lihega števila $2t+1$ pa je oblike $4(t^2+t)+1$; ostanek po deljenju s 4 je torej vedno 0 ali 1; če bi bila x in y liha, pa bi imela vrednost x^2+y^2 po deljenju s 4 ostanek 2, torej to ne bi bil popoln kvadrat in z ne bi bil celo število.

Zdaj smo že našteli vse primitivne pitagorejske trojice. Če je (x', y', z') neprimitivna pitagorejska trojica, torej ima $\gcd(x', y', z') = d > 1$, lahko vsa tri števila delimo z njihovim skupnim deliteljem d in dobimo primitivno trojico $(x'/d, y'/d, z'/d)$. Neprimitivne trojice bomo torej upoštevali tako, da bomo vsako primitivno trojico šteli po večkrat, namreč po enkrat za vsak tak d, s katerim bi jo lahko pomnožili in še dobili primerno neprimitivno trojico. Iz primitivne trojice (x, y, z) dobimo (dx, dy, dz), ki je za naše namene primerna natanko tedaj, ko je $a \le dz \le b$, torej $\lceil a/z \rceil \le d \le \lfloor b/z \rfloor$. Trojico (x, y, z) moramo torej šteti $(\lfloor b/z \rfloor - \lceil a/z \rceil + 1)$-krat.

```
program PitagorejskeTrojice3;

  function gcd(u, v: integer): integer;
  begin
    while (u > 0) and (v > 0) do
      if u < v then v := v mod u
      else u := u mod v;
    gcd := u + v;
  end; {gcd}

var k, l, z, a, b, StTrojic, MaxK: integer;
begin
  ReadLn(a, b); n := 0;
  MaxK := Trunc(Sqrt(b));
  for k := 2 to MaxK do begin
    if Odd(k) then l := 2 else l := 1;
    while l < k do begin
      if gcd(k, l) = 1 then begin { Sta si k in l tuja? }
        z := k * k + l * l;
        if z > b then break; { l-ji od tu naprej so za ta k že preveliki. }
        StTrojic := StTrojic + 2 * (b div z - (a + z - 1) div z + 1);
      end; {if}
      l := l + 2;
    end; {while l < k}
  end; {for k}
  WriteLn(StTrojic);
end. {PitagorejskeTrojice3}
```

Za računanje največjega skupnega delitelja (funkcija gcd) smo uporabili znani Evklidov algoritem. Ta temelji na naslednjem razmisleku: naj bo $d = \gcd(u, v)$; torej je $u = u'd$, $v = v'd$. Recimo (brez izgube za splošnost), da je $u > v$. Naj bo $c = u \bmod v$; če je $c = 0$, pomeni, da je u večkratnik

števila v, torej je njun največji skupni delitelj kar v in je problem s tem že rešen. Sicer pa lahko razmišljamo takole: $c = u \bmod v$ lahko dobimo tako, da od u nekajkrat (natančneje: $(u \operatorname{div} v)$-krat) odštejemo v. Zato je vsak skupni delitelj števil u in v tudi delitelj števila c. Podobno lahko dobimo u tako, da c-ju nekajkrat (spet $(u \operatorname{div} v)$-krat) prištejemo v, zato je vsak skupni delitelj števil v in c tudi delitelj števila u. Ker imata torej u in v ravno iste skupne delitelje kot u in v, mora biti $\gcd(u, v) = \gcd(c, v) = \gcd(u \bmod v, v)$. Lepo pri tem je, da je $u \bmod v$ že po definiciji manjši od v, zato pa seveda tudi od u. Če bi zdaj ta razmislek večkrat ponovili, bi torej dobivali vse manjša števila, tako da se postopek gotovo ustavi po končno mnogo korakih (pokazati je mogoče, da je število teh korakov $O(\log u)$, če je u večje od obeh števil, pri katerih smo začeli[82]). Primer: $\gcd(12345, 678) = \gcd(141, 678) = \gcd(141, 114) = \gcd(27, 114) = \gcd(27, 6) = \gcd(3, 6) = \gcd(3, 0) = 3$.

Glavni del našega programa pregleda približno $\sqrt{b}$ vrednosti k in pri vsaki od njih približno $k/2$ vrednosti l, torej kliče podprogram gcd približno $b/4$-krat. Časovna zahtevnost celega programa je torej $O(b \log b)$.

Še ena rešitev iz teorije števil. V primerih, ko nas zanima le majhen razpon možnih vrednosti z, torej ko je a blizu b, je lahko prejšnja rešitev (PitagorejskeTrojice3) potratna, saj pregleda takrat še vedno prav toliko parov (k, l), kot če bi bil $a = 1$ in bi nas zanimale vse hipotenuze od 1 do b. Tudi tistih parov (k, l), ki dajo majhno vrednost $z = k^2 + l^2$, namreč ne smemo kar ignorirati, saj lahko iz take primitivne trojice mogoče dobimo kakšno primerno neprimitivno (torej tako, ki ima $a \le z \le b$), če jo pomnožimo z neko primerno konstanto d.

V takih primerih bi znala biti koristna naslednja formula. Razcepimo z na prafaktorje in naj bodo $d_1, \ldots, d_r$ stopnje ob tistih prafaktorjih, ki so oblike $4t + 1$. Število pitagorejskih trojic s hipotenuzo z je potem[83]

$$\triangle(z) := (2d_1 + 1)(2d_2 + 1) \cdots (2d_r + 1) - 1.$$

S to formulo ni težko priti do $\triangle(z)$, vprašanje je le, koliko časa porabimo za razcep z-ja na prafaktorje (da pridemo do eksponentov $d_1, \ldots, d_r$).

Faktorizacija (razcep na prafaktorje). Tega se lahko lotimo kar s poskušanjem — poskusimo deliti z po vrsti s praštevili 2, 3, 5, 7, 11, itd., pa bomo videli, katera od teh števil so res z-jevi prafaktorji in kakšno stopnjo imajo.

```
var Prast: array [1..StPrast] of integer; { Tabela praštevil, urejena naraščajoče. }
```

[82] Glej npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 33.2 v prvi izdaji, 31.2 v drugi.

[83] Glej npr. MathWorld *s. v.* "Pythagorean Triple"; *The On-Line Encyclopedia of Integer Sequences*, A046080; in str. 116–117, 140–142 v Albert H. Beiler, *Recreations in the Theory of Numbers*, Dover Pubs., 1966. Dokaz je npr. v G. H. Hardy, E. M. Wright, *An Introduction to the Theory of Numbers*, 5. izd., Oxford, 1980, §§ 16.9–16.10 (str. 241–243).

```
function StTrojic(z: integer): integer;
var i, p, St, Stopnja: integer;
begin
  i := 1; St := 1;
  while z > 1 do begin
    Stopnja := 0; p := Prast[i]; i := i + 1;
    while z mod p = 0 do
      begin z := z div p; Stopnja := Stopnja + 1 end;
    { Zdaj vemo, da se p pojavlja kot prafaktor v razcepu števila z
      s stopnjo Stopnja. Če je Stopnja = 0, pa p sploh ni z-jev prafaktor. }
    if (Stopnja > 0) and (p mod 4 = 1) then St := St * (2 * Stopnja + 1);
  end; {while}
  StTrojic := St - 1;
end; {StTrojic}
```

Ta postopek lahko še precej izboljšamo. Zunanja zanka se trenutno izvaja tako dolgo, dokler ne pade z na 1, to pa se zgodi šele, ko ga delimo s čisto vsemi prafaktorji, tudi z največjim. Najhuje je pri praštevilskih z, ko je z kar sam svoj edini prafaktor; izkaže pa se, da imajo tudi mnoga druga števila kakšen precej velik prafaktor. Ne more pa se zgoditi, da bi imel z dva prafaktorja, večja od $\sqrt{z}$ (ali pa enega tolikšnega, ki pa bi se pojavljal s stopnjo, večjo od 1), saj bi moral biti potem že samo produkt teh dveh večji od z. Če torej v zunanji zanki pridemo do praštevila p, ki je $> \sqrt{z}$, vemo, da je vse, kar je od z-ja ostalo, lahko samo še en sam prafaktor: tako smo z pravzaprav že povsem razcepili in lahko takoj nehamo.

```
function StTrojic2(z: integer): integer;
var i, p, St, Stopnja: integer;
begin
  i := 1; St := 1;
  while z > 1 do begin
    Stopnja := 0; p := Prast[i]; i := i + 1;
    if p * p > z then break;
    while z mod p = 0 do
      begin z := z div p; Stopnja := Stopnja + 1 end;
    { Zdaj vemo, da se p pojavlja kot prafaktor v razcepu števila z
      s stopnjo Stopnja. Če je Stopnja = 0, pa p sploh ni z-jev prafaktor. }
    if (Stopnja > 0) and (p mod 4 = 1) then St := St * (2 * Stopnja + 1);
  end; {while}
  { Spremenljivka z je zdaj enaka 1 ali pa vsebuje vrednost zadnjega
    (največjega) prafaktorja (ki mu v razcepu pripada stopnja 1). }
  if (z > 1) and (z mod 4 = 1) then St := St * (2 * 1 + 1);
  StTrojic := St - 1;
end; {StTrojic2}
```

Kolikokrat se zdaj izvede zunanja zanka? Naj bo q največji prafaktor števila z; (1) če ima q v razcepu z-ja stopnjo, večjo od 1, je $z \geq q^2$, tudi če smo z že

delili z vsemi njegovimi ostalimi prafaktorji; torej bo zunanja zanka prišla do prafaktorja q, ne da bi bil dotlej kdaj izpolnjen pogoj `if p * p > z`. Ko pa pride zunanja zanka do q, bo imela spremenljivka `z` po koncu te iteracije zunanje zanke vrednost 1, saj je bil q še zadnji prafaktor, s katerim ga doslej še nismo delili. Tako se bo zunanja zanka končala, ker ne velja več $z > 1$. (2) Druga možnost pa je, da ima q v razcepu z-ja stopnjo 1; naj bo q' drugi največji prafaktor; potemtakem, ko pride zunanja zanka do q' in opravi tudi s tem prafaktorjem, ostane v spremenljivki `z` vrednost q; zdaj bo torej pogoj `if p * p > z` zagotovil, da zunanja zanka ne bo šla do večjih praštevil, kot je $\sqrt{q}$. — Obe možnosti lahko združimo v eno samo ugotovitev: če je q največji prafaktor števila z, število q' pa največji prafaktor števila z/q (če je $z = q$, si mislimo $q' = 1$), bo zunanja zanka pregledala vsa praštevila, manjša ali enaka $\max\{q', \sqrt{q}\}$. V najslabšem primeru bomo torej pregledali vsa praštevila do $\sqrt{z}$, večjih pa gotovo ne.

Koliko pa sploh je praštevil, ki so $\le t$ za nek dani t? To vrednost v teoriji števil označujejo s $\pi(t)$; izkaže se, da je $\pi(t) \approx t/(\ln t - 1)$. V našem postopku za faktorizacijo se torej zunanja zanka izvede največ $\pi(\sqrt{z})$-krat.[84] Notranja zanka pa se izvede le pri tistih praštevilih p, ki so res z-jevi prafaktorji; če ima nek prafaktor q_i v razcepu z-ja stopnjo d_i, se bo notranja zanka tam izvedla d_i-krat. Če je celoten razcep oblike $z = \prod_i q_i^{d_i}$, sledi (ker so vsa praštevila ≥ 2), da je $z \ge \prod_i 2^{d_i} = 2^{\sum_i d_i}$, torej je $\sum_i d_i$, skupno število izvajanj notranje zanke, gotovo $\le \lg z$. Tako lahko torej zaključimo, da je časovna zahtevnost našega postopka za faktorizacijo približno $O(\pi(\sqrt{z}) + \lg z) = O(\sqrt{z}/\ln z)$. Če moramo ta postopek opraviti na $n = b - a + 1$ različnih z-jih z intervala $a \le z \le b$, bo skupna časovna zahtevnost $O(n\sqrt{b}/\ln b)$.

Preden začnemo razmišljati o tem, kako si lahko pripravimo tabelo praštevil, ki jih gornji podprogram potrebuje pri faktorizaciji, lahko še omenimo, da ni nujno gledati res samo praštevil. Postopek bi še vedno deloval, četudi bi bilo v tabeli `Prast` tudi kakšno sestavljeno število; pomembno je le, da ne manjka v njej nobeno praštevilo in da je ta tabela še vedno urejena naraščajoče. To nam zagotavlja, da, če pridemo v tej tabeli do nekega sestavljenega števila p, smo pred tem videli v njej že tudi vse p-jeve prafaktorje,

[84] Ta ocena se zdi mogoče nekoliko pesimistična, saj smo videli, da moramo iti le po vseh praštevilih do $\max\{q', \sqrt{q}\}$, pri čemer je q največji prafaktor z-ja, q' pa največji prafaktor z/q. Toda števil z velikimi prafaktorji ni tako zelo malo. Izkaže se (Knuth, *The Art of Computer Programming*, vol. 2, § 4.5.4), da je pri približno polovici z-jev največji prafaktor $q \ge z^{0,6065}$. Praštevil do $\sqrt{q}$ je v tem primeru $\pi(\sqrt{q}) = \Omega(z^{0,3033}/\ln z)$; že samo zaradi takih z-jev bi torej opisani postopek faktorizacije zahteval povprečno $\Omega(z^{0,3053}/\ln z)$ deljenj. Lahko pa bi namesto z $z^{0,6056}$ začeli tudi z npr. $q \ge z^{0,9512}$, do česar pride pri približno 5 % števil z; tako bi dobili zahtevnost $\Omega(z^{0,4756}/\ln z)$ in na podoben način tudi $\Omega(z^{1/2-\varepsilon}/\ln z)$ za poljuben $\varepsilon > 0$, le konstante, skrite v asimptotičnem zapisu $\Omega(\cdot)$, bi bile vse večje (ker se tolikšni prafaktorji pojavljajo pri vse manjšem deležu z-jev). Skratka, zaključimo lahko, da je trditev o $\pi(\sqrt{z})$ deljenjih sicer res malo pesimistična, vendar asimptotično ni kaj posebej pesimistična.

tako da v z-ju sploh ni ostal nobeden od teh prafaktorjev in zato z tudi s p-jem ne more biti deljiv. Naš program bi to pač opazil in se potem mirno posvetil naslednjemu delitelju iz tabele Prast. Lepo pri tem je, da se nam ni treba posebej ukvarjati z iskanjem praštevil; pravzaprav tabele Prast sploh ne potrebujemo več, saj si lahko delitelje računamo tudi sproti. Lahko bi na primer vzeli 2 in nato vsa liha števila; lahko pa to še malo izboljšamo, npr. začnemo z $2, 3, 5$, nato pa izmenično povečujemo p za 2 in 4 ter se tako izognemo večkratnikom števila 3: $5, 7, 11, 13, 17, 19, 23, 25, 29, 31$, itd. Slabost te rešitve pa je, da nimamo opravka le s praštevili do $\sqrt{z}$ (ki jih je le $\pi(\sqrt{z})$), pač pa tudi z nekaterimi sestavljenimi števili, tako da se število izvajanj zunanje zanke, ki je bilo prej $O(\sqrt{z}/\ln z)$, povzpne na $O(\sqrt{z})$.

Odkrivanje praštevil. Če nas bodo zanimali z-ji do največ $z = b$, potrebujemo praštevila do največ $m := \sqrt{b}$. Če m ni prevelik, lahko uporabimo kar Eratostenovo rešeto ali kakšno od njegovih različic. Eratostenovo rešeto temelji na zamisli, da si napišemo vsa števila od 2 do m, nato pa ponavljamo naslednje: najmanjše napisano število je gotovo praštevilo, vsi njegovi večkratniki pa so seveda sestavljena števila in jih lahko pobrišemo (oz. prečrtamo). Ko smo se torej praštevila in njegovih večkratnikov znebili, lahko na preostalih številih ponovimo isti korak in tako nadaljujemo, pa bomo sčasoma dobili vsa praštevila do m.

```
var Prast: array [1..MaxPrast] of integer;
    StPrast: integer;

procedure EratostenovoReseto(m: integer);
var i, j: integer; Precrtano: array [2..MaxM] of boolean;
begin
  StPrast := 0; for i := 2 to m do Precrtano[i] := false;
  for i := 2 to m do if not Precrtano[i] then begin
    StPrast := StPrast + 1; Prast[StPrast] := i;  { i je praštevilo. }
    j := 2 * i; while j <= m do                   { Prečrtajmo i-jeve večkratnike. }
      begin Precrtano[j] := true; j := j + i end;
  end; {if, for i}
end; {EratostenovoReseto}
```

Ta postopek lahko še izboljšamo na razne načine. Za tabelo Precrtano bi zadostovalo že m bitov, ne pa m Booleanov, ki so mogoče (odvisno od prevajalnika) dolgi vsak vsaj po en zlog (byte). Lahko bi tudi v tej tabeli hranili le podatke o lihih številih, soda pa obravnavali kot poseben primer, saj naš podprogram že zdaj takoj na začetku razglasi 2 za praštevilo in prečrta vsa ostala soda števila. Še ena izboljšava: pri gornjem podprogramu dobiva j vrednosti $2i, 3i, \ldots$, vendar v resnici za vse te vrednosti do vključno $(i-1)i$ vemo, da imajo vsaj en delitelj, manjši od i (namreč $2, 3, \ldots, i-1$), zato pa tudi vsaj en prafaktor, manjši od i, torej smo jih morali prečrtati že, ko smo gledali večkratnike teh manjših prafaktorjev. Zato bi bilo dovolj, če bi začel j pri i^2, ne pa pri $2i$.

Iz tega tudi vidimo, da se lahko zunanja zanka ustavi, ko i preseže vrednost $\sqrt{m}$, saj odtlej ne more prečrtati nobenega dotlej neprečrtanega števila; vsa preostala še neprečrtana števila pa so praštevila in jih moramo le še zapisati v tabelo Prast. — Pri praštevilih $i > 2$, ki so gotovo liha, velja tudi, da so $i(i+1)$, $i(i+3)$ itd. soda števila, torej ni treba, da se j ukvarja z njimi (prečrtali smo jih že kot večkratnike števila 2): lahko se povečuje kar po $2i$, ne po i.

Kakšna je časovna zahtevnost tega postopka? Ko je zunanja zanka pri praštevilu i, se notranja zanka izvede približno m/i-krat. Skupno število izvajanj notranje zanke je torej $m(1/2 + 1/3 + 1/5 + 1/7 + 1/11 + \ldots + 1/P)$, če je P največje praštevilo, manjše ali enako $\sqrt{m}$. Izkaže se,[85] da je vsota v oklepajih približno enaka $\ln\ln P$, tako da ima celoten postopek $O(m \ln\ln m)$ računskih operacij.[86]

Za razcep števil do b bodo prišla prav praštevila do $\sqrt{b}$; da jih poiščemo z Eratostenovim rešetom, bomo torej potrebovali $O(\sqrt{b}\ln\ln b)$ časa. Če prištejemo k temu še čas vseh faktorizacij, vidimo, da bi celoten postopek štetja pitagorejskih trojic za vse z-je trajal $O(\sqrt{b}\ln\ln b + n\sqrt{b}/\ln b)$ časa. Pri zelo majhnih n pa bi bilo bolje, če bi se Eratostenovemu rešetu odpovedali in bi raje vsak z razcepili kar tako, da bi ga poskušali deliti z 2 in z lihimi števili do $\sqrt{z}$; to bi dalo skupaj časovno zahtevnost $O(n\sqrt{b})$. (Še bolje pa bi bilo uporabiti kakšno od izboljšanih različic Eratostenovega rešeta s časovno zahtevnostjo $O(m)$ ali celo le $O(m/\ln\ln m)$.)

Faktorizacija več števil hkrati. Če imamo dovolj pomnilnika za nekaj tabel z n celicami (po eno celico za vsak z, ki nas zanima), lahko postopek še malo izboljšamo. Doslej smo razmišljali o tem, da bi pri faktorizaciji delili vsak z z raznimi praštevili, lepo po vrsti, dokler ga povsem ne razcepimo. Vendar bi ga pri tem velikokrat poskušali deliti tudi s takimi, ki sploh niso njegovi prafaktorji; to je velika potrata, saj je praštevilo p na primer prafaktor samo

[85] Gl. npr. MathWorld *s. v.* "Prime Sums"; ohlapna izpeljava v Mairsonovem spodaj omenjem članku, str. 665a; natančnejša pa v Hardy in Wright, *op. cit.*, § 22.7, izrek 427 na str. 351).

[86] Vendar pa je mogoče Eratostenovo rešeto z raznimi prijemi še izboljšati. Gornji algoritem pregleda v notranji zanki vse i-jeve večkratnike od i^2 naprej; toda če smo za nek $k \geq i$ že ugotovili, da je sestavljen, nima smisla zdaj gledati števila ik, saj mora imeti k nekega delitelja i', manjšega od i, tako da je ik tudi večkratnik števila i' in smo ga morali že razglasiti za sestavljenega, ko smo gledali večkratnike števila i'. Dovolj je torej pregledati števila ik samo za tiste $k \geq i$, ki jih še nimamo označenih kot sestavljena števila. Za učinkovito implementacijo te zamisli bi morali celice tabele Precrtano povezati tudi v seznam (dvojno povezano verigo), iz katerega bi potem sproti brisali tista števila, ki jih prepoznamo kot sestavljena in jih prečrtamo. Tako izboljšan algoritem ima časovno zahtevnost $O(m)$, ker vsako sestavljeno število j prečrta le enkrat (takrat, ko je i njegov najmanjši prafaktor), ne pa (tako kot osnovna oblika Eratostenovega rešeta) po enkrat za vsak različen j-jev prafaktor (Harry G. Mairson, *Some new upper bounds on generation of prime numbers*, CACM 20(9):664–669, Sept. 1977; David Gries, Jayadev Misra, *A linear sieve algorithm for finding prime numbers*, CACM 21(12):999–1003, Dec. 1978). Možne so še nadaljnje izboljšave, ki zbijejo časovno zahtevnost na $O(m/\ln\ln m)$ (Paul Pritchard, *A sublinear additive sieve for finding prime numbers*, CACM 24(1):18–23, Jan. 1981).

vsakemu p-temu številu z. Namesto da bi obdelovali z-je enega za drugim (in pri vsakem izračunali, v koliko trojicah nastopa), lahko postavimo za začetek $\triangle[z] := 1$ za vse z, nato pa obdelujemo praštevila eno za drugim in ko vidimo, da se trenutno praštevilo pojavlja v razcepu nekega z s stopnjo d, pomnožimo $\triangle[z]$ z $2d+1$. Prave vrednosti $\triangle(z)$ se tako postopoma računajo v tej tabeli. Na koncu vse $\triangle[z]$ zmanjšamo za 1 in dobljene vrednosti seštejemo. Prihranek izvira iz tega, da moramo iti pri vsakem praštevilu p le po tistih z-jih, ki so njegovi večkratniki, ostale pa lahko preskočimo.

```
program MnozicnaFaktorizacija;

const StPrast = ...;
var Prast: array [1..StPrast] of integer; { Tabela praštevil, ki so ≤ √b. }
    zz: array [a..b] of integer; { zz[z] = tisto, kar je še ostalo od z-ja po tistem,
                                   ko smo ga delili z dosedanjimi praštevili. }
    StTrojic: array [a..b] of integer;   { V koliko trojicah nastopa posamezni z? }
    Rezultat: integer;                   { Skupno število trojic. }
    z, i, p: integer;
begin
  PripraviPrastevila; { npr. z Eratostenovim rešetom }

  for z := a to b do begin zz[z] := z; StTrojic[z] := 1 end;
  for i := 1 to StPrast do begin
    p := Prast[i];                       { i-to praštevilo. }
    z := ((a + p - 1) div p) * p;        { Najmanjši p-jev večkratnik, večji ali enak a. }
    while z <= b do begin
      Stopnja := 0;                      { Kakšna je stopnja p-ja v razcepu z-ja? }
      while zz[z] mod p = 0 do
        begin Stopnja := Stopnja + 1; zz[z] := zz[z] div p end;
      if p mod 4 = 1 then                { Je to tak prafaktor, ki vpliva na število trojic? }
        StTrojic[z] := StTrojic[z] * (2 * Stopnja + 1);
      z := z + p;                        { Pojdimo na naslednji večkratnik. }
    end; {while}
  end; {for i}

  Rezultat := 0;
  for z := a to b do begin
    if (zz[z] > 1) and (zz[z] mod 4 = 1) then
      StTrojic[z] := StTrojic[z] * 3;    { Še zadnji prafaktor z-ja. }
    Rezultat := Rezultat + StTrojic[z] - 1;
  end; {for z}
  WriteLn(Rezultat);
end. {MnozicnaFaktorizacija}
```

Tu moramo za vsako praštevilo, ki se pojavlja v nekem z-ju s stopnjo d, izvesti največ $d+1$ deljenj (zadnje je tisto, ki se ne izide), vendar le, če je $d > 0$. S praštevili, ki niso prafaktorji nekega z, pa tega z-ja ne bomo sploh nikoli

poskušali deliti. Naj bo $n := b - a + 1$; neko praštevilo p se pojavlja kot prafaktor v približno n/p možnih z-jih, od tega v približno n/p^2 s stopnjo 2 ipd. Če to seštejemo, imamo $\leq \sum_{d=1}^{\infty} n/p^d$ deljenj (parov operacij div in mod), kar je naprej enako $n/(p-1) = O(n/p)$. (To so bila deljenja, ki se izidejo, poleg tega pa je še n/p deljenj, ki se ne izidejo, tako da ostanemo pri $O(n/p)$.) To moramo potem sešteti po vseh praštevilih p, manjših od $\sqrt{b}$; vsota $1/p$ po teh praštevilih je približno $\ln\ln\sqrt{b}$, tako da dobimo skupaj časovno zahtevnost $O(\pi(\sqrt{b}) + n\ln\ln b)$. Če uporabimo za pripravo seznama praštevil kar osnovno obliko Eratostenovega rešeta, je skupna časovna zahtevnost $O((\sqrt{b}+n)\ln\ln b)$. To je bolje kot prej, cena za to pa je večja poraba pomnilnika.[87]

Če si tako velikih tabel ne moremo privoščiti in moramo kljub vsemu izračunati $\triangle(z)$ za vsak z v celoti, preden se lotimo naslednjega z, je dobro vsaj vedeti, da obstajajo tudi učinkovitejši (vendar bolj zapleteni) algoritmi za faktorizacijo (razcep na prafaktorje) od tega s poskušanjem in deljenjem z vsemi dovolj majhnimi praštevili.

Gornji program za množično faktorizacijo bi lahko izboljšali še tako, da bi hranil podatek o tem, koliko izmed trenutno opazovanih z-jev je že čisto razcepil; če je razcepil že vse, lahko zanko, ki pregleduje praštevila (**for** i v gornjem programu), takoj prekine. To lahko pride prav, če nas zanima le peščica zaporednih z-jev in to takih, ki bi imajo vsi same majhne prafaktorje (da nam ne bo treba pregledovati vseh praštevil do $\sqrt{b}$).

Zaključek. Če je $n = b - a + 1$ (število različnih vrednosti z-ja, ki nas zanimajo) zelo majhen, se lahko zgodi, da za iskanje praštevil z Eratostenovim rešetom porabimo več časa kot nato za faktorizacijo; takrat je zato najbolje, če faktoriziramo brez seznama praštevil (časovna zahtevnost: $O(n\sqrt{b})$). Lepo pri tem je tudi, da nam ne bo treba pripravljati vseh praštevil do $\sqrt{b}$, pač pa bomo pregledali le toliko deliteljev, kolikor jih je nujno potrebnih za faktorizacijo tiste peščice z-jev, ki nas zanimajo. Če nas na primer zanima en sam z in ima same majhne prafaktorje, ga bomo lahko razcepili zelo hitro; veliko deliteljev bomo morali pregledati le v primeru, če ima z velike prafaktorje (npr. če je kar praštevilo). Sejanje bi se pri zelo majhnih n splačalo le, če bi uporabili katerega od izboljšanih algoritmov za sejanje; do seznama praštevil bi lahko prišli z $O(\sqrt{b}/\ln\ln b)$ operacijami, za faktorizacijo pa bi jih nato porabili manj ($O(n\sqrt{b}/\ln b)$), če je n res dovolj majhen.

Ko gledamo večje vrednosti n, začne časovna zahtevnost faktorizacije prej

[87] Namesto tabel velikosti $n = b - a + 1$ bi zadostovale že tabele velikosti $\sqrt{b}$: če gledamo toliko zaporednih vrednosti z naenkrat, lahko potem pregledamo vsa praštevila do $\sqrt{b}$ in z vsakim delimo vse njegove večkratnike iz tistega intervala $\sqrt{b}$ zaporednih vrednosti z. Ker so naša praštevila manjša od $\sqrt{b}$, bo imelo vsako na tem intervalu gotovo vsaj en večkratnik in zato naše ukvarjanje s tem praštevilom ne bo odveč. Tako lahko razcepimo teh $\sqrt{b}$ zaporednih z-jev in se nato na enak način lotimo naslednjih $\sqrt{b}$ zaporednih z-jev. Časovna zahtevnost ostane taka, kot je bila, le prostorska se zmanjša — namesto $O(n)$ je le še $O(\min\{n, \sqrt{b}\})$.

n	Št. pitagorejskih trojic s hipotenuzo $\le n$ primitivne	vse trojice
10	1	4
100	16	104
1000	158	1 762
10^4	1 593	24 942
10^5	15 919	322 872
10^6	159 139	3 961 284
10^7	1 591 579	46 942 950
10^8	15 915 492	542 721 306
10^9	159 154 994	6 160 150 864
10^{10}	1 591 549 475	68 930 865 718
10^{11}	15 915 494 180	762 602 219 838
10^{12}	159 154 943 063	8 358 957 806 784
10^{13}	1 591 549 430 580	90 918 934 019 936
10^{14}	15 915 494 309 496	982 482 900 002 656

Ilustracija k rešitvi naloge 2001.1.4.
Ta tabela prikazuje število pitagorejskih trojic s hipotenuzo, manjšo ali enako n, za nekaj vrednosti n. Pokazati je mogoče, da je takih trojic približno $(n \ln n)/\pi$; če štejemo le take, ki so primitivne (števila v trojici so si tuja) in ne ločimo med (x, y, z) in (y, x, z), jih je približno $n/(2\pi)$.

---

ali slej prevladovati nad zahtevnostjo iskanja praštevil, tako da se potem vedno splača najprej poiskati praštevila do $\sqrt{b}$. (Meja, od katere naprej to drži, je odvisna od tega, kakšno različico sejanja in faktorizacije uporabljamo. Pri faktorizaciji vsakega z posebej je korist od seznama praštevil večja kot pri množični faktorizaciji.) Dokler n ni prevelik, lahko faktoriziramo vse z-je (od a do b) naenkrat in je časovna zahtevnost tega postopka $O(n \ln \ln b)$. Takšna množična faktorizacija je praktično vedno hitrejša od postopka, pri katerem faktoriziramo vsak z posebej. Slednjega (s časovno zahtevnostjo $O(n\sqrt{b}/\ln b)$) se torej splača uporabiti le, če ne moremo ali nočemo žrtvovati $O(\min\{n, \sqrt{b}\})$ pomnilnika za pomožni tabeli pri množični faktorizaciji.

Pri dovolj velikih n postane konkurenčen tudi algoritem z naštevanjem vseh primitivnih pitagorejskih trojic, ki zahteva, kot smo videli, $O(b \ln b)$ računskih operacij. To je vsekakor hitreje od faktorizacije vsakega z posebej (kar ima zahtevnost $O(n\sqrt{b}/\ln b)$). Množična faktorizacija pa bi morala biti asimptotično sicer boljša ($O(n \ln \ln b)$), vendar v praksi ni nujno tako: pri naših poskusih z $n = b = 10^8$ je na primer rešitev z naštevanjem primitivnih trojic porabila 11,1 s, rešitev s faktorizacijo pa 69,9 s, torej 6,3-krat dlje. Rešitev s faktorizacijo bo torej hitrejša od tiste z naštevanjem primitivnih trojic šele pri tako velikih b, za katere bo razmerje $\ln b : \ln \ln b$ vsaj 6,3-krat večje kot pri $b = 10^8$ (in bo s tem izničilo začetno prednost rešitve z naštevanjem primitivnih trojic). Do tega pride šele pri $b = 3{,}3 \cdot 10^{92}$; takrat bi seveda porabila

oba algoritma nesprejemljivo veliko časa, algoritem s faktorizacijo pa tudi veliko preveč pomnilnika. V praksi je torej, če nas zanima velik n, verjetno najpametneje uporabiti algoritem z naštevanjem vseh primitivnih trojic.

REŠITVE NALOG ZA DRUGO SKUPINO

R2001.2.1 Iskalnik

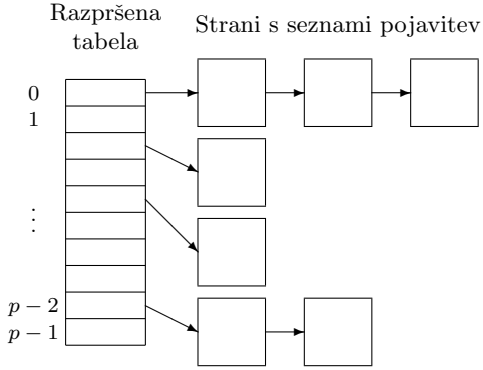
N: 431 Glede na vrsto poizvedb, ki jih želimo izvajati, je še najkoristneje, če datotekam dodelimo neke preproste številske oznake in si za vsako besedo pripravimo seznam datotek, v katerih se pojavlja. Te sezname bi seveda hranili na disku in pri vsaki poizvedbi naložili le tiste, ki se tičejo iskanih besed. Potem moramo organizirati le še nekakšno „kazalo“, s pomočjo katerega bomo lahko čim hitreje prišli do seznama za določeno besedo. V ta namen lahko uporabimo razpršeno tabelo ali pa kakšno drevesasto indeksno strukturo. Če je na voljo dovolj pomnilnika, lahko kazalo hranimo tudi v pomnilniku; ali pa imamo eno kazalo, ki je v pomnilniku in vsebuje pogosteje uporabljane besede, česar pa ne najdemo v njem, poiščemo potem v drugem kazalu, ki ga hranimo na disku in vsebuje vse preostale besede.

Oglejmo si na primer, kako bi naredili kazalo s pomočjo razpršene tabele. Potrebovali bomo „razprševalno funkcijo“ (*hash function*), ki vsaki besedi w pripiše neko celo število $h(w)$ med 0 in $p-1$. Na primer:

```
function H(S: string): integer;
var i, x: integer;
begin
  x := 0;
  for i := 1 to Length(S) do
    x := ((x * 256) + Ord(S[i])) mod p;
  H := x;
end; {H}
```

Datoteko, ki hrani sezname pojavitev besed, razdelimo na „strani“, velike po 1 KB ali kaj podobnega (gl. sliko na str. 459). Za vsako razpršilno kodo od 0 do $p-1$ imejmo po eno stran, ki vsebuje sezname pojavitev vseh besed s to razpršilno kodo (pri vsakem seznamu pa hranimo tudi besedo, na katero se nanaša). Če pri kakšni razpršilni kodi vsi ti seznami ne gredo na eno stran, ustanovimo zanje še dodatne strani in jih povežimo v verigo. Poleg teh strani pa potrebujemo še tabelo s p celicami, ki za vsako razpršilno kodo povedo številko prve strani s podatki za to razpršilno kodo. Ime „razpršena tabela“ izvira iz dejstva, da smo besede s pomočjo funkcije h „razpršili“ (upajmo, da čim bolj enakomerno) med števila $0, \ldots, p-1$.

Ko pri odgovarjanju na poizvedbe potrebujemo seznam pojavitev za neko besedo w, moramo samo izračunati $h(w)$ in se lotiti v veliki datoteki prebiranja strani, ki vsebujejo sezname za besede s to razpršilno kodo, dokler ne



najdemo tistega za besedo w ali pa pregledamo vse te sezname in ugotovimo, da besede w v našem kazalu sploh ni. Slabost tega pristopa je, da se lahko več besed preslika v isto razpršilno kodo in moramo zato, preden pridemo do seznama pojavitev besede, ki nas zanima, včasih prebrati še sezname pojavitev nekaj drugih besed, ki imajo isto razpršilno kodo kot beseda, ki nas zanima. Vendar pa ta problem ne bo prehud, če je le p dovolj velik in če funkcija h dovolj dobro razprši besede. Za besede, ki imajo zelo dolge sezname pojavitev, je mogoče koristno, če hranimo njihove sezname v neki pomožni datoteki, v glavni datoteki pa le položaj začetka posameznega seznama v tisti pomožni datoteki; tako sicer potrebujemo za branje takega seznama en dostop do diska več kot sicer, zato pa nam pri dostopu do drugih seznamov za isto razpršilno kodo ne bo treba prebirati takih dolgih seznamov.

Opisana zasnova razpršene tabele je koristna v primeru, ko hočemo tudi kasneje, med obratovanjem iskalnika, dodajati nove datoteke in brisati ali spreminjati podatke o obstoječih.[88] Če pa bi hoteli imeti le iskalnik za popolnoma statičen in vnaprej znan nabor datotek, nam niti ne bi bilo treba kompliciratiz razdelitvijo datoteke na strani, pač pa bi lahko v datoteko preprosto po vrsti zapisali vse sezname za besede z določeno razpršilno kodo, nato vse za besede z naslednjo razpršilno kodo in tako naprej.

Pri poizvedbi moramo pregledati sezname, ki pripadajo besedam, katerih prisotnost je uporabnik zahteval, in tiste, ki pripadajo besedam, ki jih je uporabnik prepovedal. Rezultat poizvedbe so datoteke, katerih oznake se pojavljajo na vseh seznamih iz prve in na nobenem od seznamov iz druge skupine. Te preseke in razlike seznamov lahko najbrž računamo kar v pomnilniku, saj se

[88] Morebitna slabost opisane podatkovne strukture je tudi v tem, da potrebujemo za vsako razpršilno kodo od 0 do $p-1$ vsaj eno stran, tudi če ji pripada npr. le ena beseda in je zato prostor na tisti strani večinoma neizkoriščen. Do boljše izkoriščenosti prostora lahko pridemo, če dovolimo, da si več razpršilnih kod deli isto stran; dobro znan sistematičen pristop k temu je na primer *extendible hashing* (R. Fagin *et al.*, *Extendible hashing — a fast access method for dynamic files*, ACM TODS 4(3):315–344, September 1979).

posamezna beseda ne pojavlja v zelo veliko datotekah (take, ki se, pa iskalniki običajno tako ali tako ignorirajo). Preseke in razlike bo lažje računati, če bodo seznami številk datotek za posamezne besede urejeni naraščajoče; potem lahko uporabimo zlivanje in seznamov niti ni treba v celoti nalagati v pomnilnik.

Vhod: besede $w_1, \ldots, w_n$, ki morajo nastopati v iskani datoteki,
in besede $w_{n+1}, \ldots, w_m$, ki ne smejo nastopati v iskani datoteki.
Izhod: seznam L z oznakami vseh datotek, ki ustrezajo iskalnim pogojem.

1 Naj bo L_i seznam oznak vseh datotek, v katerih nastopa beseda w_i.
Pripravi se na branje seznamov $L_1, \ldots, L_m$ z diska.
$L :=$ prazen seznam.
2 Dokler nismo v celoti prebrali vseh seznamov $L_1, \ldots, L_n$:
3 Za vsak L_j $(j = 1, \ldots, m)$, ki ga še nismo prebrali v celoti,
poglej trenutno številko datoteke iz tega seznama;
naj bo d najmanjša izmed teh številk.
4 Če se d pojavlja v kakšnem L_j, $j \leq n$, in v nobenem L_j, $j > n$,
jo dodaj v seznam L.
5 Premakni se naprej po vseh tistih seznamih L_j, pri katerih je
trenutna beseda ravno d.

Če ustreza iskalnim pogojem zelo veliko datotek, lahko naredimo podobno kot mnogi spletni iskalniki: omejimo se na prikaz prvih sto ali tisoč zadetkov in glavno zanko gornjega postopka prekinemo, čim postane seznam L dovolj dolg.

Naš iskalnik bi lahko še izboljšali, na primer s podporo iskanju po frazah. Tako bi lahko uporabnik zahteval, da mora datoteka ne le vsebovati določene besede, ampak tudi, da morajo nastopati neposredno ena za drugo. Ena možnost je, da za začetek poiščemo datoteke, ki sploh vsebujejo vse besede iz fraze, nato pa te datoteke preberemo in za vsako posebej še preverimo, če vsebuje tudi frazo. To je lahko neučinkovito, če veliko besed vsebuje vsako besedo posebej, malo pa celo frazo; v tem primeru bi bilo bolje, če bi sezname pojavitev besed v datotekah dopolnili še s podatki o tem, kje v datoteki se beseda pojavlja. Besede v datoteki lahko kar oštevilčimo od 1 naprej in v seznamu za vsako pojavitev navedemo, katera po vrsti je ta beseda v tisti datoteki. Slabost te rešitve je, da če se neka beseda v neki datoteki pojavlja po večkrat, moramo zdaj našteti vse te pojavitve, medtem ko je prej zadostoval že podatek, da se ta beseda pač pojavlja v datoteki (ne glede na število pojavitev). Druga možnost je, da vsako datoteko v mislih razdelimo na recimo 32 delov in jo predstavimo z 32-bitno karto, v kateri je posamezni bit prižgan, če se beseda pojavlja v tisti dvaintridesetini datoteke. Dve besedi se v datoteki pojavljata kot fraza le, če se druga pojavlja v isti ali pa v naslednji dvaintridesetini kot prva; tako dobimo poceni preverljiv potreben (čeprav še ne tudi zadosten) pogoj za prisotnost fraze v datoteki. Na podoben način bi lahko uporabniku pustili tudi zahtevati, da se določena fraza v datoteki ne sme

pojavljati, le da moramo biti zdaj malo previdnejši in posamezne datoteke ne smemo zavrniti kot morebitnega zadetka, dokler nismo res povsem zanesljivo prepričani, da vsebuje kakšno od prepovedanih fraz (ne pa npr. le posameznih besed iz nje).

Še ena pomembna slabost našega doslej opisanega iskalnika pa je, da smo zanemarili problem razvrščanja rezultatov. Če uporabnikovim iskalnim pogojem ustreza sto datotek, mi pa bi mu jih radi za začetek prikazali le deset, katerih deset naj izberemo? Koristno je upoštevati, kje v datoteki se iskalne besede pojavljajo (če se iskalna beseda pojavlja v naslovu datoteke, so možnosti, da bo ta datoteka za uporabnika zanimiva, najbrž večje, kot če bi se pojavljala le v navadnem besedilu); spletni iskalniki pa poskušajo uporabiti tudi povezave med stranmi, da ocenijo, katere strani so že same po sebi videti pomembnejše ali zanimivejše.[89]

R2001.2.2 **Psevdo-tetris**

Igralni površini lahko določimo koordinatni sistem: vrstice oštevilčimo od 0 (zgoraj) do $Y_P - 1$ (spodaj), stolpce od 0 (levo) do $X_P - 1$ (desno). Položaj lika na igralni površini je torej določen s parom koordinat (X, Y). Ker lahko lik premikamo le navzdol, levo in desno, lahko določeno polje (X, Y) doseže le, če lahko najprej pride na neko polje $(X', Y - 1)$ eno vrstico više, nato se od tam premakne navzdol na (X', Y), torej v pravo vrstico, nato pa (če ni $X = X'$) s premiki levo ali desno pride do (X, Y). N: 432

Torej je koristno, preden ugotavljamo, katere koordinate (X, Y) so dosegljive za določen Y, vedeti, katere so dosegljive pri $Y - 1$. Na začetku lahko predpostavimo, da so dosegljive vse $(X, -1)$ za $0 \le X < X_P$, saj naloga pravi, da je lik sprva nad igralno površino in tam ga lahko premikamo levo in desno. Potem pa, če za $Y - 1$ že poznamo vse dosegljive X, lahko za vsakega od njih pogledamo, če se lahko lik od tam premakne za eno polje navzdol na nek (X, Y). Nato za vse tako dobljene pare (X, Y) pogledamo še, če se lahko lik s premikanjem na levo in/ali na desno iz tega položaja premakne še na kaj sosednjih mest. S tem smo ugotovili vse dosegljive položaje v vrstici Y. Podatke o dosegljivosti položajev v vrstici $Y - 1$ lahko nato pozabimo, saj jih ne bomo več potrebovali.

Ta postopek ponavljamo, dokler ne pridemo do $Y = Y_P$ (čim je dosegljiv kak položaj v tej vrstici, je lik padel skozi igralno površino; pravzaprav velja to že pri $Y = Y_P - 1$) ali pa do nekega Y, pri katerem ni dosegljiv noben X več (v tem primeru se lahko ustavimo in vrnemo Y: kajti če je dosegljivo neko polje v vrstici $Y - 1$, je razdalja od vrha igralne površine do spodnjega roba lika v tem primeru ravno Y).

[89] Nekaj zanimive literature: S. Brin, L. Page: *The anatomy of a large-scale hypertextual web search engine*, Computer Networks 30(1–7):107–117, April 1998; J. M. Kleinberg: *Authoritative sources in a hyperlinked environment*, JACM 46(5):604–632, September 1999.

Razmislimo še o razširjeni obliki te naloge, pri kateri lik, ki ga premikamo po igralni površini, ni nujno kvadratek 1×1, ampak je lahko tudi poljuben večji lik nad karirasto mrežo takih kvadratkov. Glavna razlika v primerjavi s primerom, ko je lik vedno kvadratek 1×1, je pri preverjanju, ali se lahko lik z določenega položaja premakne v določeno smer (levo, desno ali dol). Ena možnost je, da bi preprosto za vsako polno polje našega lika preverili, če se, ko bo lik na novem položaju, res ne bo prekrivalo z nobenim polnim poljem igralne površine. Vendar pa je lahko to neučinkovito, kajti premik je nemogoče le v primeru, če se rob lika pri premiku zaleti v kako polno polje igralne površine (ali pa v primeru, ko bi rob lika pogledal čez levi ali desni rob igralne površine, a tega pač ni težko preveriti). Zato je dovolj že, če pogledamo, ali se robna polja našega lika na novem položaju ne bodo prekrivala s polnimi polji igralne površine; kajti če je bilo neko robno polje tako pri starem kot pri novem položaju lika na praznem polju igralne površine, ima zdaj tudi prostor za premik s starega položaja na novega (saj gledamo le premike za eno enoto). Pri premikih na levo je treba gledati tista polna polja lika, ki na svoji levi mejijo na prazno polje; pri premikih desno oz. dol pa podobno le tista polna polja lika, ki na svoji desni oz. spodnji stranici mejijo na prazna polja. Če je lik neugodne oblike, je sicer takšnih robnih polj še vseeno lahko zelo veliko, pri mnogih likih pa je vendarle robnih polj malo v primerjavi z vsemi polnimi polji. Pametno bi si bilo vnaprej pripraviti sezname levih, desnih in spodnjih robnih polj in potem pri vsakem premiku gledati le polja z ustreznega seznama.

R2001.2.3 CD-predalček

N: 433 Program vodi podatek o zahtevani smeri gibanja predalčka. Zaznati mora, kdaj uporabnik pritisne tipko (če je zdaj pritisnjena, pri prejšnji meritvi pa še ni bila) in obrniti zahtevano smer. Če predalček pri svojem gibanju doseže končno lego, motor ugasnemo. Če je zahtevana sprememba smeri, poženemo motor v novi smeri. Poseben primer je še možnost, da motor stoji, predalček pa ni v končni legi; v tem primeru ga tudi poženemo (do tega lahko pride na začetku izvajanja programa, če je bil predalček ob zagonu predvajalnika na pol odprt; ker je IzbranaSmer na začetku Noter, se bo predalček zaprl).

```
program MotoriziraniPredalcek(Input, Output);

  type SmerT = (Stop, Noter, Ven);
  var IzbranaSmer, TrenutnaSmer: SmerT;
      TipkaNovo, TipkaStaro: boolean;
      Pogon: boolean;

  function TipkaPritisnjena: boolean; external;
  function Odprto: boolean; external;
  function Zaprto: boolean; external;
```

```
  procedure Motor(IzbranaSmer: SmerT); external;

begin {MotoriziraniPredalcek}
  IzbranaSmer := Noter; TrenutnaSmer := Noter;
  TipkaStaro := false;
  Motor(Stop); Pogon := false;
  repeat
    TipkaNovo := TipkaPritisnjena;
    if TipkaNovo > TipkaStaro then                { začetek pritiska tipke }
    begin { obrnimo izbrano smer }
      if IzbranaSmer = Noter then IzbranaSmer := Ven
                             else IzbranaSmer := Noter;
    end; {if}
    TipkaStaro := TipkaNovo;
    if ((TrenutnaSmer = Noter) and Pogon and Zaprto) or
      ((TrenutnaSmer = Ven)  and Pogon and Odprto) then
      begin Motor(Stop); Pogon := false end;    { zaustavitev v končni legi }
    if (IzbranaSmer <> TrenutnaSmer) or
      not (Zaprto or Odprto or Pogon) then
                          { sprememba smeri, ali pa ustavljeno v vmesni legi }
    begin { vključimo pogon v pravo smer }
      Motor(IzbranaSmer); Pogon := true;
      TrenutnaSmer := IzbranaSmer;
    end; {if}
  until false;
end. {MotoriziraniPredalcek}
```

R2001.2.4 3-D križci in krožci

N: 434

Vsako možno smer, v kateri lahko iščemo M enakih znakov v vrsti, lahko opišemo z vektorjem $(\triangle x, \triangle y, \triangle z)$, pri čemer je vsaka od teh komponent lahko -1, 0 ali 1 in nam pove, kako se spreminja ustrezna koordinata. Vse te smeri lahko oštevilčimo: $(\triangle x, \triangle y, \triangle z)$ predstavimo s številom $j = 9(\triangle x + 1) + 3(\triangle y + 1) + (\triangle z + 1)$. Tako jih lahko tudi naštejemo. Da ne bi kdaj šteli v neko smer in še v nasprotno smer, ne naštejemo vseh 27, ampak le prvih 13, torej tiste s številkami od 0 do 12. Iz formule za j namreč takoj sledi, da ima nasprotna smer, $(-\triangle x, -\triangle y, -\triangle z)$, številko $26 - j$, tako da bi, če bi gledali kakšno od smeri s številkami 14..26, videli iste skupine enakih znakov kot pri eni od smeri s številkami 0..12, le da v nasprotni smeri. Smer 13 pa se nanaša na vektor $(0, 0, 0)$, pri katerem se torej položaj sploh ne premika in nas ta smer zato ne zanima. Nato se pomikamo z začetno koordinato (X_0, Y_0, Z_0) po celi kocki in vsakič pogledamo, če lahko v opazovani smeri najdemo dovolj enakih znakov. Če pri tem pogledamo čez rob kocke, si mislimo, da smo naleteli na napačen znak (podprogram JeZnak).

```
function Prestej(Kocka: KockaT; M: integer): integer;
```

```
  function JeZnak(X, Y, Z: integer; Znak: ZnakT): boolean;
  begin
    if (0 <= X) and (X < N) and (0 <= Y) and (Y < N) and
      (0 <= Z) and (Z < N) then JeZnak := (Kocka[X, Y, Z] = Znak)
                           else  JeZnak := False;
  end; {JeZnak}

var X, Y, Z, X0, Y0, Z0, DX, DY, DZ, Koliko, i, j: integer; Znak: ZnakT;
begin {Prestej}
  Koliko := 0;
  for j := 0 to 12 do begin
    DX := (j div 9) - 1; DY := ((j div 3) mod 3) - 1; DZ := (j mod 3) - 1;
    for X0 := 0 to N-1 do for Y0 := 0 to N-1 do for Z0 := 0 to N-1 do begin
      X := X0; Y := Y0; Z := Z0; Znak := Kocka[X, Y, Z]; i := 1;
      while i < M do begin
        X := X + DX; Y := Y + DY; Z := Z + DZ;
        if not JeZnak(X, Y, Z, Znak) then break;
        i := i + 1;
      end; {while}
      if i = M then Koliko := Koliko + 1;
    end; {for X0, Y0, Z0}
  end; {for j}
  Prestej := Koliko;
end; {Prestej}
```

REŠITVE NALOG ZA TRETJO SKUPINO

R2001.3.1 Števila v ogledalu

N: 437

Ta naloga je bila mišljena kot izredno lahka, čeprav se je potem izkazalo, da je imelo nekaj tekmovalcev z njo vseeno težave. Rešimo jo lahko na več načinov. Lahko bi na primer prebrali prvo vrstico vhodne datoteke v nek niz (spremenljivko tipa **string**), nato bi ta niz obrnili, izluščili iz njega obe števili in ju sešteli. Vsoto bi tolikokrat delili z 10, da se bi več končala na števko 0, nato pa lahko vsoto zapišemo v nek niz, ga obrnemo in končno izpišemo v izhodno datoteko.

Malo elegantnejšo rešitev dobimo, če namesto obračanja nizov obračamo kar števila (v spremenljivkah tipa integer). Če zapišemo število n v desetiškem zapisu, ima njegova najbolj desna števka vrednost n mod 10, preostanek števila pa n div 10. Če bi takemu številu na desni pripisali neko novo števko, recimo d, bi se vrednost celega števila spremenila v $10n+d$. Podprogram Obrni v spodnji rešitvi si pomaga s temi dejstvi, da iz danega števila n pobira števke od desne proti levi in jih dodaja na konec števila r, ki zato nazadnje vsebuje ravno n-jeve števke v nasprotnem vrstnem redu (razen morebitnih ničel na koncu n-ja, ki se pri tej pretvorbi izgubijo, kar je dobro, saj naloga prav to tudi zahteva).

```
program StevilaVOgledalu;

  function Obrni(n: integer): integer;
  var r: integer;
  begin
    r := 0;
    while n > 0 do begin
      r := r * 10 + n mod 10; { Pripišimo r-ju zadnjo števko n-ja. }
      n := n div 10;          { Pobrišimo iz n-ja njegovo zadnjo števko. }
    end; {while}
    Obrni := r;
  end; {Obrni}

var T: text; a, b: integer;
begin {StevilaVOgledalu}
  { Preberimo dve števili iz vhodne datoteke. }
  Assign(T, 'addrev.in'); Reset(T); ReadLn(T, a, b); Close(T);
  { Izračunajmo rezultat in ga izpišimo v izhodno datoteko. }
  Assign(T, 'addrev.out'); Rewrite(T);
  WriteLn(T, Obrni(Obrni(a) + Obrni(b))); Close(T);
end. {StevilaVOgledalu}
```

Če pa bi morali delati z zelo velikimi števili, bi lahko števila ves čas hranili kar kot nize, za seštevanje pa bi simulirali postopek, ki ga uporabljamo tudi pri ročnem seštevanju števil: najprej seštejemo enice, nato desetice, stotice in tako naprej, pri tem pa ves čas upoštevamo še morebitni prenos s prejšnjega mesta. Ker je vrstni red števk pri tej nalogi obrnjen, se moramo pri seštevanju premikati po nizih od leve proti desni, ne pa od desne proti levi, kot je sicer navada pri ročnem seštevanju. Posebej moramo paziti še na to, da poreženo morebitne ničle z začetka niza, ki predstavlja vsoto danih dveh števil.

```
program StevilaVOgledalu2;
var T: text; S, R: string; i, j, Prenos: integer;
begin
  { Preberimo obe števili. }
  Assign(T, 'addrev.in'); Reset(T); ReadLn(T, S); Close(T);
  j := 1; while S[j] <> ' ' do j := j + 1; { Poiščimo presledek med številoma. }
  { Med seštevanjem bo i kazal na trenutno števko prvega, j pa drugega števila. }
  i := 1; j := j + 1;
  S := S + ' '; { Tako bo tudi za drugim številom prišel v nizu S presledek. }
  { Seštejmo števili; vsoto pripravimo v nizu R. }
  Prenos := 0; R := '';
  while (S[i] <> ' ') or (S[j] <> ' ') or (Prenos > 0) do begin
    if S[i] <> ' ' then
      begin Prenos := Prenos + Ord(S[i]) - Ord('0'); i := i + 1 end;
```

```
      if S[j] <> ' ' then
        begin Prenos := Prenos + Ord(S[j]) - Ord('0'); j := j + 1 end;
      R := R + Chr(Ord('0') + Prenos mod 10); Prenos := Prenos div 10;
      if R = '0' then R := ''; { Sproti režimo ničle na začetku. }
    end; {while}
    if R = '' then R := '0'; { Poseben primer, če je vsota 0. }
    { Izpišimo rezultat. }
    Assign(T, 'addrev.out'); Rewrite(T); WriteLn(T, R); Close(T);
end. {StevilaVOgledalu2}
```

R2001.3.2 Oklepajski izrazi

N: 437 **Rešitev z naštevanjem vseh izrazov:** naštejemo vseh 2^{2N} nizov, ki jih sestavlja $2N$ znakov, samih oklepajev in zaklepajev. Nize predstavimo kar z $2N$-bitnimi števili (enice predstavljajo oklepaje, ničle zaklepaje). Ko se pomikamo po nizu, spremljamo globino gnezdenja oklepajev; nikoli ne sme priti pod 0 (to bi pomenilo, da smo so bili že vsi oklepaji zaprti z ustreznimi zaklepaji, nato pa smo prebrali še en zaklepaj) in na koncu mora biti enaka 0 (da se vsi oklepaji zaprejo z ustreznimi zaklepaji).

```
program OklepajskiIzrazi1;
var i, N, Koliko: longint; Bit, Globina: integer;
begin
  { Izpišimo kar rezultate za vse N.
    Ta program je tako ali tako prepočasen za oddajo. }
  for N := 0 to 15 do begin
    { Preglejmo vsa zaporedja 2N bitov. }
    Koliko := 0;
    for i := 0 to (longint(1) shl (2 * N)) - 1 do begin
      { Preverimo, če globina gnezdenja kdaj pade pod 0. }
      Globina := 0; Bit := 0;
      while (Bit < 2 * N) and (Globina >= 0) do begin
        if ((i shr Bit) and 1) = 1 then Globina := Globina + 1
        else Globina := Globina - 1;
        Bit := Bit + 1;
      end; {while}
      { Na koncu pa mora biti globina gnezdenja enaka 0. }
      if Globina = 0 then Koliko := Koliko + 1;
    end; {for i}
    WriteLn('N = ', N, ', število izrazov: ', Koliko);
  end; {for N}
end. {OklepajskiIzrazi1}
```

To rešitev lahko še izboljšamo, če si za npr. vse možne skupine desetih bitov vnaprej potabeliramo, koliko se po celi skupini spremeni globina gnezdenja in

kakšna je najgloblja točka, ki jo znotraj skupine dosežemo. Tako lahko za vsak niz zelo hitro ugotovimo, ali je prave oblike ali ne (tak program lahko pregleda vse nize pet- do desetkrat hitreje kot gornja naivna implementacija). Gornji program, ki je brez tovrstnih izboljšav, bi se verjetno izvajal predolgo, da bi ga lahko tekmovalec v tej obliki oddal in bi mu ga sprejeli kot uspešno rešitev; lahko pa bi ga tekmovalec pognal na svojem računalniku in oddal program, ki ima pravilne rešitve za vseh petnajst možnih vrednosti N definirane kar kot konstante. Spodaj je izboljšana različica te rešitve.

```
program OklepajskiIzrazi2;
var i, j, j10, CikCak, N, Koliko: longint; Bit, Globina: integer;
    Dvig, Dno: array [0..1023] of integer;
begin
  { Za vsa zaporedja desetih bitov izračunajmo
    spremembo v globini gnezdenja in najnižjo globino. }
  for i := 0 to 1023 do begin
    Globina := 0; Dno[i] := 0;
    for Bit := 0 to 9 do begin
      if ((i shr Bit) and 1) = 1 then Globina := Globina + 1
      else Globina := Globina - 1;
      if Globina < Dno[i] then Dno[i] := Globina;
    end; {for Bit}
    Dvig[i] := Globina;
  end; {for i}
  { Izpišimo kar rezultate za vse N. }
  for N := 0 to 15 do begin
    { Preglejmo vsa zaporedja 2N bitov. Ker hočemo vsako tako
      zaporedje razbiti na tri kose po 10 bitov, mu bomo na
      koncu dodali cikcakast vzorec oklepajev — kot da bi
      dodali 15 - N parov „()“. To na veljavnost izraza ne vpliva. }
    Koliko := 0; CikCak := $55555555 shl (2 * N);
    for i := 0 to (longint(1) shl (2 * N)) - 1 do begin
      j := i or CikCak;
      { Preverimo, če globina gnezdenja kdaj pade pod 0. }
      j10 := j and 1023; { prvih 10 bitov }
      if Dno[j10] < 0 then continue;
      Globina := Dvig[j10];
      j10 := (j shr 10) and 1023; { drugih 10 bitov }
      if Globina + Dno[j10] < 0 then continue;
      Globina := Globina + Dvig[j10];
      j10 := (j shr 20) and 1023; { zadnjih 10 bitov }
      if (Globina + Dno[j10] >= 0) and (Globina + Dvig[j10] = 0) then
        Koliko := Koliko + 1;
    end; {for i}
    WriteLn('N = ', N, ', število izrazov: ', Koliko);
  end; {for N}
```

end. {*OklepajskiIzrazi2*}

Rešitev z rekurzivno formulo: opazimo, da je vsak oklepajski izraz oblike $(S)T$, kjer sta S in T spet oklepajska izraza (začne se torej z oklepajem, ki mu nekje pozneje pripada nek zaklepaj; niz S med njima, pa tudi niz T za zaklepajem, sta spet oklepajska izraza). Nemogoče je, da bi se nek oklepaj začel v S, pripadajoči zaklepaj pa bi imel šele v T; saj če si mislimo prvi tak oklepaj iz niza S, je jasno, da bi zaklepaj med S in T potem pripadal prav njemu, ne pa tistemu oklepaju pred nizom S, to pa je v nasprotju s tem, kako smo niza S in T sploh definirali. Iz tega pa sledi, da imajo vsi oklepaji, ki se začnejo v S, tudi svoje pripadajoče zaklepaje že v S, tako da je S res oklepajski izraz, T pa zato tudi. — Če je v celem nizu $(S)T$ recimo N oklepajev, v S pa M oklepajev, mora veljati $0 \le M < N$, v T pa mora biti $N-1-M$ oklepajev. Da naštejemo vse oklepajske izraze reda N, moramo torej upoštevati vse možne M in pri vsakem vse možne izbire nizov S in T. Naj $a(k)$ predstavlja število oklepajskih izrazov reda k; tako dobimo formulo $a(N) = \sum_{M=0}^{N-1} a(M)a(N-1-M)$. To je sicer mogoče izraziti tudi eksplicitno, $a(N) = (2N)!/[N!\,(N+1)!]$, vendar ne bi bilo s to obliko tule nič laže računati (števec in imenovalec, $(2N)!$ in $N!\,(N+1)!$, postaneta hitro prevelika za 32-bitne spremenljivke, tako da ju moramo bodisi sproti krajšati — kar se sicer dá, saj imata veliko skupnih faktorjev — ali pa zaupati številom s plavajočo vejico).

```
program OklepajskiIzrazi3;
var i, j, N: integer; Koliko: array [0..15] of longint;
begin
  Koliko[0] := 1;
  { Izpišimo kar rezultate za vse N. }
  for N := 1 to 15 do begin
    Koliko[N] := 0;
    for i := 0 to N - 1 do
      Koliko[N] := Koliko[N] + Koliko[i] * Koliko[N - 1 - i];
    WriteLn('N = ', N, ', število izrazov: ', Koliko[N]);
  end; {for N}
end. {OklepajskiIzrazi3}
```

Tabela na str. 469 kaže število oklepajskih izrazov reda N za prvih nekaj vrednosti N. Ta števila se imenujejo *Catalanova števila*; nanje naletimo še pri več drugih kombinatoričnih problemih, npr. pri štetju dvojiških dreves, triangulacij in lomljenih (cikcakastih) funkcij.[90]

[90] Glej npr. MathWorld *s. v.* "Catalan Number" in *The On-Line Encyclopedia of Integer Sequences*, A000108.

N	$a(N)$	N	$a(N)$	N	$a(N)$
0	1	7	429	14	2 674 440
1	1	8	1 430	15	9 694 845
2	2	9	4 862	16	35 357 670
3	5	10	16 796	17	129 644 790
4	14	11	58 786	18	477 638 700
5	42	12	208 012	19	1 767 263 190
6	132	13	742 900	20	6 564 120 420

Število oklepajskih izrazov reda N iz naloge 2001.3.2.

R2001.3.3 Parlament

Spodnja rešitev preprosto našteje vse koalicije in med njimi poišče najšibkejšo. N: 438
Če imamo N strank, lahko predstavimo koalicije z N-bitnimi števili, pri čemer vsak bit pove, ali je določena stranka v koaliciji ali zunaj nje. Potem moramo le našteti vsa števila od 0 do $2^N - 1$ in pri vsakem sešteti moči strank, ki so v koaliciji. Tako bomo lahko ugotovili, katera je najšibkejša večinska koalicija.

```
program Parlament1;
const MaxN = 20;
var i, N, MocVlade, MinMocVlade, Vsi: integer; j, jMin: longint;
    P: array [1..MaxN] of integer; T: text; Prva: boolean;
begin
  { Preberimo vhodne podatke. }
  Assign(T, 'parlamen.in');
  Reset(T); Read(T, N);
  for i := 1 to N do Read(T, P[i]);
  Close(T);
  { Koalicija vseh strank. }
  Vsi := 0;
  for i := 1 to N do Vsi := Vsi + P[i];
  jMin := (longint(1) shl N) - 1; MinMocVlade := Vsi;
  { Preizkusimo vse ostale (neprazne) koalicije. }
  for j := 1 to (longint(1) shl N) - 2 do begin
    MocVlade := 0;
    for i := 1 to N do
      if ((j shr (i - 1)) and 1) = 1 then
        MocVlade := MocVlade + P[i];
    if (MocVlade > Vsi - MocVlade) and (MocVlade < MinMocVlade) then
      { najšibkejša večinska koalicija doslej }
      begin MinMocVlade := MocVlade; jMin := j end;
  end; {for j}
  { Izpišimo rezultat. }
  Assign(T, 'parlamen.out'); Rewrite(T);
```

```
  Prva := true;
  for i := 1 to N do if ((jMin shr (i − 1)) and 1) = 1 then begin
    if Prva then Prva := false else Write(T, ' ');
    Write(T, i);
  end; {for i, if}
  Close(T);
end; {Parlament1}
```

Gornja rešitev deluje zadovoljivo hitro, ker imamo le dvajset strank. Lahko pa rešimo to nalogo tudi z dinamičnim programiranjem, po zgledu znanega problema s polnjenjem nahrbtnika. Če je vsega skupaj p poslancev, jih je treba za večino imeti vsaj $(p \operatorname{div} 2) + 1$; opozicija ima torej kvečjemu $(p-1) \operatorname{div} 2$.[91] Poiščimo najmočnejšo skupino strank, ki imajo vse skupaj največ $(p-1) \operatorname{div} 2$ glasov, pa bo komplement te skupine (torej množica vseh strank, ki niso v tej skupini) ravno najšibkejša možna večina. Stranke lahko v mislih dodajamo eno po eno in vsakič tvorimo vse možne manjšinske skupine. Ko dodamo novo stranko, obdržimo vse dosedanje skupine, poleg tega pa iz vsake naredimo še eno novo skupino, v kateri je poleg dotedanjih še pravkar dodana stranka. Pri tem pa, če se zgodi, da dobimo več enako močnih skupin strank, ni treba hraniti več kot ene, saj nas ne zanimajo vse možne rešitve, ampak je dovolj ena sama. Zato imamo pri vsakem številu strank opravka z največ 5000 skupinami, saj pravi naloga, da poslancev ni več kot 10000. Da lahko učinkovito odkrivamo skupine z enako močjo, jih imamo urejene po naraščajoči moči.

```
program Parlament2;
const MaxN = 20; MaxPoslancev = 10000;
type
  TabelaI = array [0..(MaxPoslancev − 1) div 2] of integer;
  TabelaL = array [0..(MaxPoslancev − 1) div 2] of longint;
var
  P: array [1..MaxN] of integer; T: text; Prva: boolean;
  i, j1, j2, StKoal, StKoalPrej, Moc1, Moc2, Moc, MaxMoc, N: integer;
  Moci, MociPrej, MociTemp: ↑TabelaI;
  Koal, KoalPrej, KoalTemp: ↑TabelaL;
begin
  New(Moci); New(MociPrej); New(Koal); New(KoalPrej);
  { Preberi vhodne podatke. }
  Assign(T, 'parlamen.in');
  Reset(T); Read(T, N);
  for i := 1 to N do Read(T, P[i]);
  Close(T);
  { Zanimale nas bodo šibke koalicije — dovolj šibke,
    da so njihovi komplementi večinske koalicije. }
```

[91] Ker je $p-(\lfloor p/2 \rfloor+1) = p-\lfloor p/2 \rfloor-1 = \lceil p/2 \rceil-1 = \lceil p/2-1 \rceil = \lceil (p-2)/2 \rceil = \lfloor (p-1)/2 \rfloor$. Pri tem razmisleku smo upoštevali dejstvi, da je $p = \lfloor p/2 \rfloor + \lceil p/2 \rceil$ in $\lceil p/d \rceil = \lfloor (p+d-1)/d \rfloor$.

```
  MaxMoc := 0; for i := 1 to N do MaxMoc := MaxMoc + P[i];
  MaxMoc := (MaxMoc - 1) div 2;

  { Koalicije (no, pravzaprav je ena sama) 0 strank. }
  StKoal := 1; Moci↑[0] := 0; Koal↑[0] := 0;

  { Preglejmo večje koalicije. }
  for i := 1 to N do begin
    MociTemp := Moci; Moci := MociPrej; MociPrej := MociTemp;
    KoalTemp := Koal; Koal := KoalPrej; KoalPrej := KoalTemp;
    StKoalPrej := StKoal; StKoal := 0;

    { Zdaj so v MociPrej/KoalPrej vse nevečinske koalicije,
      v katerih nastopajo le stranke do vključno i - 1.
      Dodajmo koalicije, v katerih nastopa tudi stranka i. }
    j1 := 0; j2 := 0;
    while (j1 < StKoalPrej) or (j2 < StKoalPrej) do begin

      { Seznam koalicij hočemo imeti urejen po naraščajoči moči.
        Z j1 se sprehajamo po dosedanjem seznamu koalicij, z j2 pa
        po istem seznamu, le da je zamaknjen za P[i], kar bo ponazarjalo
        dodajanje stranke i v te koalicije. }
      if j1 < StKoalPrej then Moc1 := MociPrej↑[j1] else Moc1 := MaxMoc + 1;
      if j2 < StKoalPrej then Moc2 := MociPrej↑[j2] + P[i]
                         else Moc2 := MaxMoc + 1;
      if Moc1 < Moc2 then Moc := Moc1 else Moc := Moc2;
      if Moc > MaxMoc then break; { od tu naprej so le še premočne koalicije }

      { Dodajmo novo koalicijo in povečajmo števca j1 in/ali j2. }
      if Moc = Moc1 then Koal↑[StKoal] := KoalPrej↑[j1]
      else Koal↑[StKoal] := KoalPrej↑[j2] or (longint(1) shl (i - 1));
      Moci↑[StKoal] := Moc; StKoal := StKoal + 1;
      if Moc = Moc1 then j1 := j1 + 1;
      if Moc = Moc2 then j2 := j2 + 1;

    end; {while}
    if Moci↑[StKoal - 1] = MaxMoc then break;
  end; {for i}

  { Najšibkejša večinska koalicija je ravno komplement
    najmočnejše manjšinske koalicije. }
  Assign(T, 'parlamen.out'); Rewrite(T);
  Prva := true;
  for i := 1 to N do if (Koal↑[StKoal - 1] shr (i - 1)) and 1 = 0 then begin
    if Prva then Prva := false else Write(T, ' ');
    Write(T, i);
  end; {for i, if}
  Close(T); Dispose(Moci); Dispose(MociPrej); Dispose(Koal); Dispose(KoalPrej);
end. {Parlament2}
```

R2001.3.4 Kletke

N: 439 Mislimo si, da bi dodelili vsaki kletki neko barvo. Sprehodimo se po zemljišču po vrsticah od severa proti jugu, znotraj vsake vrstice pa od zahoda proti vzhodu, ter barvajmo celice. Če ima neka celica tako severni kot zahodni zid, predpostavimo začasno, da pripada neki novi, doslej še neznani kletki, in ji dodelimo novo barvo. Če ima zahodni zid, ne pa severnega, pripada isti kletki kot celica nad njo in dobi torej isto barvo kot le-ta. Podobno prevzame barvo od celice levo od sebe, če ima severni zid, ne pa zahodnega. Če pa nima ne severnega ne zahodnega zidu, sta kletki, katerima pripadata severna in zahodna soseda te celice, pravzaprav ena in ista kletka: če še nimata enake barve, ju zdaj združimo (eno od obeh barv prebarvajmo v drugo). V danem trenutku potrebujemo le podatke o barvah celic levo od trenutne celice v trenutni vrstici in nad ter desno od trenutne celice v predhodni vrstici; vse ostalo lahko sproti pozabljamo.

V spodnjem programu hrani tabela Kletke barve celic, Velikost ploščine kletk, StBarv pa število doslej dodeljenih barv (nekatere mogoče sploh ne predstavljajo več neke kletke, saj se je le-ta mogoče že zlila s kakšno drugo; take primere prepoznamo po ploščini, enaki 0). Na koncu preštejemo, koliko barv dejansko predstavlja kletke, ter poiščemo največjo in najmanjšo.

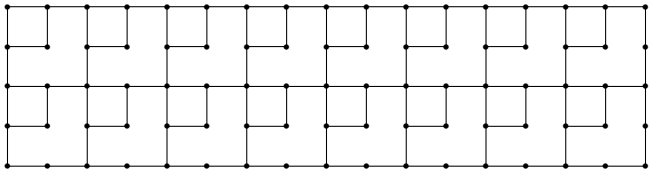
```
program Kletke1;
const MaxM = 100; MaxN = 100; MaxKletk = MaxM * MaxN;
      Zgoraj = 1; Levo = 8;
var
  { Velikost[k] = velikost (skupno število celic) kletke k }
  Velikost: array [1..MaxKletk] of integer;
  { Kletka[x] = kletka, ki ji pripada celica x }
  Kletka: array [1..MaxM] of integer;
  { statistike, ki jih bomo morali na koncu izpisati }
  Najvecja, Najmanjsa, StKletk, StBarv: integer;
  M, N: integer; { M = širina mreže, N = višina }
  T: text; Celica, y, x, xx, k, kk: integer;
begin
  { Preberimo velikost mreže. }
  Assign(T, 'kletke.in'); Reset(T); ReadLn(T, M, N);
  StBarv := 0;
  { Preglejmo celo mrežo. }
  for y := 1 to N do begin
    for x := 1 to M do begin
      Read(T, Celica); Celica := Celica and (Zgoraj or Levo);
      if Celica = 0 then
        { Celica ima zidova zgoraj in levo; ustanovimo zanjo novo kletko. }
        begin StBarv := StBarv + 1; k := StBarv end
      else if Celica = Zgoraj then
```

```
          k := Kletka[x] { Celica pripada isti kletki kot tista nad njo. }
        else if Celica = Levo then
          k := Kletka[x − 1] { Celica pripada isti kletki kot tista levo od nje. }
        else { Zlijmo kletko nad njo in tisto levo od nje
               (če nista to že zdaj ena in ista kletka). }
          if Kletka[x] <> Kletka[x − 1] then begin
            k := Kletka[x − 1]; kk := Kletka[x];
            for xx := 1 to M do if Kletka[xx] = kk then Kletka[xx] := k;
            Velikost[k] := Velikost[k] + Velikost[kk]; Velikost[kk] := 0;
          end; {if}
        { Trenutna celica pripada kletki k. }
        Kletka[x] := k; Velikost[k] := Velikost[k] + 1;
      end; {for x}
      ReadLn(T);
    end; {for y}
    Close(T);
    { Izračunajmo razne statistike. Kletke, ki so pri zlivanju postale
      del kakšne večje, imajo zdaj velikost 0 in jih ne smemo upoštevati. }
    Najvecja := 0; Najmanjsa := M * N; StKletk := 0;
    for k := 1 to StBarv do if Velikost[k] > 0 then begin
      StKletk := StKletk + 1;
      if Velikost[k] > Najvecja then Najvecja := Velikost[k];
      if Velikost[k] < Najmanjsa then Najmanjsa := Velikost[k];
    end; {for k}
    { Izpišimo rezultate. }
    Assign(T, 'kletke.out'); Rewrite(T);
    WriteLn(T, StKletk); WriteLn(T, Najmanjsa); WriteLn(T, Najvecja);
    Close(T);
end. {Kletke1}
```

Razmislimo še o časovni zahtevnosti tega postopka. Z vsako celico imamo konstantno veliko dela, razen če je treba izvesti zlivanje, tedaj pa imamo $O(m)$ dela. Kasneje imamo še konstantno mnogo dela z vsako barvo (tudi tistimi, ki smo jih z zlivanjem odpravili), barv pa je največ toliko kot celic. Nerodno je to, da lahko v najslabšem primeru do zlivanja pride pri $O(mn)$ celicah (glej primer na sliki na str. 474) in je zato časovna zahtevnost celotnega postopka v najslabšem primeru $O(m^2n)$. No, najmanj, kar bi lahko naredili, je to, da bi v primeru, ko je $m > n$, zamenjali vrstice in stolpce, kar bi dalo zahtevnost $O(mn \min\{m, n\})$.

Lahko pa bi uporabili znano podatkovno strukturo za disjunktne množice,[92] ki zagotavlja, da bo pri c celicah in $O(c)$ operacijah nad njimi skupna

[92] Več o podatkovnih strukturah za disjunktne množice je npr. v Cormen *et al.*, *Introduction to Algorithms*, 22. poglavje v prvi izdaji, 21. v drugi. Opisano mejo časovne zahtevnosti podaja izrek 21.13 na str. 517 v 2. izd.; v 1. izd. pa je dokazana malo ohlapnejša meja (izrek 22.7 na str. 455).



Primer mreže, kjer se izvaja zlivanje pri vsaki četrti celici.

časovna zahtevnost le $O(c\alpha(c))$, kjer je α neka zelo počasi naraščajoča funkcija, tako da lahko $\alpha(c)$ v praksi obravnavamo kot konstanto. Časovna zahtevnost za naš problem je tako pravzaprav $O(mn)$.

Osnovna zamisel te podatkovne strukture je, da pri zlivanju opravimo vedno le toliko dela, kolikor je nujno potrebno, ostalo pa odložimo za kasneje. Ko je treba zliti dve kletki, si zapomnimo za eno od njiju, da je postala „podkletka“ druge; celice, ki pripadajo novopečeni podkletki, pa pustimo pri miru in jih ne popravljamo. Zato postane zlivanje zelo poceni. Pač pa imamo zdaj malo več dela pri ugotavljanju, kateri kletki pripada neka celica: celica x sicer lahko pravi, da pripada neki kletki k, vendar je k mogoče medtem že postala podkletka neke druge, ta pa podkletka neke tretje in tako naprej. Zato moramo slediti tem kazalcem od kletke na „nadkletko“ (tabela NadKletka v spodnjem programu), dokler ne pridemo do neke take kletke k', ki ni podkletka nobene druge. Ko to enkrat naredimo, je vsekakor pametno vse kazalce na poti od celice x do kletke k' popraviti, tako da bodo kazali naravnost na k'; tako si lahko v bodoče prihranimo delo (podprogram NajdiKletko). Pri zlivanju je pametno vedno narediti manjšo kletko za podkletko večje, ne pa obratno, saj bomo s tem zagotovili, da gnezdenje podkletk ne bo šlo pregloboko. Za takšno podatkovno strukturo in opisani način uporabe se že da pokazati ugodno časovno zahtevnost iz prejšnjega odstavka.

Pazimo še na naslednje: ko neko celico, ki ji manjka levi (ali pa zgornji) zid, pridružimo h kletki, ki pokriva tudi njeno levo (ali pa zgornjo) sosedo, naj ta nova celica vedno pokaže na pravo kletko, ne pa na kakšno od njenih podkletk, tudi če njena soseda še kaže na kakšno podkletko.

Kot pri zgornjem programu bomo tudi zdaj vedno hranili le podatke o celicah levo od trenutne v trenutni vrstici in desno od trenutne v prejšnji vrstici. Ko torej obdelujemo celico (x, y), hrani Kletke[i] podatek o celici $(i, y-1)$, če je $i > x$, in o (i, y), če je $i \leq x$. Za vsako kletko lahko tudi vzdržujemo podatek o tem, koliko izmed teh celic trenutno kaže naravnost nanjo (in ne mogoče na kakšno njeno podkletko); to je tabela VTejVrsti v spodnjem programu. Ko ta števec pri neki kletki pade na 0 (ker smo se premaknili za eno vrstico niže in neka celica zdaj v resnici predstavlja spodnjo sosedo celice, ki jo je predstavljala doslej, in zato mogoče kaže na neko drugo kletko), lahko to kletko v mislih pobrišemo (njeno številko lahko kasneje ponovno uporabimo za kakšno novo kletko); če ni podkletka kakšne druge, lahko zdaj tudi povečamo globalni

števec kletk in mogoče popravimo podatka o velikosti največje in najmanjše kletke. Ni se treba bati, da bi pobrisali kletko, ki ima še kakšno podkletko. Recimo namreč, da bi se to nekoč vendarle zgodilo; naj bo torej k prva kletka, ki jo pobrišemo, ko ima še neko podkletko j. Toda od trenutka, ko sta se j in k zlili (in je j postala podkletka kletke k), je vsaj ena celica kazala na k (namreč tista (recimo (x, y), ki se hrani v Kletke[x]), pri kateri je do zlitja prišlo), in odtlej ni mogla nobena nova celica več mogla pokazati na j (saj smo rekli, da vedno, ko neki novi celici pripisujemo kletko, pokažemo na glavno kletko, ne pa na kakšno podkletko). Na j lahko torej kažejo le še kletke v preostanku trenutne vrstice ter v naslednji vrstici do pred tiste, ki leži tik pod našo (x, y); no, medtem ko obdelujemo te celice, se vsi kazalci na j v elementih tabele Kletke počasi spreminjajo v kazalce na kaj drugega — tudi če ostanejo v isti kletki, morajo kazalci po novem kazati na k in mogoče še na kakšno k-jevo nadkletko, če se tudi k s čim zlije. Ko torej pri obdelovanju trenutne in naslednje vrstice pridemo do celice $(x, y+1)$, smo se morali torej znebiti že vseh kazalcev na j, pri tem pa celica (x, y) sama še vedno kaže na k. Torej problem brisanja k-ja dotlej še ni mogel nastopiti, za povrhu pa bi do tega trenutka že zbrisali kletko j, saj smo se znebili vseh kazalcev nanjo. Tako smo prišli v protislovje: v resnici se ne more zgoditi, da bi morali zbrisati k, medtem ko bi imela ta še neko podkletko j.

Število kletk, ki so v danem trenutku „odprte“, je lahko največ M, saj smo videli, da mora na vsako kletko kazati vsaj en element tabele Kletke (kajti če ni tako, pade VTejVrsti[k] na 0 in bi kletko k pobrisali), ta pa ima le M elementov.

```
program Kletke2;
const MaxM = 100; Zgoraj = 1; Levo = 8;
var
  { Velikost[k] = velikost (skupno število celic) kletke k }
  Velikost: array [1..MaxM] of integer;
  { VTejVrsti[k] = koliko elementov tabele Kletka ima trenutno vrednost k }
  VTejVrsti: array [1..MaxM] of integer;
  { Kletka[x] = kletka, ki ji pripada celica x }
  Kletka: array [1..MaxM] of integer;
  { NadKletka[k] = kletka, katere del je postala kletka k med zlivanjem }
  NadKletka: array [1..MaxM] of integer;
  { sklad neuporabljenih številk kletk, torej takih,
    na katere ne kaže noben element tabele Kletka }
  StProstih: integer; Proste: array [1..MaxM] of integer;
  { statistike, ki jih bomo morali na koncu izpisati }
  Najvecja, Najmanjsa, StKletk: integer;
  { M = širina mreže, N = višina }
  M, N: integer;

  { Zmanjša VTejVrsti[k] za 1. Če pri tem pade na 0, kletko pobriše. }
  procedure ZmanjsajKletko(k: integer);
```

```
  begin
    VTejVrsti[k] := VTejVrsti[k] − 1;
    if VTejVrsti[k] > 0 then exit;
    if NadKletka[k] = k then begin
      { To je bila res samostojna kletka, ne pa del kakšne
        večje. Zato osvežimo statistike. }
      if Velikost[k] > Najvecja then Najvecja := Velikost[k];
      if Velikost[k] < Najmanjsa then Najmanjsa := Velikost[k];
      StKletk := StKletk + 1;
    end; {if}
    { Ta številka kletke je zdaj prosta in jo bomo lahko uporabili
      za kakšno novo kletko. }
    StProstih := StProstih + 1; Proste[StProstih] := k;
  end; {ZmanjsajKletko}

  { Pove, v katero kletko spada celica x, in popravi kazalce na
    poti do te kletke, da kažejo naravnost nanjo in ne na podkletke. }
  function NajdiKletko(x: integer): integer;
  var k, Nad, r: integer;
  begin
    r := Kletka[x];
    while NadKletka[r] <> r do r := NadKletka[r];
    { Prevežimo kazalce na poti do r, da bodo kazali naravnost na r.
      Tudi celica x naj odslej kaže naravnost na r. }
    k := Kletka[x];
    if k <> r then begin
      Nad := NadKletka[k]; ZmanjsajKletko(k); k := Nad;
      Kletka[x] := r; VTejVrsti[r] := VTejVrsti[r] + 1;
      while k <> r do
        begin Nad := NadKletka[k]; NadKletka[k] := r; k := Nad end;
    end; {if}
    NajdiKletko := r;
  end; {NajdiKletko}

  { Zlije dani kletki — manjša postane podkletka večje. }
  function ZlijKletki(k1, k2: integer): integer;
  var k: integer;
  begin
    if Velikost[k1] < Velikost[k2] then k := k2 else k := k1;
    NadKletka[k1] := k; NadKletka[k2] := k;
    Velikost[k] := Velikost[k1] + Velikost[k2];
    ZlijKletki := k;
  end; {ZlijKletki}

var T: text; Celica, y, x, k: integer;
begin
```

```
{ Preberimo velikost mreže. }
Assign(T, 'kletke.in');
Reset(T); ReadLn(T, M, N);
{ Nikoli ne bo hkrati „odprtih“ več kot M kletk. Zato bomo
  za kletke vedno uporabljali števila od 1 do M. Tista, ki
  trenutno niso uporabljena za nobeno kletko, hranimo v seznamu. }
StProstih := M; for k := 1 to M do Proste[k] := k;

{ Preglejmo celo mrežo. }
Najvecja := 0; Najmanjsa := M * N; StKletk := 0;
for y := 1 to N do begin
  for x := 1 to M do begin
    Read(T, Celica); Celica := Celica and (Zgoraj or Levo);

    if Celica = 0 then begin
      { Celica ima zidova zgoraj in levo; ustanovimo zanjo novo kletko. }
      if y > 1 then ZmanjsajKletko(Kletka[x]);
      k := Proste[StProstih]; StProstih := StProstih - 1;
      Velikost[k] := 1; VTejVrsti[k] := 1; NadKletka[k] := k; Kletka[x] := k;

    end else if Celica = Zgoraj then begin
      { Celica pripada isti kletki kot tista nad njo. NajdiKletko(x) nam bo že
        tudi postavila Kletka[x] na k in popravila VTejVrsti[k]. }
      k := NajdiKletko(x);
      Velikost[k] := Velikost[k] + 1;

    end else if Celica = Levo then begin
      { Celica pripada isti kletki kot tista levo od nje. }
      if y > 1 then ZmanjsajKletko(Kletka[x]);
      { Spomnimo se, da Kletka[x - 1] zdaj že kaže na pravo kletko, ne na
        kakšno podkletko, saj smo imeli ravno v prejšnji iteraciji opraviti
        s tisto celico in smo zato gotovo poskrbeli, da je takrat kazala
        na pravo kletko; medtem pa se ta kletka tudi ni imela časa s čim zliti.
        Zato ni treba klicati NajdiKletko(x - 1). }
      k := Kletka[x - 1]; Kletka[x] := k;
      VTejVrsti[k] := VTejVrsti[k] + 1; Velikost[k] := Velikost[k] + 1;

    end else begin
      { Zlijmo kletko nad njo in tisto levo od nje
        (če nista to že zdaj ena in ista kletka). }
      k := NajdiKletko(x); { že tudi popravi Kletka[x] }
      if Kletka[x - 1] <> k then begin { sta različni → treba bo zlivati }
        k := ZlijKletki(Kletka[x - 1], k);
        if Kletka[x] <> k then begin
          { Element Kletka[x] se bo spremenil na k, zato ima dosedanja
            kletka Kletka[x] eno referenco manj, k pa eno več. }
          ZmanjsajKletko(Kletka[x]);
          Kletka[x] := k; VTejVrsti[k] := VTejVrsti[k] + 1;
        end; {if}
```

```
            end; {if}
            Velikost[k] := Velikost[k] + 1;
          end; {if}
      end; {for x}
      ReadLn(T);
    end; {for y}
    Close(T);
    { Zaključimo celice zadnje vrstice. }
    for x := 1 to M do ZmanjsajKletko(Kletka[x]);
    { Izpišimo rezultate. }
    Assign(T, 'kletke.out'); Rewrite(T);
    WriteLn(T, StKletk); WriteLn(T, Najmanjsa); WriteLn(T, Najvecja);
    Close(T);
end. {Kletke2}
```

Na testnih primerih z našega tekmovanja se ta in prejšnji program izvajata praktično enako hitro, saj so mreže velikosti do 100×100 vendarle zelo majhne in se porabi za branje vhodne datoteke več časa kot pa za samo pregledovanje mreže.

R2001.3.5 Prefiksi

N: 440 Mislimo si, da med znake niza S postavljamo oznake, s katerimi si zaznamujemo, do kod smo se po tem nizu uspeli prebiti s stikanjem vzorcev $S_1, \ldots, S_N$. Na začetku si postavimo oznako pred prvi znak niza S. Potem se vsakič pomaknemo do naslednje (najbolj leve še neobdelane) oznake in za vsak vzorec pogledamo, če se nadaljevanje niza S od te oznake naprej začne s tem vzorcem; če je tako, lahko na konec te pojavitve tega vzorca postavimo novo oznako. S tem smo mogoče postavili eno ali več novih oznak in tako nadaljujemo, dokler ne obdelamo vseh oznak (ali pa pridemo do konca niza S). Na koncu vrnemo položaj zadnje (skrajno desne) oznake.

Malo lahko izboljšamo še porabo pomnilnika, če opazimo, da oznak, mimo katerih smo že šli, ne potrebujemo več, poleg tega pa doslej postavljene oznake prav gotovo ne ležijo dlje kot za eno dolžino najdaljšega vzorca naprej od trenutnega mesta v nizu. Torej ni treba imeti v pomnilniku celega niza S hkrati; zadosti je že toliko znakov, kolikor je dolg najdaljši vzorec — največ sto znakov. Enako velja tudi za tabelo oznak (Oznake v spodnjem programu). Ti dve tabeli, dolgi po sto znakov, potem uporabljamo kot krožni pomnilnik (če potrebujemo znak na indeksu 12345, ga najdemo na indeksu 45). Postopek deluje nekako tako, kot da bi se po našem dolgem nizu S pomikali z oknom, ki je dolgo le toliko kot najdaljši možni vzorec.

```
program Prefiksi;
const MaxN = 100; MaxP = 100;
```

```
var Vzorci: array [1..MaxN] of string[MaxP];
    Oznake: array [0..MaxP − 1] of boolean;
    S: array [0..MaxP − 1] of char;
    T: text; iBranje, iOznaka, iZadnja: longint; i, j, L, N, StOznak: integer;
begin
  { Preberimo vzorce. }
  Assign(T, 'prefiksi.in');
  Reset(T); ReadLn(T, N);
  for j := 1 to N do ReadLn(T, Vzorci[j]);
  for i := 0 to MaxP − 1 do Oznake[i] := false;
  Oznake[0] := true; StOznak := 1;

  { Berimo niz in označujmo dosegljiva mesta. }
  iBranje := 0; iOznaka := 0; iZadnja := 0;
  while (iOznaka <= iBranje) or (not Eoln(T)) do begin
    if not Eoln(T) then { preberimo naslednji znak }
      begin Read(T, S[iBranje mod MaxP]); iBranje := iBranje + 1 end;
    if Eoln(T) or (iBranje >= MaxP) then begin
      if Oznake[iOznaka mod MaxP] then begin

        { Poglejmo, če se da od tega označenega mesta kako nadaljevati. }
        iZadnja := iOznaka; Oznake[iOznaka mod MaxP] := false;
        StOznak := StOznak − 1;
        for j := 1 to N do begin { primerjajmo S[iOznaka..] z vzorcem j }
          i := 1; L := Length(Vzorci[j]);
          while (i <= L) and (iOznaka + i − 1 < iBranje) do
            if S[(iOznaka + i − 1) mod MaxP] = Vzorci[j, i]
            then i := i + 1 else break;
          if i > L then if not Oznake[(iOznaka + L) mod MaxP] then begin
            { označimo doseženo mesto }
            Oznake[(iOznaka + L) mod MaxP] := true;
            StOznak := StOznak + 1;
          end; {if}
        end; {for j}
        if StOznak = 0 then break;

      end; {if}
      iOznaka := iOznaka + 1;
    end; {if}
  end; {while}

  { Izpišimo rezultate. }
  Assign(T, 'prefiksi.out'); Rewrite(T); WriteLn(T, iZadnja); Close(T);
end. {Prefiksi}
```

R2001.3.6 Podobnost med dokumenti

N: 441

Pri tej nalogi je treba le paziti na „knjigovodstvo“. Ker je vseh besed in besedil malo, hrani spodnji program besede v vseh besedilih kar kot nize (ne pripiše jim npr. kakšnih številskih oznak, kar bi bilo sicer pri kakšni večji zbirki besedil koristno narediti in si pri tem pomagati z razpršeno tabelo). Ko dodaja novo besedo v besedilo, mora preveriti, če je nismo v njem mogoče že zasledili — v tem primeru je treba le povečati njen števec pojavitev. Pri izračunu števca v formuli za podobnost (skalarni produkt vektorjev $\mathbf{x}$ in $\mathbf{y}$) je treba pravzaprav sešteti produkte števil pojavitev posamezne besede v obeh besedilih. Pri tako kratkih besedilih, kot jih obravnava ta naloga, bi lahko vzeli kar dve gnezdeni zanki, ki gresta po vseh besedah prvega in vseh besedah drugega besedila ter v primeru, da sta besedi enaki, zmnožita njuna števca. Lepše pa je, če seznama besed najprej uredimo; potem se sprehajamo z enim indeksom po prvem in z enim po drugem seznamu; če sta trenutni besedi (npr. w_1 in w_2) enaki, zmnožimo njuna števca (in povečamo oba indeksa), sicer pa premaknemo ali prvi indeks ali pa drugega: če je $w_1 < w_2$, besede w_1 prav gotovo ne bomo našli v drugem besedilu, pač pa utegnemo w_2 še najti v drugem; zato povečamo prvi indeks (premaknemo se naprej po seznamu besed prvega besedila). Če pa je $w_1 > w_2$, je stvar ravno obrnjena in moramo povečati drugi indeks.

```
program PodobnostMedBesedili;
const MaxD = 20; MaxBesed = 20; MaxDolz = 20;
type BesedaT = string[MaxDolz];
var StBesed: array [1..MaxD] of integer;
    Besede: array [1..MaxD, 1..MaxBesed] of BesedaT;
    StPojavitev: array [1..MaxD, 1..MaxBesed] of integer;
    Norma: array [1..MaxD] of integer;
    T: text; S: string; B: BesedaT;
    i, i2, j1, j2, k, kk, L, D: integer; { D = število besedil }
    Naj1, Naj2: integer; { najpodobnejši besedili doslej }
    Pod, NajPod: real; { trenutna in največja podobnost }
begin
  { Preberimo vhodno datoteko. }
  Assign(T, 'docclust.in');
  Reset(T); ReadLn(T, D); Naj1 := 0; Naj2 := 0;

  for i := 1 to D do begin
    ReadLn(T, S); L := Length(S); j1 := 1;
    StBesed[i] := 0;

    { Razrežimo niz S na besede. }
    while j1 <= L do begin
      while j1 <= L do { preskočimo presledke }
        if S[j1] = ' ' then j1 := j1 + 1 else break;
      if j1 > L then break;
```

```
        j2 := j1;
        while j2 <= L do { preberimo besedo }
          if S[j2] = ' ' then break else j2 := j2 + 1;
        { Zdaj bomo besedo vstavili v zaporedje Besedila[i]. Če je že v njem, bomo
          samo povečali število pojavitev. Seznam besed naj bo urejen po abecedi. }
        k := 1; B := Copy(S, j1, j2 - j1); j1 := j2;
        while k <= StBesed[i] do
          if B <= Besede[i, k] then break else k := k + 1;
        if (k <= StBesed[i]) and (B = Besede[i, k]) then
          StPojavitev[i, k] := StPojavitev[i, k] + 1
        else begin
          kk := StBesed[i]; while kk >= k do begin
            StPojavitev[i, kk + 1] := StPojavitev[i, kk];
            Besede[i, kk + 1] := Besede[i, kk]; kk := kk - 1;
          end; {while}
          Besede[i, k] := B; StPojavitev[i, k] := 1;
          StBesed[i] := StBesed[i] + 1;
        end; {if}
      end; {while}

      Norma[i] := 0;
      for k := 1 to StBesed[i] do
        Norma[i] := Norma[i] + StPojavitev[i, k] * StPojavitev[i, k];

      { Primerjajmo to besedilo z vsemi prejšnjimi. }
      for i2 := 1 to i - 1 do begin
        Pod := 0; j1 := 1; j2 := 1;
        while (j1 <= StBesed[i]) and (j2 <= StBesed[i2]) do
          if Besede[i, j1] = Besede[i2, j2] then begin
            Pod := Pod + StPojavitev[i, j1] * StPojavitev[i2, j2];
            j1 := j1 + 1; j2 := j2 + 1;
          end else if Besede[i, j1] < Besede[i2, j2] then j1 := j1 + 1
          else j2 := j2 + 1;
        Pod := Pod / Sqrt(Norma[i] * Norma[i2]);
        { Ali je to nov najpodobnejši par besedil? }
        if (Naj1 <= 0) or (Pod > NajPod) then
          begin Naj1 := i2; Naj2 := i; NajPod := Pod end;
      end; {for i2}
    end; {for i}
    Close(T);

    { Izpišimo rezultate. }
    Assign(T, 'docclust.out'); Rewrite(T); WriteLn(T, Naj1, ' ', Naj2); Close(T);
end. {PodobnostMedBesedili}
```

Viri nalog za leto 2001: stopniščni avtomat, CD-predalček — Mark Martinec; besedilo v stolpcu, 3-D križci in krožci — Mitja Lasič; iskalnik — Uroš Jovanovič; kletke — Blaž Novak; podobnost med dokumenti — Marko Grobelnik; tipkanje, pitagorejske trojice, oklepajski izrazi, parlament — Janez Brank; psevdo-tetris — po zgledu CERC 1999; števila v ogledalu — CERC 1998, tudi #713 na `online-judge.uva.es`; prefiksi — IOI 1996. Hvala Marjanu Šterku in Blažu Novaku za implementacijo rešitev nekaterih nalog iz tretje skupine.

Primer pri nalogi *Besedilo v stolpcu* je iz 21. poglavja romana *Drakula* Brama Stokerja. Verzi v primeru pri nalogi *Podobnost med dokumenti* tvorijo prvi dve kitici neke pesmi, ki jo pripisujejo siru Thomasu Wyattu (1503–42); besedilo je iz izdaje A. K. Foxwellove (London, 1913).

26. državno tekmovanje v znanju računalništva (2002)

NALOGE ZA PRVO SKUPINO

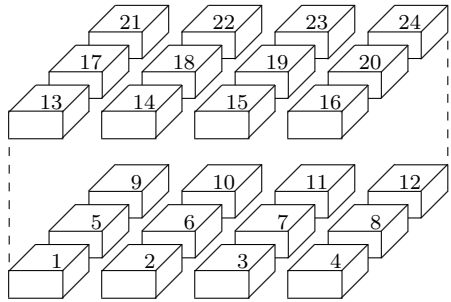
2002.1.1 Pristanišče

R: 500

Naročniki večjih količin tovora se običajno odločajo za prevoz tovora z ladjo. Ko zabojniki prispejo v luko, jih morajo tam uskladiščiti, vse dokler ponje ne pride naročnik ali pa jih morajo natovoriti na vlak. V neki luki zabojnike nalagajo enega ob drugega in ko zapolnijo celo vrsto, se lotijo nalaganja v drugo vrsto, nato tretjo in tako dalje. Ko se zapolni cela površina, začnejo zabojnike nalagati še na naslednjo višino v enakem zaporedju kot prej (torej prvi zabojnik na drugi višini pride položen na prvi zabojnik v prvi višini itd.).

Podano imamo število zabojnikov v vsaki vrsti (recimo n), število vrst v vsaki plasti (recimo m) in število plasti (recimo l).

Na spodnji sliki je za primer prikazana postavitev 24 zabojnikov za $n = 4$, $m = 3$, $l = 2$.



Napiši program ali podprogram, ki bo za dane n, m in l ter za neko številko zabojnika izračunal, v kateri plasti in v kateri vrsti je ta zabojnik ter kateri v svoji vrsti je. Vse troje se šteje od 1 naprej in to v takem vrstnem redu, da manjše številke predstavljajo tiste dele skladišča, ki so bili zapolnjeni prej.

Primer: zabojnik s številko 16 na gornji sliki je četrti zabojnik v prvi vrsti druge plasti.

2002.1.2 Najdaljši cikel v permutaciji

R: 500 Liliput je majhno mesto z veliko avtobusnimi postajami. Vse proge mestnih avtobusov so krožne. Vsaka postaja pa pripada natančno eni progi, torej nobena dva avtobusa nikoli ne obiščeta iste avtobusne postaje.

Težava je v tem, da Liliputanci objavljajo vozne rede svojih avtobusov na prav poseben način. Avtobusne postaje so oštevilčene s števili od 1 do N, opis vseh prog v njihovem mestu pa predstavijo kot zaporedje N števil.

Za primer vzemimo zaporedje števil (8, 10, 6, 3, 9, 4, 2, 5, 1, 7), ki ga lahko predstavimo kot:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 8 & 10 & 6 & 3 & 9 & 4 & 2 & 5 & 1 & 7 \end{pmatrix}$$

Prvo število v zaporedju je 8, kar pomeni, da gre avtobus, ki odpelje s postaje 1, na postajo 8. Osmo število v zaporedju je 5, kar pomeni, da avtobus, ki pripelje na postajo številka 8, nadaljuje svojo pot do postaje številka 5; peto število je 9, zato avtobus nadaljuje pot do postaje 9 in tako naprej, dokler se ne vrne na začetno postajo.

Iz gornjega zaporedja lahko po tem postopku izluščimo tri krožne avtobusne proge:

$$(1, 8, 5, 9),\ (2, 10, 7),\ (3, 6, 4).$$

Tvoja naloga je **napisati program**, ki bo nevajenemu tujcu za poljubno zaporedje števil izpisal vse proge mestnih avtobusov ter mu povedal, koliko postaj ima najdaljša proga.

Na voljo imaš polje velikosti N, ki vsebuje poljubno zaporedje števil, ki ustreza opisu prog mestnih avtobusov:

```
const N = ...;
var Postaje: array [1..N] of integer;
```

2002.1.3 Razbijanje kode

R: 501 Pri uporabi bančne kartice moramo dokazati, da poznamo kodo PIN (osebno identifikacijsko številko). Podobno v deželi PinLand uporabljajo štirimestne *črkovne* kode PIN tudi za dostop do pomembnih podatkov na mreži. Pri razbijanju kode PIN si pomagajmo s poskušanjem: vemo, kakšne so dovoljene kode, in preizkusimo vse možnosti.

Za testiranje imamo na voljo funkcijo

```
function Test(Geslo: string): boolean; external;
```

oziroma

```
extern bool Test(const char* Geslo);
```

Ta funkcija vrne `true`, če je koda pravilna.

Za kodo PIN veljata naslednji omejitvi:

- dolga je točno štiri znake;
- dovoljeni so le znaki `A`, ..., `Z`, torej ravno vse velike črke angleške abecede.

Napiši program, ki bo s poskušanjem odkril pravo kodo PIN in jo izpisal.

2002.1.4 Bencin

R: 501

Janez P. se odpravlja na pot z avtomobilom iz Ljubljane v Bruselj. Na poti bo moral večkrat doliti gorivo. Ker je cena goriva na bencinskih črpalkah različna, ga zanima, kako naj v Bruselj pride kar najceneje. **Opiši postopek**, ki mu bo pomagal. Na voljo imaš naslednje podatke: dolžina poti (recimo 1200 km); velikost posode za gorivo (recimo 60 l); poraba goriva (recimo 7 litrov na 100 km);[93] število črpalk ob poti; za vsako črpalko pa še njen položaj na poti (oddaljenost od Ljubljane v kilometrih) in ceno bencina na njej. (Za cene bencina si misli, da so vse izražene v istih denarnih enotah — s pretvarjanjem med valutami se ti ni treba ubadati.)

Janez P. pot začne s prazno posodo, prva črpalka pa je že takoj na začetku poti. **Opiši postopek**, ki lahko za poljubno razporeditev črpalk in cene bencina ugotovi, kje in koliko goriva je treba doliti, da ga ne bo nikjer zmanjkalo, obenem pa bomo za gorivo porabili čim manj denarja.

NALOGE ZA DRUGO SKUPINO

2002.2.1 Kodiranje

R: 505

Recimo, da se lahko v besedilih, s katerimi imamo opravka, pojavijo le črke od a do h. V tipičnem besedilu se ne pojavljajo vse črke enako pogosto, pač pa so pogostosti pojavljanja posameznih črk takšne:

Črka	a	b	c	d	e	f	g	h
Pogostost	12,5 %	6,25 %	25 %	6,25 %	25 %	6,25 %	12,5 %	6,25 %

Ta tabela pove, da se v tisoč črkah značilnega besedila pojavi črka a 125-krat, črka c 250-krat itd. Primer tovrstnega (a malo krajšega) besedila bi bil npr.:

ccfdcceabbeeegceabccefhaageaaahgbegceefbcecacfaddeegeeeecefghgcecdcccgeghahccdcg

[93] Pri teh številkah je sicer videti, kot da ni treba dotočiti bencina več kot enkrat, vendar utegne biti ceneje, če dotakamo večkrat po malem. Tako ali tako so te konkretne številke mišljene le za oporo pri razmišljanju, sicer pa nas zanima postopek, ki bi deloval za poljubne vrednosti teh podatkov.

Opiši, kako bi z ničlami in enicami predstavil (kodiral) posamezno od teh črk, da bi bilo tipično besedilo zapisano čim krajše (vendar pa bi se ga še vedno dalo dekodirati nazaj v prvotno obliko)?

Primer takšnega kodiranja za neko drugo vrsto besedil: recimo, da bi bila naša besedila lahko sestavljena le iz štirih črk, p, q, r in s, pri čemer bi se p pojavil v 90 % primerov, q v 8 %, r in s pa v 1 % primerov. Potem bi bilo smiselno p predstaviti s kodo 0, q s kodo 10, r s kodo 110 in s s kodo 111.

2002.2.2 Prijatelji in sovražniki

R: 505 Jurij W. Grm mlajši je sila pomemben in vpliven mož, ki pozna veliko ljudi. Pozna jih celo toliko, da sploh ne ve več, kateri so njegovi pravi prijatelji in kateri pravi sovražniki. Jurij je najel skupino detektivov, ki je za vsakega od ljudi, ki jih kakorkoli pozna (tudi posredno), naredila seznam *neposrednih prijateljev* in *neposrednih sovražnikov*.

Jurija zanima, kdo so njegovi pravi prijatelji in kdo pravi sovražniki. Pri tem veljajo naslednje definicije:

Jurijevi *prijatelji* so tisti, ki so bodisi njegovi neposredni prijatelji, neposredni prijatelji njegovih prijateljev ali pa neposredni sovražniki njegovih sovražnikov.

Njegovi *sovražniki* pa so tisti, ki so njegovi neposredni sovražniki, neposredni prijatelji kakšnega sovražnika ali neposredni sovražniki Jurijevih prijateljev.

Jurijev *pravi prijatelj* je vsakdo, ki je njegov prijatelj, ne pa tudi sovražnik. *Pravi sovražnik* pa je vsakdo, ki je njegov sovražnik, ne pa tudi prijatelj.

Opomba: te definicije dopuščajo tudi, da je Jurij sam svoj sovražnik. V tem primeru so vsi, ki jih posredno ali neposredno pozna, hkrati njegovi prijatelji in sovražniki. Tudi to se pač lahko zgodi.

Na voljo imaš tabelo s podatki o neposrednih prijateljih in neposrednih sovražnikih vsakega človeka:

```
const N = ...;
type KdoJeKdo = (Neznanec, Prijatelj, Sovrag);
var Odnos: array [0..N − 1, 0..N − 1] of KdoJeKdo;
```

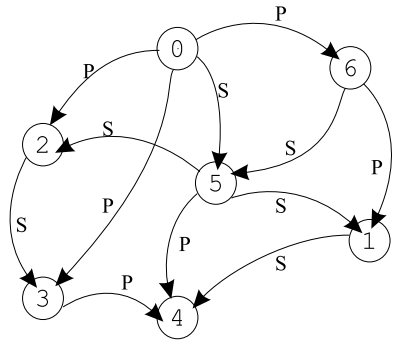
```
#define N ...
typedef enum { Neznanec, Prijatelj, Sovrag } KdoJeKdo;
KdoJeKdo Odnos[N][N];
```

Polje Oseba je velikosti $N \times N$ in vsebuje podatke o neposrednih prijateljstvih in neposrednih sovraštvih. Vrednost Oseba[i, j] pove, kaj je oseba j osebi i. Poznanstva so enostranska, torej sploh ni nujno, da velja Oseba[i, j] = Oseba[j, i].

Tako lahko Jurij misli, da je Vlado njegov (neposredni) prijatelj, Vlado pa ima Jurija za svojega (neposrednega) sovražnika. Jurij ima v tej tabeli številko 0.

Opiši postopek, s katerim bi Jurij ugotovil, kdo so njegovi pravi prijatelji in kdo pravi sovražniki. Lahko si pomagaš z gornjo deklaracijo tabele Odnos, ni pa treba pisati izvorne kode v kakšnem konkretnem programskem jeziku.

Primer (povezava P oz. S od a do b pomeni, da je b a-jev neposredni prijatelj oz. neposredni sovražnik):



V tem primeru so 2, 6 in 1 Jurijevi pravi prijatelji, 5 pravi sovražnik, osebi 3 in 4 pa mu nista niti prava prijatelja niti prava sovražnika.

Opomba: na tej sliki lahko od Jurija do vseh oseb pridemo že prek samo dveh poznanstev, kar pa v splošnem ne velja — do nekaterih oseb včasih potrebujemo več kot dva koraka (ali pa do njih sploh ne moremo priti).

2002.2.3 Razbijanje gesel

Radi bi vdrli v nek sistem, zaščiten z geslom. Pri razbijanju gesel si pomagamo s poskušanjem. Vemo, kakšna so dovoljena gesla, in preizkusimo vse možnosti. R: 506

Za testiranje imamo na voljo funkcijo

```
function Test(Geslo: string): boolean; external;
```

oziroma

```
extern bool Test(const char* Geslo);
```

Ta funkcija vrne true, če je koda pravilna, drugače pa false.

Za geslo veljajo naslednje omejitve:

- dolgo je najmanj tri znake in največ osem znakov;
- dovoljeni znaki so `A`, ..., `Z`, `a`, ..., `z` in `0`, ..., `9` (velike in male črke angleške abecede ter številke od 0 do 9);

- vsako geslo mora vsebovati vsaj eno črko in vsaj eno števko.

Naloga:

- Napiši program, ki bo s poskušanjem poiskal pravo geslo in ga izpisal.
- Oceni, kakšno je navečje, najmanjše in pričakovano število poskusov za naključno geslo.
- Ali je tako razbijanje gesel smiselno (časovno sprejemljivo)? Odgovor utemelji.

2002.2.4 TiVo

R: 508

Pri gledanju televizijskega programa bi si včasih želeli, da bi lahko začasno ustavili program med kakšnim opravkom, potem pa nadaljevali z gledanjem, ne da bi kaj zamudili — pa tudi kakšno posebno zanimivo sceno bi radi v miru ponovno pogledali. Nastalo zamudo bi lahko nadoknadili ob prvi primerni priložnosti, na primer s hitrim preskakovanjem reklam.

Za silo si lahko pomagamo z videorekorderjem, vendar ta metoda ne deluje dobro za spremljanje živega programa. Ko bi le lahko imeli isto kaseto speljano skozi dva snemalnika...

Na tržišču so se že pojavili snemalniki (pravzaprav računalniki), ki obvladajo tudi tovrstne zahteve gledalcev. Namesto da bi snemali na trak, zapisujejo sliko na računalniški disk; zmorejo pa tudi snemati program iz sprejemnika in hkrati predvajati katerikoli del že posnetega programa, tako najbolj svežega skoraj brez zamika, kot tudi starejšega z zamikom. Ker je velikost diska omejena na nekaj ur, se najstarejši deli posnetka samodejno brišejo, ko začne zmanjkovati prostora za nov posnetek.

Denimo, da uspemo zapisati eno televizijsko sličico na en diskovni blok. Sličice si pri snemanju in pri normalnem predvajanju sledijo s hitrostjo 25 na sekundo. Diskovni bloki (sličice) so oštevilčeni od 0 do 300 000, kar zadošča za dobre tri ure posnetka. Ob zagonu sistema so vsi bloki prazni, snemanje in predvajanje se začneta takoj (istočasno) in potem potekata neprekinjeno.

Določi in opiši podatkovno strukturo, ki ti bo pomagala pri učinkovitem določanju oz. iskanju zaporednih sličic tako, da bo omogočeno stalno snemanje (shranjevanje novih sličic, prihajajočih iz sprejemnika), brisanje oziroma pozabljanje najstarejših sličic, ter hkrati s snemanjem tudi predvajanje poljubnega dela posnetega programa v načinih: normalna hitrost predvajanja, zamrznjena slika in pospešeno predvajanje z 10-kratno hitrostjo naprej ali nazaj.

Napiši tudi naslednje **tri podprograme**, potrebne za upravljanje s takšnim snemalnikom:

- za snemalni del:

  - podprogram KamShranitiNovo, ki ga bo sprejemniški del klical 25-krat v sekundi (za vsako novo sprejeto sličico), podprogram pa bo vsakokrat vrnil številko diskovnega bloka, na katerega naj sprejemniški del shrani pravkar sprejeto sličico;

- za predvajalni del:

  - podprogram KateroSlikoPrikazati, ki ga bo predvajalni del elektronike klical 25-krat v sekundi; podprogram mora vsakokrat vrniti številko diskovnega bloka, na katerem je zapisana sličica, ki mora biti prikazana na televizijskem ekranu za naslednjo petindvajsetinko sekunde. Na zaporedje sličic naj vpliva zadnja nastavitev, določena s klicem tvojega podprograma IzberiNacinPredvajanja;
  - podprogram IzberiNacinPredvajanja, ki ga aktivira gledalec s pritiskom na tipko. Trenutna nastavitev naj vpliva na zaporedje sličic, kot jih mora vračati tvoj podprogram KateroSlikoPrikazati. Kot argument dobi celo število, ki pomeni:

    0 pavza — mirujoča slika, vsakokrat vidimo isto sličico;
    1 normalno predvajanje naprej — sličice naj si vrstijo v istem zaporedju, kot so bile posnete;
    10 hitro predvajanje naprej z desetkratno hitrostjo — prikaže naj se le vsaka deseta sličica, vmesne izpustimo;
    −10 hitro predvajanje nazaj z desetkratno hitrostjo, torej proti vedno starejšim sličicam (dokler ne naletimo na najstarejše).

Predpostaviš lahko, da se ne bo hkrati izvajalo po več tvojih podprogramov; torej, medtem ko se eden izvaja, ga drugi ne more prekiniti, pač pa se lahko začne izvajati šele, ko se prvi konča.

Obnašanje ob robnih pogojih (na primer ko pri hitrem predvajanju naletimo na najstarejše ali na najnovejše posnetke) ni predpisano, vendar se spodobi, da se program odloči za varianto, ki bo gledalca čim manj presenetila. **Napiši**, kako se tvoja rešitev obnaša v takšnih primerih.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

[Na začetku tekmovanja smo tekmovalcem najprej razdelili naslednja navodila. Nekaj minut kasneje so dobili tudi besedilo nalog, za reševanje pa so imeli slabe tri ure časa. — *Op. ur.*]

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*`.in` in izpisujejo svoje rezultate v *imenaloge*`.out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno

po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil mapo s svojim uporabniškim imenom ter navodili, ki jih pravkar prebiraš. Ko boš sedel pred računalnik, boš dobil nadaljnja navodila za prijavo v sistem.

Na vsakem računalniku imaš na voljo enoto (disk) `U:`, na kateri lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku Pascal, C ali C++, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal in GNU C++. Za delo lahko uporabiš `turbo` (Turbo Pascal), `fp` (FreePascal), `tc` (Turbo C), ali `gcc`/`g++` (GNU C/C++ — command line compiler). Ves potreben softver lahko najdeš na `c:\Programi` ter v meniju `Start` pod `Programs` in `Prevajalniki`.

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `rtk.exe`, ki ga lahko uporabiš za preverjanje svojih rešitev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

`rtk` *imenaloge*`.pas`
`rtk` *imenaloge*`.c`
`rtk` *imenaloge*`.cpp`

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Na spletni strani boš dobil obvestilo o tem, ali je program na testne primere odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z drugimi datotekami kot z vhodno in izhodno. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov, prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Če si oddal N programov za to nalogo in je najboljši med njimi pravilno rešil M (od desetih) testnih primerov, dobiš pri tej nalogi $\max\{0, 10M - 3(N-1)\}$ točk. Z drugimi besedami: vsak pravilno rešen testni primer ti prinese 10 točk, za vsako oddajo (razen prve) pri tej nalogi pa se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk.

Skupno število točk tekmovalca je vsota po vseh nalogah. Tekmovalce razvrstimo po skupnem številu točk.

Vsak tekmovalec se mora sam zase odločiti o tem, katerim nalogam bo posvetil svoj čas, v kakšnem vrstnem redu jih bo reševal in podobno. Verjetno je priporočljivo najprej reševati lažje naloge.

Poskusna naloga (ne šteje k tekmovanju) `poskus.in`, `poskus.out`

Napiši program, ki iz vhodne datoteke prebere eno število (le-to je v prvi vrstici, okoli njega ni nobenih dodatnih presledkov ipd.) in izpiše njegov desetkratnik v izhodno datoteko.

Primer vhodne datoteke:	Ustrezna izhodna datoteka:
`123`	`1230`

Primer rešitve:

```
program PoskusnaNaloga;
var T: text; i: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * i); Close(T);
end. {PoskusnaNaloga}
```

```
#include <stdio.h>
int main() {
  FILE *f = fopen("poskus.in", "rt");
  int i; fscanf(f, "%d", &i); fclose(f);
  f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * i);
  fclose(f); return 0;
}
```

```
#include <fstream.h>
int main() {
  ifstream ifs("poskus.in"); int i; ifs >> i;
  ofstream ofs("poskus.out"); ofs << 10 * i;
  return 0;
}
```

NALOGE ZA TRETJO SKUPINO

2002.3.1 Limuzine `limo.in`, `limo.out`

Podjetje LepeLimuzine ima vozni park limuzin, ki so na voljo poslovnežem, ki se jim mudi na sestanke. Zaradi narave posla šoferji veliko časa presedijo na parkiriščih; zaradi visokih stroškov parkiranja so se v podjetju odločili, da bodo v vsako limuzino vgradili računalniški sistem, ki bi jim znal svetovati, kam naj gredo parkirat.

R: 509

Napiši program, ki bo prebral ceno ure vožnje, trajanje sestanka in seznam parkirišč z njihovimi cenami parkiranja in oddaljenostmi od kraja sestanka. Program naj ugotovi, kje je treba parkirati (in za koliko časa), da bodo skupni stroški (vožnja + parkirnina) najmanjši in da bo limuzina ravno ob koncu sestanka spet prišla nazaj na prvotno mesto.

Vhodna datoteka vsebuje v prvi vrstici ceno ure vožnje (v SIT) in trajanje sestanka (v minutah). Nato sledi za vsako parkirišče po ena vrstica, v kateri sta dve števili; prvo pove ceno ure parkiranja na tem parkirišču (v SIT), drugo pa pove, koliko minut potrebujemo, da se pripeljemo od kraja, kjer smo odložili potnika, do tega parkirišča (ali pa nazaj — predpostavimo, da traja vožnja v nasprotno smer enako dolgo). Na koncu je vrstica, ki vsebuje dve ničli. Parkirišč ni več kot 1000; za ceno vožnje in vse cene parkiranja velja, da so vsaj 1 in največ 10000 SIT/h; čas vožnje do posameznega parkirišča je vsaj 1 minuto in največ 1000 h, ista omejitev pa velja tudi za trajanje sestanka. Vse cene in trajanja so cela števila.

V *izhodno datoteko* izpiši številko parkirišča, na katerem moramo parkirati, in trajanje našega parkiranja na njem (v minutah). Številka 1 pomeni prvo parkirišče, 2 drugo in tako naprej. Če so parkirišča predraga ali predaleč in je zato bolje kar voziti po mestu, ne da bi kje parkirali, izpiši dve ničli. Če bi se dalo do najmanjših stroškov priti na različne načine (npr. z različnimi parkirišči), je vseeno, katerega opišeš. Če parkiraš na nekem parkirišču, moraš tam ostati vsaj eno minuto.

Primer vhodne datoteke:

```
500 60
200 10
700 20
0 0
```

Ustrezna izhodna datoteka:

```
1 40
```

2002.3.2 Uvrstitve tekmovalcev

`tekma.in`, `tekma.out`

R: 510

Skupina kolesarjev hkrati začne krožno dirko. Vsakič, ko kateri izmed njih prevozi štartno-ciljno črto, štarter vtipka njegovo štartno številko, računalnik pa jo zapiše v datoteko. Dirka se konča, ko prvi tekmovalec prevozi predpisano število krogov — preostali le še končajo trenutni krog (tudi to se vpiše v datoteko).

Napiši program, ki prebere datoteko in izpiše uvrstitve tekmovalcev. (Kriterij za razvrstitev je ta, da so višje uvrščeni tisti tekmovalci, ki so prevozili več krogov, med takimi, ki so prevozili enako število krogov, pa tisti, ki so jih prevozili prej.) Če neki tekmovalec ne konča nobenega kroga, pomeni, da ni uvrščen in ga tudi v rezultatih ne sme biti. Upoštevaj, da lahko tekmovalci prehitevajo eden drugega tudi za več krogov.

Vhodna datoteka vsebuje v prvi vrstici število tekmovalcev (recimo N, ki je celo število, $1 \leq N \leq 2000$). V naslednjih vrsticah (ki jih ni več kot milijon) so štartne številke tekmovalcev v takšnem vrstnem redu, v kakršnem so vozili čez štartno-ciljno črto; v vsaki vrstici je po ena, na koncu pa pride vrstica, ki vsebuje število 0. (Štartne številke tekmovalcev so cela števila od 1 do N.)

V *izhodno datoteko* izpiši štartne številke tekmovalcev po uvrstitvah, vsako v svojo vrstico: v prvo vrstico zmagovalca, v drugo drugega in tako naprej.

Primer vhodne datoteke:

```
4
1
2
3
4
2
4
3
4
2
1
4
1
2
3
0
```

Pripadajoča izhodna datoteka:

```
4
2
1
3
```

2002.3.3 Število vsot

`vsote.in`, `vsote.out`

Zanima nas, na koliko načinov lahko naravno število N (naravna števila so tista, ki so cela in večja od nič) zapišemo kot vsoto K naravnih števil (K je celo število, za katero velja $1 \leq K \leq N$). Pri tem nam vrstni red seštevancev v vsoti ni pomemben (če lahko eno izražavo dobimo iz druge tako, da le premešamo seštevance, ju obravnavamo kot eno in isto in ne štejemo vsake posebej). Tudi zapis $N = N$ velja kot izražava N-ja kot vsote enega samega seštevanca (za $K = 1$ bi torej dobili odgovor 1, ker lahko N samo na ta edini način izrazimo kot vsoto enega pozitivnega celega števila).

R: 511

Napiši program, ki prebere N in K ter izračuna, koliko je teh načinov:

- *Vhodna datoteka* vsebuje v prvi vrstici dve celi števili, najprej N in nato K, ločeni z enim presledkom. Veljalo bo $1 \leq K \leq N \leq 100$.
- V *izhodno datoteko* izpiši število, ki pove, na koliko načinov lahko N zapišemo kot vsoto K seštevancev.

Primer vhodne datoteke:

```
10 3
```

Pripadajoča izhodna datoteka:

```
8
```

Število 10 lahko namreč kot vsoto treh pozitivnih celih števil zapišemo na osem načinov:

$$\begin{aligned} 10 &= 8+1+1 = 7+2+1 = 6+3+1 = 6+2+2 \\ &= 5+4+1 = 5+3+2 = 4+4+2 = 4+3+3. \end{aligned}$$

2002.3.4 Sestanki

`sestanki.in`, `sestanki.out`

R: 513 Jurij W. Grm mlajši bi rad sestankoval. Ker pa so vsi, ki naj bi bili prisotni na sestanku, zelo zaposleni z drugimi sestanki, ni enostavno izbrati takega termina, ki bi ustrezal vsem. Zato je za pomoč prosil svojega svetovalca za informatiko — tebe, da mu **napišeš program**, ki mu bo pomagal pri določitvi ustreznega časa za sestanek.

Vhod: vhodni podatki programa so urniki vseh oseb, ki bodo prisostvovale na sestanku, in želeno časovno obdobje, v katerem si Jurij želi sestanka. Oblika vhodne datoteke je naslednja:

f t d
s_{11} d_{11}
s_{12} d_{12}
. . .
0 0
s_{21} d_{21}
. . .
0 0
. . .
0 0

Števili f in t predstavljata meji časovnega intervala, znotraj katerih je možen sestanek (sestanek se ne sme začeti prej kot ob času f in se mora končati najkasneje do časa t). Število d je trajanje sestanka (v minutah). Nato so za vsakega sodelavca na sestanku navedeni drugi sestanki, zaradi katerih v določenih obdobjih nima časa. Vrednost s_{ij} je čas, ko se osebi i prične j-ti sestanek, d_{ij} pa je trajanje tega sestanka (v minutah). Seznam sestankov posameznega sodelavca se konča z vrstico, ki vsebuje dve ničli, čisto na koncu datoteke pa je še dodatna vrstica z dvema ničlama. Sodelavcev je največ sto in vsak od njih ima največ sto drugih sestankov. Vsi časi (f, t, s_{ij}) so navedeni kot število minut, ki so potekle od polnoči 6. aprila 2002. Vsi časi so med 6. aprilom 2002 in 1. januarjem 5500. Predpostaviš lahko, da za prehajanje z enega sestanka na drugega ljudje ne porabijo nič časa.

Izhod programa naj bo ena sama vrstica: število minut od polnoči 6. aprila 2002, ko naj se sestanek prične, tako da bodo lahko na celotnem sestanku

sodelovali vsi povabljeni in da bo sestanek kar se da kmalu. Če takšen termin ne obstaja, naj program izpiše „`SESTANEK NI MOZEN`“ (brez teh narekovajev).

Primera dveh vhodnih datotek:

```
10 30 5
0 15
25 10
0 0
10 10
0 0
0 0
```

```
10 30 6
0 15
25 10
0 0
10 10
0 0
0 0
```

Pripadajoči izhodni datoteki:

```
20
```

```
SESTANEK NI MOZEN
```

2002.3.5 Produkt števil

`produkt.in`, `produkt.out`

Dano je zaporedje celih števil: $\langle a_1, a_2, \ldots, a_{n-1}, a_n \rangle$; vsako od njih je ali enako 0 ali pa je oblike $\pm 2^k$ za kakšno nenegativno celo število k, $0 \leq k \leq 30$. R: 515

Napiši program, ki poišče tako podzaporedje $\langle a_i, a_{i+1}, \ldots, a_{j-1}, a_j \rangle$ (za neka i in j, pri katerih velja $1 \leq i \leq j \leq n$), da bo produkt števil $a_i \cdot a_{i+1} \cdot \ldots \cdot a_{j-1} \cdot a_j$ največji.

V prvi vrstici *vhodne datoteke* bo podano število n (zanj velja $1 \leq n \leq 100\,000$), v naslednji a_1, v naslednji a_2, itd., v zadnji ($(n+1)$-vi vrstici) pa a_n.

V *izhodno datoteko* napiši, v eni sami vrstici, števili i in j (ločeni s presledkom), pri katerih je produkt $a_i \cdot a_{i+1} \cdot \ldots \cdot a_j$ največji. Če je možnih več enako dobrih parov (i, j), je vseeno, katerega izpišeš.

Dva primera vhodnih datotek:

```
5
4
-4
8
-2
-8
```

Možni pripadajoči izhodni datoteki:

```
1 4
```

```
5                5 5
4
0
4
0
8
```

2002.3.6 **Prüferjev kod** `prufer.in`, `prufer.out`

R: 517

Drevo je graf, ki ga sestavljajo *vozlišča* (ali *točke*) in neusmerjene *povezave* med njimi, pri tem pa povezave nikoli ne tvorijo ciklov (pri sprehajanju iz neke točke ne moremo priti nazaj v isto točko drugače kot tako, da se obrnemo in gremo nazaj po isti poti, po kateri smo prišli), je pa mogoče iz vsake točke priti po enem ali več korakih do vsake druge (temu rečemo, da je graf *povezan*). Če sta dve vozlišči povezani s povezavo, pravimo, da sta *soseda*. Vozlišče, ki ima enega samega soseda, imenujemo *list*.

Profesor Hrast je bil v mladih letih zelo navdušen nad drevesi. Iznašel je postopek, s katerim je lahko drevo z N vozlišči zakodiral v zaporedje $N-2$ števil. (Dobljeno zaporedje je poimenoval „Prüferjev kod“ začetnega drevesa.) To je naredil takole:

1. Poljubno je oštevilčil vsa vozlišča drevesa s števili od 1 do N; vsakemu je pripisal drugo številko.
2. Potem je ponavljal naslednji postopek, dokler mu ni ostala le ena povezava:
   (a) poiskal je list z največjo številko,
   (b) ga izbrisal iz drevesa
   (c) in zapisal številko njegovega soseda.

Ta postopek se vedno izteče in nam da za različna drevesa vedno tudi različna zaporedja $N-2$ števil. Iz dobljenega zaporedja lahko vedno rekonstruiramo prvotno drevo.

Leta so minila, profesor se je postaral in v tem času so njegova drevesa precej zrastla. Ročno iskanje zaporedja postaja vedno bolj zamudno. Pomagaj profesorju in **napiši program**, ki bo prebral opis drevesa, izračunal njegov Prüferjev kod in izpisal tako dobljeno zaporedje $N-2$ števil. (Pravzaprav mora izvesti le korak 2, ker bo dobil vozlišča že oštevilčena.)[94]

Vhodna datoteka vsebuje v prvi vrstici število vozlišč, recimo mu N. Sledi za vsako povezavo po ena vrstica, v njej pa sta dve števili, ki povesta, kateri

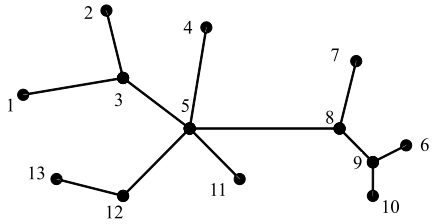
[94] Zanimiva naloga je tudi sestaviti postopek za dekodiranje, torej takega, ki bo znal zaporedja $N-2$ števil pretvarjati v drevesa (izpisal bi na primer seznam parov števil, ki predstavljajo povezave drevesa).

dve vozlišči povezuje ta povezava. Drevo z N vozlišči ima vedno $N-1$ povezav; vozlišča so oštevilčena s celimi števili od 1 do N. Povezave so neusmerjene, zato je lahko na primer povezava med vozliščema 12 in 34 navedena bodisi kot `12 34` ali pa kot `34 12`. Predpostaviš lahko, da velja $3 \le N \le 10000$.

V *izhodno datoteko* izpiši zaporedje $N-2$ števil, ki ga dobiš po zgoraj opisanem postopku iz drevesa, ki si ga prebral iz vhodne datoteke. Vsako število naj bo v svoji vrstici.

Primer: drevo na spodnji sliki nam da naslednje zaporedje:

$$\langle 12, 5, 5, 9, 8, 9, 8, 5, 5, 3, 3 \rangle.$$



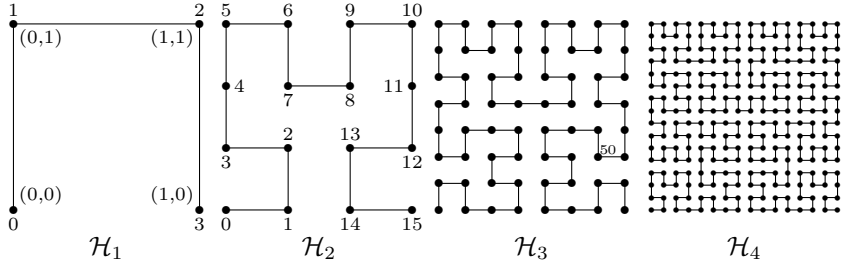
Ena od možnih vhodnih datotek za to drevo:	Pripadajoča izhodna datoteka:
`13`	`12`
`5 4`	`5`
`5 11`	`5`
`3 1`	`9`
`8 9`	`8`
`13 12`	`9`
`5 8`	`8`
`12 5`	`5`
`2 3`	`5`
`9 10`	`3`
`8 7`	`3`
`9 6`	
`3 5`	

2002.3.7 Hilbertova krivulja

`hilbert.in`, `hilbert.out`

Dano je zaporedje krivulj, recimo jim $\mathcal{H}_1, \mathcal{H}_2, \ldots$. Krivulja $\mathcal{H}_k$ poteka po karirasti mreži velikosti $2^k \times 2^k$ in obišče vsako točko v tej mreži natanko enkrat. Prvih nekaj krivulj je prikazanih na spodnji sliki. Vidimo, da lahko do $\mathcal{H}_{k+1}$ pridemo tako, da vzamemo štiri kopije krivulje $\mathcal{H}_k$, jih postavimo v kvadrat, spodnji dve zavrtimo drugo proti drugi in jih nato vse štiri povežemo.

R: 520



Pri določenem k lahko s pomočjo krivulje $\mathcal{H}_k$ točke na mreži $2^k \times 2^k$ oštevilčimo: začnimo v spodnjem levem kotu, tisti točki pripišimo številko 0, nato pa

se sprehajajmo vzdolž krivulje in pripisujmo točkam po vrsti vse višja cela števila. Končamo v točki v spodnjem desnem kotu, ki dobi številko $4^k - 1$.

Točke lahko opišemo tudi s koordinatami; tiste na spodnji vrstici imajo y-koordinato 0, tiste tik nad njimi 1, tiste v vrhnji vrstici pa $2^k - 1$. Podobno imajo tiste v najbolj levem stolpcu x-koordinato 0, tiste tik desno od njih 1, tiste v najbolj desnem stolpcu pa $2^k - 1$.

Primeri: na krivulji $\mathcal{H}_2$ ima točka s koordinatama $(1, 2)$ številko 7, tista s koordinatama $(2, 1)$ pa številko 13. Na krivulji $\mathcal{H}_3$ ima točka s koordinatama $(6, 2)$ številko 50.

Napiši program, ki iz *vhodne datoteke* prebere cela števila k, a in b, ki so vsa v prvi vrstici, ločena s po enim presledkom. Vedno velja $1 \le k \le 15$, $0 \le a < 2^k$ in $0 \le b < 2^k$. V *izhodno datoteko* naj tvoj program izpiše številko, ki jo na krivulji $\mathcal{H}_k$ dobi točka s koordinatama (a, b) (a je x-koordinata, b pa y-koordinata).[95]

Trije primeri vhodnih datotek:	Pripadajoče izhodne datoteke:
`1 0 0`	`0`
`3 6 2`	`50`
`10 1023 0`	`1048575`

2002.3.8 Slovar

`slovar.in`, `slovar.out`

R: 522

Polde se je že v svoji rani mladosti zasvojil s televizijo. Vsak dan je od jutra do večera presedel pred ekranom in se navduševal nad vsako še tako brezzvezno oddajo. Tako je bilo tudi ob nedeljah zvečer, vse dokler niso na Pok TVju uvedli novega besednega kviza. Kviz je Poldeta tako prevzel, da se je odločil v njem tudi sam sodelovati, ker pa se zaradi skromnega televizijskega besednega zaklada ni mogel potegovati za glavne nagrade, se sedaj obrača na vas, da mu spišete program, s katerim si bo lahko spomagal pri goljufanju.

Kviz zahteva, da na osnovi danega besedišča opišete najkrajši način, kako iz ene besede priti do druge, pri čemer lahko v vsakem koraku spremenite le eno črko, vsaka vmesna beseda pa mora tudi dejansko obstajati v slovarju. Možne besede v tem sprehodu morajo torej vedno imeti enako število črk.

Za primer vzemimo naslednji slovar (besede niso nujno urejene — niti po številu znakov niti po abecednem redu):

kot	*pot*	*ris*	*kapa*	*kepa*	*koča*	*milo*	*peka*
pesa	*reka*	*repa*	*roka*	*solo*	*šapa*	*šepa*	*teka*

[95] Zanimiva naloga je tudi pretvarjanje v nasprotno smer, torej iz zaporedne številke v koordinate.

Zanima nas, kakšen je najkrajši sprehod od besede *kapa* do *pesa*. Dobimo sprehode:

$$kapa \to kepa \to repa \to reka \to peka \to pesa;$$
$$kapa \to šapa \to šepa \to repa \to reka \to peka \to pesa;$$
$$kapa \to kepa \to repa \to reka \to teka \to peka \to pesa; \ldots$$

Najkrajši sprehod je očitno prvi, ki zahteva 5 korakov (število puščic „$\to$“).

Napiši program, ki na podlagi danega slovarja ter parov začetnih in končnih besed za vsak tak par poišče najkrajši sprehod od začetne do končne besede. Kot rezultat naj izpiše le število korakov ali pa „`Ni prehoda`“.

Vhod: v vhodni datoteki je najprej podan slovar; v prvi vrstici je celo število N, ki pove, koliko je besed v slovarju (vsaj 1, največ 10000). V naslednjih N vrsticah sledijo besede, vsaka v svoji vrstici; posamezne besede so dolge vsaj 2 in največ 10 znakov. Vse besede skupaj so dolge največ 55000 znakov. Parov besed, ki se razlikujejo le v eni črki, je največ 55000 (če parov $\{a, b\}$ in $\{b, a\}$ ne štejemo vsakega posebej) in pri tem nobena beseda ne nastopa v paru z več kot 30 drugimi. Vse besede so vedno podane le z malimi črkami angleške abecede, drugih znakov v besedah ni. — Sledi vrstica, ki vsebuje le celo število M, ki pove, koliko parov besed sledi ($1 \le M \le 10$). V naslednjih M vrsticah so pari besed „⟨*začetna beseda*⟩ ⟨*končna beseda*⟩“ (obe v eni vrstici, ločeni s presledkom). Obe besedi (začetna in končna) sta vedno vsebovani v slovarju.

Izhod: za vsak par besed v samostojni vrstici izpiši dolžino najkrajše poti oziroma niz „`Ni prehoda`“ (brez teh narekovajev), če poti med njima sploh ni.

Primer dveh vhodnih datotek:

```
6
kot
pot
pet
ris
kapa
kepa
3
kapa kepa
kot pet
kot ris
```

```
2
pet
pot
1
pet pot
```

Pripadajoči izhodni datoteki:

```
1
2
Ni prehoda
```

```
1
```

REŠITVE NALOG ZA PRVO SKUPINO

R2002.1.1 Pristanišče

N: 483 Naj bo z številka zabojnika, ki nas zanima. V vsaki plasti je $n \cdot m$ zabojnikov. Zabojniki v prvi plasti imajo torej številke od 1 do nm, tisti v drugi številke od $nm + 1$ do $2nm$ in tako naprej. Da pridemo do številke plasti, si lahko pomagamo z deljenjem: če delimo $z - 1$ z nm, nam celi del količnika pove številko plasti (le 1 ji moramo še prišteti, ker štejemo plasti od 1 naprej in ne od 0 naprej), ostanek (recimo mu s) pa si lahko mislimo kot številko zabojnika znotraj plasti: za tiste v prvi vrsti dobimo številke od 0 do $n - 1$, za tiste v drugi vrsti številke od n do $2n - 1$ in tako naprej. Do številke vrste in položaja znotraj vrste pridemo zdaj na podoben način, torej spet z deljenjem.

```
var N, M, L, Z, S: integer;
begin
  ReadLn(N, M, L, Z);
  WriteLn('Plast: ', 1 + (Z − 1) div (N * M));
  S := (Z − 1) mod (N * M);
  WriteLn('Vrsta: ', 1 + S div N);
  WriteLn('Stolpec: ', 1 + S mod N);
end.
```

R2002.1.2 Najdaljši cikel

N: 484 Če začnemo na kateri koli postaji (najprej recimo kar na prvi) in sledimo navodilom iz tabele Postaje, bomo sčasoma obhodili ves cikel in prišli nazaj na postajo, pri kateri smo začeli. Ob tem lahko izpisujemo postaje te proge in tudi štejemo, koliko postaj smo obhodili. Nato začnemo pri kakšni postaji, ki je doslej še nismo obiskali, in na enak način poiščemo naslednjo progo. Ob vsaki progi še preverimo, če je mogoče daljša od doslej najdaljše znane. V tabeli Obiskana si zapisujemo, katere postaje smo že obiskali.

```
const N = ...;
var Postaje: array [1..N] of integer;
    i, j, Dolzina, Najdaljsa, StProg: integer;
    Obiskana: array [1..N] of boolean;
begin
  for i := 1 to N do Obiskana[i] := false;
  Najdaljsa := 0; StProg := 0;
  for i := 1 to N do if not Obiskana[i] then begin
    Dolzina := 0; j := i;
    StProg := StProg + 1; Write('Proga ', StProg, ':');
```

```
    repeat
      Dolzina := Dolzina + 1;
      Obiskana[j] := true; Write(' ', j);
      j := Postaje[j];
    until j = i;
    WriteLn(', dolžina: ', Dolzina);
    if Dolzina > Najdaljsa then Najdaljsa := Dolzina;
  end; {for, if}
  WriteLn('Dolžina najdaljše proge: ', Najdaljsa, '.');
end.
```

R2002.1.3 Razbijanje kode

N: 484

Ker so kode omejene le na štiri znake, lahko uporabimo kar štiri gnezdene zanke. Ker je kod le $26^4 = 456974$, bi jih lahko tudi našteli kot števila in vsako pretvorili (podobno kot pri nalogi o zabojnikih) v štiri znake dolg niz, ki bi ga potem podali funkciji Test. Lahko bi uporabili tudi različico malo splošnejšega postopka, kakršnega potrebujemo pri nalogi „Razbijanje gesel“ za drugo skupino.

```
program IskanjeGesla;

  function Test(Geslo: string): boolean; external;

var A, B, C, D: char;
    S: string[4];
begin
  S := '....';
  for A := 'A' to 'Z' do begin S[1] := A;
  for B := 'A' to 'Z' do begin S[2] := B;
  for C := 'A' to 'Z' do begin S[3] := C;
  for D := 'A' to 'Z' do begin S[4] := D;
    if Test(S) then begin WriteLn(S); exit end;
  end; end; end; end;
end. {IskanjeGesla}
```

R2002.1.4 Bencin

N: 485

Na vsaki črpalki pogledamo, katera je naslednja cenejša (in kako daleč je do nje). Če potrebujemo do nje polno posodo goriva ali manj, natočimo toliko bencina, da ga bomo imeli ravno dovolj za do tam (če imamo bencina že od prej dovolj, ga mogoče sploh ni treba dotakati), sicer pa polno. Če na poti ne bo nobene cenejše črpalke več, natočimo ravno dovolj goriva za do Bruslja (ali pa poln tank, če bi potrebovali več); to pravilo lahko gledamo kot poseben primer prejšnjega, če si mislimo, da je v Bruslju še ena črpalka, kjer dajejo bencin zastonj.

Prepričajmo se, da je ta načrt dotakanja bencina (recimo mu A) res najboljši. Recimo, da obstaja nek še boljši razpored B. Glede dotakanja na prvih nekaj črpalkah se mogoče ujema z A, prej ali slej pa mora nastopiti razlika (saj bi bila drugače razporeda čisto enaka). Recimo, da je prva razlika na i-ti črpalki; do sem torej A in B prideta z enako količino bencina v tanku in si do sem nakopljeta enake stroške, na i-ti črpalki pa dolijeta različno količino goriva.

Če napolni A tank do konca, pomeni, da B vzame tu manj bencina kot A; toda A napolni tank do konca le v primerih, ko se do naslednje cenejše črpalke (recimo ji j) ne da priti z manj bencina; recimo torej, da se za pot do j porabi poln tank in še x litrov bencina; B bo moral torej po poti doliti več kot x litrov, ker zapušča i z manj kot polnim tankom; ker pa so vse črpalke med i in j vsaj tako drage kot i (če ne še dražje), se ne bi B-jev razpored nič poslabšal, če bi na i natočil poln tank in nato pri naslednjem dotakanju malo manj.

Če pa A na črpalki i ne napolni tanka do konca, pomeni, da je natočil le toliko, kolikor potrebuje do naslednje cenejše črpalke (spet ji recimo j). Če je B natočil manj, bo moral dotakati nekje vmes (ker črpalke i ne zapušča z dovolj bencina, da bi dosegel j), tam pa bo bencin vsaj tako drag kot pri i; torej ne bi bil B nič na slabšem, če bi pri i natočil toliko kot A in nato med i in j ne bi nič dotakal (kot tudi A ne bo). Če pa je B natočil na i več kot A, ima ob odhodu z i več bencina, kot je potrebno za pot do j; torej bi lahko nekaj prihranil, če bi natočil na i malo manj in to razliko dotočil pri j, saj je tam bencin cenejši, za pot od i do j pa bi ga bilo vseeno dovolj; toda to je potemtakem sploh nemogoče, saj smo na začetku rekli, naj bo B najboljši razpored.

Tako torej vidimo: če prva razlika med najboljšim razporedom B in našim razporedom A nastopi pri i-ti črpalki, je mogoče B spremeniti v nek razpored B', ki ni nič slabši od B, se pa z A-jem ujema tudi na i-ti črpalki. Če zdaj ta korak ponavljamo, vidimo, da lahko B predelamo v A, ne da bi se kaj poslabšal, torej je tudi A najboljši razpored.

Takšen način dokazovanja, da je naš razpored najboljši (torej da predpostavimo nek boljši razpored in se potem prepričamo, da bi ga lahko postopoma spremenili v naš razpored, ne da bi se pri tem kaj poslabšal, kar torej pomeni, da ni naš razpored nič slabši od najboljšega možnega in je zato tudi sam eden od najboljših), je zelo značilen za požrešne algoritme (*greedy algorithms*), med katere spada tudi naš gornji postopek.

```
const N = ...;                          { Število črpalk. }
var   Kje, Cena: array [1..N] of real; { Položaj črpalk in cena goriva na njih. }
      Poraba: real;                     { Poraba goriva v litrih na 100 km. }
      Posoda: real;                     { Velikost posode za gorivo v litrih. }
      Pot: real;                        { Skupna dolžina poti v kilometrih. }

var i, j: integer; Gorivo, Razd, Koliko: real;
```

```
begin
  Gorivo := 0; i := 1;
  while i <= N do begin
    j := i + 1; { j bo naslednja cenejša črpalka. }
    while j <= N do
      if Cena[j] < Cena[i] then break
      else j := j + 1;
    { Koliko goriva potrebujemo do naslednje cenejše črpalke? }
    if j > N then Razd := Pot - Kje[i] else Razd := Kje[j] - Kje[i];
    Koliko := (Razd / 100.0) * Poraba;
    if Gorivo < Koliko then begin { Treba bo dotočiti. }
      if Koliko <= Posoda then begin { Natočimo dovolj za pot do j. }
        WriteLn('Na črpalki ', i, ' dotočimo ', (Koliko - Gorivo):0:2, ' l.');
        Gorivo := 0; i := j;
      end else begin { Natočimo poln tank, a ne bo dovolj do j. }
        WriteLn('Na črpalki ', i, ' dotočimo ', (Posoda - Gorivo):0:2, ' l.');
        Gorivo := Posoda - ((Kje[i + 1] - Kje[i]) / 100.0) * Poraba; i := i + 1;
      end; {if}
    end else begin { Imamo dovolj do j. }
      Gorivo := Gorivo - Koliko; i := j;
    end; {if}
  end; {while}
end.
```

Gornji program pa ima še eno slabost: vgnezdeno zanko za iskanje naslednje cenejše črpalke. Če je vsaka črpalka dražja od prejšnje, bo šel j vsakič od i + 1 vse do N + 1; in če so črpalke obenem dovolj daleč narazen, da jih ne bomo mogli preskakovati, ampak bomo na vsaki dotakali, se bo zunanja zanka res izvedla N-krat. To dvoje skupaj pomeni, da ima naš postopek v najslabšem primeru časovno zahtevnost $O(N^2)$. V praksi do tega verjetno ne bi prišlo, pa tudi če bi, bi lahko na današnjih računalnikih vseeno dovolj hitro obdelali tudi poti z več tisoč črpalkami; kljub temu pa poskusimo razmisliti še o učinkovitejši rešitvi.

Recimo, da bi si hoteli na začetku v tabeli Nasl pripraviti za vsako črpalko podatek o naslednji najcenejši črpalki. Postopek bi bil lahko tak:

```
Iščemo := {};
for i := 1 to N do
  for j ∈ Iščemo do
    if Cena[i] < Cena[j] then
      Nasl[j] := i; Iščemo := Iščemo - {j};
  Iščemo := Iščemo ∪ {i};
```

Vzdržujemo torej množico črpalk, za katere še nismo našli naslednje cenejše; pri vsaki črpalki to množico pregledamo in zbrišemo iz nje tiste črpalke, od katerih je trenutna (i-ta) črpalka cenejša. Nato še i dodamo v množico, da

bomo v nadaljevanju poiskali tudi naslednjo cenejšo za njo. Na koncu tega postopka ostanejo v množici *Iščemo* še črpalke, za katerimi ni nobene cenejše.

Kako naj v praksi predstavimo množico *Iščemo*? Lahko bi uporabili seznam črpalk, urejen po padajočih cenah. Če je namreč i sploh cenejša od katere izmed črpalk v *Iščemo*, bo gotovo cenejša od najdražjih nekaj. Pri vsakem i bi pregledovali ta seznam od najdražjih k cenejšim; dokler opažamo take, ki so dražje od i, jih brišemo iz seznama, čim pa naletimo na cenejšo od i, se ustavimo, saj takrat vemo, da ni v seznamu nobene več dražje od i. To pa tudi pomeni, da je i vsaj tako draga kot katera koli druga v seznamu in jo lahko dodamo kar na začetek seznama. Zdaj pa, ko smo ugotovili, da bomo vedno dodajali in brisali le pri začetku seznama, vidimo, da pravzaprav ni treba pisati res splošnega seznama; dovolj bo že čisto navaden sklad. Začetek gornjega programa bi torej lahko spremenili takole:

```
var i, j: integer; Gorivo, Razd, Koliko: real;
    Sklad, Nasl: array [1..N] of integer;
begin
  j := 0;
  for i := 1 to N do begin
    while j > 0 do { Od katerih na skladu je i cenejša? }
      if Cena[Sklad[j]] <= Cena[i] then break
      else begin Nasl[Sklad[j]] := i; j := j - 1 end;
    j := j + 1; Sklad[j] := i; { Dodajmo še i na sklad. }
  end; {for}
  while j > 0 do { Ostale so črpalke brez naslednje cenejše. }
    begin Nasl[Sklad[j]] := N + 1; j := j - 1 end;
  Gorivo := 0; i := 1;
  while i <= N do begin
    j := Nasl[i];
    { Koliko goriva potrebujemo do naslednje cenejše črpalke? }
    ...
```

Zdaj vsaka iteracija zanke **while** j bodisi vzame neko črpalko s sklada ali pa se ustavi (kliče **break**). Poleg tega dodamo vsako črpalko na sklad le enkrat. Skupaj imamo torej največ N dodajanj in N brisanj, pa še največ N takih iteracij zanke **while** j, ki kličejo **break**. Tako predelan postopek ima časovno zahtevnost $O(N)$.

REŠITVE NALOG ZA DRUGO SKUPINO

R2002.2.1 Kodiranje

N: 485

Da bo povprečno besedilo predstavljeno s čim krajšim zaporedjem bitov, je koristno, če dobijo pogostejše črke krajše kode kot redkejše črke. Če bi imeli štiri črke in bi se vsaka pojavljala v 25 % primerov, bi bilo gotovo najbolj smiselno dati vsaki po eno dvobitno kodo; če bi imeli osem črk s pogostostjo pojavitev 12,5 %, bi dali vsaki eno trobitno kodo in tako naprej. Mi imamo dve črki s pogostostjo 25 %, namreč c in e, zato jima dodelimo dvobitni kodi; c-ju na primer 00, e-ju pa 01. Črki a in g, ki se pojavita vsaka v 12,5 % primerov, pokrijeta torej skupaj 25 % primerov in je zato dobro, da obe skupaj dobita tretjo dvobitno kodo, npr. 10; da ju bomo lahko ločili med sabo, pa dodajmo temu še tretji bit in torej a kodirajmo s 100, g pa s 101. Preostale štiri črke s pogostostjo 6,25 % vse skupaj tudi pokrijejo 25 % primerov in jim zato dodelimo še četrto dvobitno kodo, 11, ter vsaki še po dva bita, da bodo kode enolične. Dobimo takšen kod:

Črka	a	b	c	d	e	f	g	h
Koda	100	1100	00	1101	01	1110	101	1111

Opazimo lahko pomembno lastnost, da nobena od teh osmih kod ni podaljšek kakšne druge; drugače bi bilo mogoče nekatera zaporedja bitov težko dekodirati v prvotna besedila ali pa bi se lahko celo zgodilo, da bi se več besedil zakodiralo v isto zaporedje bitov.

Splošnejša oblika opisanega razmisleka se imenuje *Huffmanovo kodiranje*.

R2002.2.2 Prijatelji in sovražniki

N: 486

Za vsako osebo vzdržujmo podatek o tem, ali je Jurijev prijatelj in ali je njegov sovražnik. Ti podatki imajo lahko dve stanji: ali smo že ugotovili, da je ta oseba prijatelj (oz. sovražnik) ali pa še nismo (slednje lahko pomeni, da ta človek res ni naš prijatelj/sovražnik ali pa da sicer je, vendar tega še nismo odkrili). Ko o neki osebi na novo izvemo, da je prijatelj ali sovražnik, je treba preiskati še njene neposredne prijatelje in neposredne sovražnike, ker lahko zdaj tudi o teh mogoče izvemo, v kakšni zvezi so z Jurijem. Zato imejmo še seznam (pravzaprav sklad) oseb, ki si jih bo treba še pobliže ogledati. Vsaka oseba lahko vstopi na ta seznam največ dvakrat: ko izvemo, da je Jurijev prijatelj, in ko izvemo, da je njegov sovražnik. Celoten postopek lahko poženemo tako, da dodamo Jurija na seznam in označimo, da je prijatelj samemu sebi.

program ZunanjaPolitika;

```
  const N = ...;
  type KdoJeKdo = (Neznanec, Prijatelj, Sovrag);
  var Odnos: array [0..N − 1, 0..N − 1] of KdoJeKdo;

var Prij, Sovr: array [0..N − 1] of boolean; Dodaj: boolean;
    Seznam: array [0..2 * N − 1] of integer; i, j, Dolzina: integer;
begin
  for i := 0 to N − 1 do begin Prij[i] := false; Sovr[i] := false end;
  Seznam[0] := 0; Dolzina := 1; Prij[0] := true;
  while Dolzina > 0 do begin
    Dolzina := Dolzina − 1; i := Seznam[Dolzina];
    for j := 0 to N − 1 do begin
      Dodaj := false;
      if (   (Prij[i] and (Odnos[i, j] = Prijatelj))
          or (Sovr[i] and (Odnos[i, j] = Sovrag)) )  and not Prij[j] then
          begin Prij[j] := true; Dodaj := true end;
      if (   (Prij[i] and (Odnos[i, j] = Sovrag))
          or (Sovr[i] and (Odnos[i, j] = Prijatelj)) )  and not Sovr[j] then
          begin Sovr[j] := true; Dodaj := true end;
      if Dodaj then begin Seznam[Dolzina] := j; Dolzina := Dolzina + 1 end;
    end; {for}
  end; {while}
  WriteLn('Pravi prijatelji:');
  for i := 1 to N − 1 do if Prij[i] and not Sovr[i] then WriteLn(i);
  WriteLn('Pravi sovražniki:');
  for i := 1 to N − 1 do if Sovr[i] and not Prij[i] then WriteLn(i);
end. {ZunanjaPolitika}
```

R2002.2.3 Razbijanje gesel

N: 487

Gesla lahko generiramo s podprogramom, ki rekurzivno kliče samega sebe, da doda geslu še naslednjo črko, če pa je geslo že dovolj dolgo, lahko kliče tudi funkcijo Test. Pazimo še na to, da mora imeti geslo vsaj eno števko in vsaj eno črko, zato lahko, ko dodajamo zadnji znak, nekaj možnosti preskočimo, da ne bi funkcije Test po nepotrebnem klicali prevečkrat.

```
program IskanjeGesla;

  function Test(S: string): boolean; external;

const Najkrajse = 3; Najdaljse = 8;
type KajIma = set of (Crko, Stevko);
var Geslo: string; Nasel: boolean;

procedure Test2(Dolzina: integer);
begin
  Geslo[0] := Chr(Dolzina);
```

```
  if Test(Geslo) then begin WriteLn(Geslo); Nasel := true end;
end; {Test2}

procedure Generiraj(Mesto: integer; Ima: KajIma);
var i, Mala: integer;
begin
  { Geslo ima zdajle Mesto − 1 znakov. Kot naslednji znak mu lahko
    dodamo črko, razen če je to zadnji znak in ima geslo doslej same
    črke; če to ni zadnji znak in ima same črke, bomo lahko kasneje
    dodali še kakšno številko, tako da ni narobe, če zdaj dodamo črko. }
  if (Mesto < Najdaljse) or (Stevko in Ima) then
    for Mala := 0 to 1 do for i := 0 to 25 do begin
      Geslo[Mesto] := Chr(Ord('A') + Mala * (Ord('a') − Ord('A')) + i);
      { Geslo je zdaj dolgo Mesto znakov. Če ga lahko še podaljšujemo,
        rekurzivno kličimo podprogram Generiraj, sicer pa geslo le
        preizkusimo (s podprogramom Test2). }
      if Mesto = Najdaljse then Test2(Mesto)
                           else Generiraj(Mesto + 1, Ima + [Crko]);
      if Nasel then exit;
    end; {for}
  { Na enak način kot črko lahko lahko poskusimo dodati tudi številko. }
  if (Mesto < Najdaljse) or (Crko in Ima) then
    for i := 0 to 9 do begin
      Geslo[Mesto] := Chr(Ord('0') + i);
      if Mesto = Najdaljse then Test2(Mesto)
                           else Generiraj(Mesto + 1, Ima + [Stevko]);
      if Nasel then exit;
    end; {for}
  { Ostane še možnost, da gesla ne podaljšujemo in ostanemo le pri
    dosedanjih Mesto − 1 znakih. Seveda moramo še vseeno preveriti,
    da geslo ni prekratko in da vsebuje tako črke kot številke. }
  if (Mesto > Najkrajse) and (Ima = [Crko, Stevko]) then Test2(Mesto − 1);
end; {Generiraj}

begin
  Generiraj(1, []);
end. {IskanjeGesla}
```

Na voljo nam je 52 različnih črk in 10 različnih številk; torej obstaja 62^n gesel dolžine n, vendar pa jih je med temi 10^n takih, ki so iz samih številk, in 52^n takih, ki so iz samih črk. Upoštevajmo še, da je $\sum_{n=0}^{N-1} a^n = (a^N - 1)/(a-1)$ in zato $\sum_{n=b}^{c} a^n = (a^{c+1} - a^b)/(a-1)$; vseh sprejemljivih gesel dolžine od 3 do 8 je torej

$$\frac{62^9 - 62^3}{62-1} - \frac{52^9 - 52^3}{52-1} - \frac{10^9 - 10^3}{10-1} = 167\,411\,381\,963\,280.$$

Če imamo srečo, bomo sicer že ob prvem poskusu zadeli pravo geslo, v najslabšem primeru pa bo pravo šele zadnje in v povprečju lahko pričakujemo (če so res vsa enako verjetna), da jih bomo morali preizkusiti približno polovico. Četudi bi jih lahko v sekundi preizkusili milijardo, bi to trajalo skoraj ves dan. Zato ta pristop k odkrivanju gesel verjetno ni sprejemljiv.

R2002.2.4 TiVo

N: 488 Podatkovna struktura je krožni izravnalni pomnilnik, za katerega potrebujemo le kazalce na njegov začetek, na konec in na trenutno mesto, s katerega predvajamo posneto sliko. Da lažje primerjamo kazalce med seboj, vedno ohranjamo urejenost: Najstarejsi $\leq$ Prikazani $\leq$ Najnovejsi, ko pa moramo sporočiti številko bloka, jo preračunamo po modulu velikosti diska. Da nam vrednosti ne pobegnejo v neskončnost, jih brzdamo, vendar vse tri kazalce hkrati glede na najmanjšega.

Če pri predvajanju pridemo do najnovejše ali najstarejše slike, obtičimo pri njej. Če smo pri najstarejši, nam najstarejše kar naprej bežijo in je najbolje, da kar preklopimo na normalno hitrost predvajanja v izogib neenakomernim skokom zaradi morebitne časovne neusklajenosti snemanja in predvajanja. Podobno naredimo, če se s povečano hitrostjo zaletimo ob najnovejši rob.

```
const StBlokov = 300001;

{ Velja urejenost: Najstarejsi ≤ Prikazani ≤ Najnovejsi,
  dejanska številka bloka pa je po modulu StBlokov. }
var Najstarejsi: integer value 0;
    Najnovejsi: integer value 0;
    Prikazani: integer value 0;
    Shranjenih: integer value 0;    { Število shranjenih sličic. }
    Hitrost: integer value 1;       { Hitrost predvajanja (1 = normalno). }

function KamShranitiNovo: integer;
begin
  Najnovejsi := Najnovejsi + 1; { Glava se premakne. }
  if Shranjenih < StBlokov then Shranjenih := Shranjenih + 1
  else begin { Rep prirežemo. }
    Najstarejsi := Najstarejsi + 1;
    if Prikazani <= Najstarejsi then { Raje enega več, kot bi bilo nujno. }
      begin Prikazani := Najstarejsi + 1; Hitrost := 1 end;
    if Najstarejsi >= StBlokov then begin { Naokrog, da ne gre v neskončnost. }
      Prikazani := Prikazani - StBlokov;
      Najnovejsi := Najnovejsi - StBlokov;
      Najstarejsi := Najstarejsi - StBlokov;
    end; {if}
  end; {if}
  KamShranitiNovo := Najnovejsi mod StBlokov;
```

```
end; {KamShraniti Novo}

procedure IzberiNacinPredvajanja(NovaHitrost: integer);
  begin Hitrost := NovaHitrost end;

function KateroSlikoPrikazati: integer;
begin
  Prikazani := Prikazani + Hitrost;
  if Hitrost > 0 then begin
    { Rep nas ne more ujeti, lahko pa se zaletimo v najnovejšega. }
    if Prikazani >= Najnovejsi then { Bolje se je ustaviti tik pred njim. }
      begin Prikazani := Najnovejsi - 1; Hitrost := 1 end;
  end else begin
    { Stojimo ali gremo nazaj, a pred najstarejšim moramo bežati. }
    if Prikazani <= Najstarejsi then
      begin Prikazani := Najstarejsi + 1; Hitrost := 1 end;
  end; {if}
  KateroSlikoPrikazati := Prikazani mod StBlokov;
end; {KateroSlikoPrikazati}
```

REŠITVE NALOG ZA TRETJO SKUPINO

R2002.3.1 Limuzine

N: 491

Recimo, da traja sestanek t minut, ura vožnje nas stane v tolarjev, do parkirišča i je t_i minut vožnje in cena ure parkiranja na njem je c_i tolarjev. Parkiranje na parkirišču i je potem možno le, če je $2t_i < t$, stane pa nas $2t_i \cdot v + (t - 2t_i) \cdot c_i$ šestdesetink tolarja. Možnost, da sploh ne parkiramo, ampak se le vozimo naokoli, pa nas stane $t \cdot v$ šestdesetink tolarja. Zdaj ni treba drugega, kot da ugotovimo, kaj od tega je najceneje. Spodnji program si v Najceneje zapisuje stroške za doslej najcenejšo najdeno različico, v NajKje oz. NajCas pa sta številka parkirišča oz. čas parkiranja pri tej različici.

```
program Limuzine;
var T: text; CenaVoz, DolzSest, CenaPark, CasVoz, CasPark: longint;
    Cena, Najceneje, NajKje, NajCas, StPark: longint;
begin
  Assign(T, 'limo.in');
  Reset(T); ReadLn(T, CenaVoz, DolzSest);
  Najceneje := CenaVoz * DolzSest; NajKje := 0; NajCas := 0; StPark := 0;
  while true do begin { berimo parkirišča }
    ReadLn(T, CenaPark, CasVoz); StPark := StPark + 1;
    if CenaPark <= 0 then break; { konec vhodne datoteke }
    CasPark := DolzSest - 2 * CasVoz;
    if CasPark > 0 then begin { poskusimo parkirati tukaj }
```

```
        Cena := CenaPark * CasPark + CenaVoz * 2 * CasVoz;
        if Cena < Najceneje then begin { novo najcenejše parkirišče }
          Najceneje := Cena; NajKje := StPark;
          NajCas := DolzSest - 2 * CasVoz;
        end; {if}
      end; {if}
    end; {while}
    Close(T);

    Assign(T, 'limo.out');
    Rewrite(T); WriteLn(T, NajKje, ' ', NajCas); Close(T);
end. {Limuzine}
```

R2002.3.2 Uvrstitve tekmovalcev

N: 492

Sprehodimo se skozi zaporedje in si za vsakega tekmovalca zapomnimo, koliko krogov je naredil in kdaj je naredil zadnjega. Nato jih uredimo padajoče po številu krogov, znotraj istega števila krogov pa po naraščajočih časih. V tem vrstnem redu jih izpišemo. Spodnji program si pomaga s strukturo KolesarT, ki hrani za posameznega tekmovalca poleg štartne številke (St) še število prevoženih krogov (StKrogov) in čas, ko je prevozil zadnjega od njih (Cas; to v resnici ni čisto pravi čas, pač pa indeks znotraj zaporedja prehodov skozi ciljno črto, kar pa nam že zadostuje, da lahko ugotovimo, kdo je prej prevozil svoj zadnji krog).

```
program Tekma;
const MaxN = 2000;
type KolesarT = record St, StKrogov: integer; Cas: longint end;
var T: text; j, k, N: integer; Zdaj: longint;
    Kolesarji: array [1..MaxN] of KolesarT; Kol: KolesarT;
begin
  { Preberimo podatke o tekmi. }
  Assign(T, 'tekma.in');
  Reset(T); ReadLn(T, N); Zdaj := 0;
  for k := 1 to N do with Kolesarji[k] do
    begin St := k; StKrogov := 0; Cas := Zdaj end;
  while true do begin
    ReadLn(T, k); Zdaj := Zdaj + 1; if k <= 0 then break;
    with Kolesarji[k] do begin StKrogov := StKrogov + 1; Cas := Zdaj end;
  end; {while}
  Close(T);

  { Uredimo kolesarje po rezultatih. }
  for k := 2 to N do begin
    Kol := Kolesarji[k]; j := k - 1;
    { Prvih k - 1 celic tabele Kolesarji je že urejenih. Vrinimo kolesarja k
```

```
      { na pravo mesto v ta del tabele (torej pred vse take, ki so se na tekmi odrezali }
    while j > 0 do with Kolesarji[j] do                    { slabše od njega). }
      { Je bil kolesar j boljši od trenutnega? Če da, moramo nehati in
        vpisati trenutnega na (j + 1)-vo mesto; če pa ne, se bo trenutni
        uvrstil pred j in moramo kolesarja j premakniti za eno mesto naprej. }
      if StKrogov > Kol.StKrogov then break
      else if (StKrogov = Kol.StKrogov) and (Cas < Kol.Cas) then break
      else begin Kolesarji[j + 1] := Kolesarji[j]; j := j - 1 end;
    Kolesarji[j + 1] := Kol;
  end; {for k}

  { Izpišimo rezultate. }
  Assign(T, 'tekma.out'); Rewrite(T);
  for k := 1 to N do if Kolesarji[k].StKrogov > 0 then
    WriteLn(T, Kolesarji[k].St);
  Close(T);
end. {Tekma}
```

Postopek, ki smo ga uporabili za urejanje kolesarjev, se imenuje urejanje z vstavljanjem (*insertion sort*). Ko se začnemo ukvarjati s kolesarjem k, imamo kolesarje $1, \ldots, k-1$ že urejene v pravem vrstnem redu. Potem gremo od konca tega zaporedja proti začetku in dokler srečujemo kolesarje, slabše od k, jih premikamo za eno mesto naprej; takoj ko naletimo na takega, ki je boljši od k, pa vstavimo kolesarja k v izpraznjeno celico tik za njim. Tako imamo zdaj urejeno zaporedje kolesarjev $1, \ldots, k$ in se lahko lotimo kolesarja $k+1$.

R2002.3.3 Število vsot

Naj bo $a(n,k)$ število načinov, na katere lahko n zapišemo kot vsoto k števil. N: 493
Glavni razmislek, ki ga moramo opraviti, da pridemo do rešitve, je naslednji: vsak razcep n-ja na vsoto k števil ima ali vsaj en seštevanec enak 1 ali pa vse seštevance večje od 1. V prvem primeru predstavljajo ostali seštevanci razcep števila $n-1$ na vsoto $k-1$ števil, teh razcepov pa je $a(n-1,k-1)$; v drugem primeru pa bi, če bi vsakega od seštevancev zmanjšali za 1, dobili nek razcep števila $n-k$ na vsoto k števil, takih razcepov pa je $a(n-k,k)$. Torej je $a(n,k) = a(n-1,k-1) + a(n-k,k)$. Posebni (robni) primeri so: $a(n,k) = 0$ za $n < k$; $a(0,0) = 1$; in $a(n,0) = 0$ za $n > 0$.

Da se dokopljemo do vrednosti $a(N,K)$, ki nas zanima, moramo prej izračunati vrednosti $a(n,k)$ za n od 1 do N in za k od 1 do K. To lahko naredimo z dvema gnezdenima zankama, ki gresta po n od 1 do N in pri vsakem n-ju še po k od 1 do $\min\{n,K\}$ (ker števila n ne moremo zapisati kot vsoto več kot n števil, vsote več kot K števil pa nas ne zanimajo).

```
program SteviloVsot;
const MaxN = 100;
```

```
var T: text; ni, ki, K, N: integer; a: array [0..MaxN, 0..MaxN] of longint;
begin
  Assign(T, 'vsote.in'); Reset(T); ReadLn(T, N, K); Close(T);

  { Izračunajmo. }
  a[0, 0] := 1;
  for ni := 1 to N do begin
    a[ni, 0] := 0; ki := 1;
    while (ki <= ni) and (ki <= K) do begin
      a[ni, ki] := a[ni - 1, ki - 1] + a[ni - ki, ki];
      ki := ki + 1;
    end; {while ki}
  end; {for ni}

  { Izpišimo rezultat. }
  Assign(T, 'vsote.out'); Rewrite(T); WriteLn(T, a[N, K]); Close(T);
end. {SteviloVsot}
```

Gornji program mora hraniti v tabeli a vrednosti $a(n,k)$ za vse doslej izračunane pare (n,k), zato je njegova prostorska (pomnilniška) zahtevnost reda $O(NK)$. To lahko zmanjšamo na samo $O(N)$, če računamo vrednosti $a(n,k)$ po naraščajočih k in pri vsakem k še po vseh potrebnih n (od k do N). Pri tem vrstnem redu bomo, ko enkrat dosežemo določeno vrednost k, potrebovali od starih rezultatov le še tiste za $k-1$, starejših (za $k-2$, $k-3$ itd.) pa ne. Spodnji program ima v tabeli a prostora le za vrednosti $a(n,k)$ (v celici a[j, n]) in $a(n,k-1)$ (v celici a[1 − j, n]). Ko pridemo pri trenutnem k do konca, lahko s prireditvijo j := 1 − j dosežemo tak učinek, kot da bi se vrstici tabele a zamenjali; rezultati, ki jih bomo računali pri $k+1$, se bodo vpisovali čez stare rezultate za $k-1$, ki jih ne bomo več potrebovali.

```
program StevilоVsot2;
const MaxN = 100;
var T: text; ni, ki, K, N, j: integer; a: array [0..1, 0..MaxN] of longint;
begin
  Assign(T, 'vsote.in'); Reset(T); ReadLn(T, N, K); Close(T);

  { Izračunajmo. }
  j := 0; a[j, 0] := 1;
  for ki := 1 to K do begin
    j := 1 - j;
    for ni := ki to N do begin
      a[j, ni] := a[1 - j, ni - 1];
      if ni - ki >= ki then a[j, ni] := a[j, ni] + a[j, ni - ki];
    end; {for ni}
  end; {for ki}

  { Izpišimo rezultat. }
  Assign(T, 'vsote.out'); Rewrite(T); WriteLn(T, a[j, N]); Close(T);
end. {StevilоVsot2}
```

Kot zanimivost lahko omenimo, da je največje število, s katerim imamo opravka pri tej nalogi, $a(100, 18) = 11\,087\,828$. Vsota $A(n) := \sum_{k=1}^{n} a(n, k)$ pa se pri velikih n obnaša približno tako kot[96] $\frac{1}{4n\sqrt{3}} \cdot e^{\pi\sqrt{2n/3}}$; $A(100) = 190\,569\,292$.

R2002.3.4 Sestanki

Začetne in končne čase vseh sestankov si uredimo v naraščajočem vrstnem redu, pri vsakem pa si še zapomnimo, ali gre za začetni ali za končni čas. Če se nato sprehajamo po tem vrstnem redu, bomo za vsako obdobje med dvema takima časoma vedeli, koliko sestankov je takrat v teku (števec sestankov povečamo vsakič, ko naletimo na začetni čas nekega sestanka, in ga zmanjšamo ob vsakem končnem času). Čim najdemo dovolj dolg interval, ko ni v teku nobenih drugih sestankov, lahko tja postavimo naš novi sestanek in nehamo. Da upoštevamo še pogoja f in t, lahko dodamo še 0 kot začetni in f kot končni čas. Če pridemo do konca zaporedja, ne da bi našli dovolj dolg prost interval s koncem pred časom t, lahko zaključimo, da sestanek ni mogoč.

N: 494

```
program Sestanki1;
const MaxSest = 10000;
var T: text; Casi: array [1..2 * MaxSest + 2] of longint; i, StCasov, StSest: integer;

  procedure Dodaj(Cas: longint);
  var i: integer;
  begin
    i := StCasov; while i >= 1 do
      if Abs(Casi[i]) < Abs(Cas) then break
      else begin Casi[i + 1] := Casi[i]; i := i - 1 end;
    Casi[i + 1] := Cas; StCasov := StCasov + 1;
  end; {Dodaj}

var MinCas, MaxCas, Dolz, Zac, Trajanje: longint;
begin
  { Preberimo vhodne podatke. }
  Assign(T, 'sestanki.in');
  Reset(T); ReadLn(T, MinCas, MaxCas, Dolz); StSest := 0;
  while true do begin
    ReadLn(T, Zac, Trajanje);
    if Zac + Trajanje = 0 then { 0 0 preskočimo; če sta dva zaporedna, končamo }
      begin ReadLn(T, Zac, Trajanje); if Zac + Trajanje = 0 then break end;
    { Časi koncev bodo dobili negativni predznak, da jih bomo lahko
      ločili od časov začetkov. Predpostavili bomo, da se noben
      sestanek ne začne ob negativnem času ali pa konča ob času 0. }
    { Vstavimo čas začetka in konca v urejeno zaporedje. }
```

[96] Glej *The On-Line Encyclopedia of Integer Sequences*, A000041.

```
      Dodaj(Zac); Dodaj(−(Zac + Trajanje));
    end; {while}
    Close(T);

    { V zaporedje dodajmo še „sestanek“ 0..MinCas. To nas bo prisililo,
      da sestanka ne bomo začeli prezgodaj. }
    if MinCas > 0 then begin Dodaj(0); Dodaj(−MinCas); Zac := Casi[1] end
    else Zac := MinCas;

    { Poiščimo primeren čas začetka sestanka. }
    i := 1; StSest := 0;
    while (i <= StCasov) and (Zac + Dolz <= MaxCas) do begin
      if (StSest <= 0) and (Zac + Dolz <= Abs(Casi[i])) then break;
      if Casi[i] < 0 then StSest := StSest − 1 else StSest := StSest + 1;
      Zac := Abs(Casi[i]); i := i + 1;
    end; {while}

    { Izpišimo rezultat. }
    Assign(T, 'sestanki.out'); Rewrite(T);
    if Zac + Dolz <= MaxCas then WriteLn(T, Zac)
    else WriteLn(T, 'SESTANEK NI MOZEN');
    Close(T);
end. {Sestanki1}
```

Še ena različica te rešitve je, da uredimo sestanke po začetnih časih, nato pa se sprehajamo po zaporedju in hranimo v neki spremenljivki najzgodnejši naslednji čas (recimo S), ko ni nobenega sestanka (na začetku postavimo S na f). Ob sestanku (s_{ij}, d_{ij}) je mogoče, da je $S + d \leq s_{ij}$ in v tem primeru lahko novi sestanek začnemo ob času S; sicer pa bo treba novi sestanek začeti po času $s_{ij} + d_{ij}$, zato S povečamo do toliko, če je trenutno manjši. Če pride S čez vrednost $t - d$, pa vemo, da sestanek ni mogoč. Lepo pri tej drugi rešitvi je, da moramo pri k sestankih urejati le k elementov, ne pa $2k$ (oz. $2k + 2$) kot zgoraj. Postopek urejanja z vstavljanjem, ki ga uporabljamo tu za urejanje časov, je preprost, vendar precej neučinkovit, saj za urejanje n elementov porabi $O(n^2)$ časa. To pomeni, da porabi zgornji program, ki mora urejati dvakrat več elementov kot spodnji, za to približno štirikrat več časa, ta razlika pa se pošteno pozna tudi pri celotnem času izvajanja, saj porabita oba programa večino svojega časa ravno za urejanje (drugi deli postopka imajo linearno časovno zahtevnost). Pri kakšnem učinkovitejšem postopku urejanja bi bila ta razlika najbrž manjša.

```
program Sestanki2;
const MaxSest = 10000;
var T: text; Zacetki, Konci: array [1..MaxSest] of longint; i, StSest: integer;
    MinCas, MaxCas, Dolz, Zac, Trajanje: longint;
begin
  { Preberimo vhodne podatke. }
```

```
  Assign(T, 'sestanki.in');
  Reset(T); ReadLn(T, MinCas, MaxCas, Dolz); StSest := 0;
  while true do begin
    ReadLn(T, Zac, Trajanje);
    if Zac + Trajanje = 0 then { 0 0 preskočimo; če sta dva zaporedna, končamo }
      begin ReadLn(T, Zac, Trajanje); if Zac + Trajanje = 0 then break end;
    { Zaporedje sestankov naj bo urejeno naraščajoče po začetnem času. }
    i := StSest; while i >= 1 do
      if Zacetki[i] <= Zac then break
      else begin Zacetki[i+1] := Zacetki[i]; Konci[i+1] := Konci[i]; i := i-1 end;
    Zacetki[i + 1] := Zac; Konci[i + 1] := Zac + Trajanje;
    StSest := StSest + 1;
  end; {while}
  Close(T);
  { Poiščimo primeren čas začetka sestanka. }
  Zac := MinCas; i := 1;
  while (i <= StSest) and (Zac + Dolz <= MaxCas) do begin
    if Zac + Dolz <= Zacetki[i] then break;
    if Konci[i] > Zac then Zac := Konci[i];
    i := i + 1;
  end; {while}
  { Izpišimo rezultat. }
  Assign(T, 'sestanki.out'); Rewrite(T);
  if Zac + Dolz <= MaxCas then WriteLn(T, Zac)
  else WriteLn(T, 'SESTANEK NI MOZEN');
  Close(T);
end. {Sestanki2}
```

R2002.3.5 Produkt števil

Ničle se v podzaporedje gotovo ne splača vzeti, če hočemo imeti velik produkt (razen če so vsi drugi produkti, ki jih lahko dobimo, negativni). Torej v mislih razrežimo zaporedje pri ničlah in se posebej posvetimo vsakemu od nastalih kosov (zanje torej predpostavimo, da ne vsebujejo ničel); za vsak kos poiščemo njegovo najboljše podzaporedje in na koncu izpišemo tisto, ki nam da največji produkt od vseh. N: 495

Ker imamo opravka s celimi števili, so po absolutni vrednosti vsa večja ali enaka 1, torej absolutni vrednosti našega produkta ne morejo škodovati — če jih vzamemo več, se bo absolutna vrednost lahko le povečala ali ostala enaka. Paziti moramo le še na predznak. Če je v opazovanem kosu zaporedja sodo mnogo negativnih števil, lahko vzamemo kar celoten kos, saj bo dal pozitiven produkt z največjo možno absolutno vrednostjo. Drugače pa se moramo odpovedati enemu od negativnih števil, torej vzeti ali čim več pred takim številom

ali pa čim več za njim. Očitno bomo imeli največje možnosti za velik produkt, če bomo vzeli vse pred zadnjim negativnim številom ali pa vse za prvim. Če je kos sestavljen iz enega samega števila in je le-to negativno, nam ne ostane drugega, kot da vzamemo to in upamo, da bo v kakšnem drugem kosu produkt boljši (pravzaprav bo boljša že katera od ničel med kosi, če imamo več kosov).

Ker lahko postanejo zmnožki, s katerimi moramo tu delati, zelo veliki, si je koristno pomagati z naslednjo predstavitvijo števil: število $\pm 2^k$ predstavimo s parom $(\pm 1, k)$. Produkt dveh takih števil, recimo (s_1, k_1) in (s_2, k_2), lahko potem predstavimo s parom $(s_1 s_2, k_1 + k_2)$. Ničlo lahko predstavimo s parom $(0, 0)$. Števili (s_1, k_1) in (s_2, k_2) lahko tudi preprosto primerjamo — najprej po prvi komponenti (ker je vsako pozitivno število večje od vsakega negativnega), če pa sta po tej enaki, primerjamo še k_1 in k_2 (če sta pozitivni, je večje tisto z večjim eksponentom, sicer pa tisto z manjšim).

Spodnji program računa v (s1, k1) produkt vsega, kar se v trenutnem kosu pojavlja pred zadnjim doslej najdenim negativnim številom; v (s2, k2) produkt vsega za prvim negativnim številom v kosu; in v (s3, k3) vse od vključno zadnjega negativnega števila naprej (če v trenutnem kosu še ni negativnih števil, je to produkt celega kosa). Na koncu zaporedja (pri i = N + 1) si mislimo še eno ničlo, da nam zaključi trenutni kos.

```
program ProduktStevil;
var sb, kb, bOd, bDo: longint; { najboljši doslej znani zmnožek }

  procedure Kandidat(s, k, iOd, iDo: longint);
  begin
    if (iDo >= iOd) and ((s > sb) or ((s = sb) and (s * k > sb * kb))) then
      begin sb := s; kb := k; bOd := iOd; bDo := iDo end;
  end; {Kandidat}

var T: text; i, N, a, KosOd, PrvaNeg, ZadnjaNeg, s, k, s1, k1, s2, k2, s3, k3: longint;
begin
  { Preberimo vhodne podatke. }
  Assign(T, 'produkt.in');
  Reset(T); ReadLn(T, N); KosOd := 0;
  for i := 1 to N + 1 do begin
    { Preberimo naslednje število, a. Na koncu zaporedja si
      mislimo še eno ničlo, da nam konča prejšnjo skupino števil. }
    if i <= N then ReadLn(T, a) else a := 0;
    { Izrazimo a v obliki s · 2^k.
      k je torej pravzaprav dvojiški logaritem vrednosti |a| (če a ni 0).
      Pomagali si bomo z operatorjem shl: „x shl y“ je vrednost x,
      zamaknjena za y bitov v levo. }
    if a < 0 then begin s := -1; a := -a end
    else if a > 0 then s := 1 else s := 0;
    k := 0; while a > longint(1) shl k do k := k + 1;
```

```
    { Začetni kandidat za najboljši zmnožek naj bo kar prvo število zaporedja. }
    if i = 1 then begin sb := s; kb := k; bOd := i; bDo := i end;

    { s1 · 2^k1 = produkt v trenutnem kosu pred zadnjim negativnim številom.
      s2 · 2^k2 = produkt v trenutnem kosu po prvem negativnem številu.
      s3 · 2^k3 = produkt v trenutnem kosu od vklj. zadnjega negativnega števila. }
    if KosOd = 0 then begin { inicializacija na začetku kosa }
      s1 := 1; k1 := 0; s2 := 1; k2 := 0; s3 := 1; k3 := 0;
      PrvaNeg := 0; ZadnjaNeg := 0; KosOd := i;
    end; {if}

    if s < 0 then begin { negativno število }
      if PrvaNeg = 0 then PrvaNeg := i
      else begin s2 := s2 * s; k2 := k2 + k end;
      s1 := s1 * s3; k1 := k1 + k3; s3 := s; k3 := k; ZadnjaNeg := i;
    end else if s > 0 then begin { pozitivno število }
      if PrvaNeg > 0 then begin s2 := s2 * s; k2 := k2 + k end;
      s3 := s3 * s; k3 := k3 + k;
    end else begin { konec kosa }
      Kandidat(s1, k1, KosOd, ZadnjaNeg - 1); { vse pred zadnjim neg. številom }
      Kandidat(s2, k2, PrvaNeg + 1, i - 1);   { vse po prvem neg. številu }
      Kandidat(s1 * s3, k1 + k3, KosOd, i - 1); { cel kos }
      if i <= N then Kandidat(s, k, i, i);      { ničla }
      KosOd := 0; { začenja se nov kos }
    end; {if}
  end; {for i}

  { Izpišimo rezultat. }
  Assign(T, 'produkt.out'); Rewrite(T); WriteLn(T, bOd, ' ', bDo); Close(T);
end. {ProduktStevil}
```

R2002.3.6 Prüferjev kod

Drevo je mogoče predstaviti na različne načine, katerega izbrati, pa je odvisno od tega, kakšne operacije bomo izvajali nad njim. Nas bo zanimalo vedeti, kdaj smo vozlišču zbrisali že toliko sosedov, da je to vozlišče postalo list; ko postane list, nas zanima, kateri je tisti edini preostali sosed; in ko nato list zbrišemo, moramo v njegovem sosedu evidentirati, da ima ta zdaj enega soseda manj. Zanimalo nas bo tudi, kateri izmed trenutnih listov ima največjo oznako. N: 496

Zato je koristna naslednja podatkovna struktura: za vsako vozlišče (struktura VozlisceT v spodnjem programu) hranimo stopnjo (Stopnja) (število sosedov) in kazalec na seznam njegovih sosedov (Sosedje). Ta seznam naj bo dvojno povezan (Prejsnja in Naslednja v strukturi PovezavaT), da bo enostavneje zbrisati iz njega poljuben element. Če je vozlišče a sosed vozlišča b, je tudi b sosed vozlišča a in zato za vsako povezavo dobimo dve strukturi PovezavaT, eno v seznamu a-jevih sosedov in eno v seznamu b-jevih sosedov. Koristno

je, če ti dve strukturi kažeta druga na drugo (Nasprotna); tako lahko za vozlišče a z enim samim sosedom b hitro pridemo do strukture, ki predstavlja a-ja v seznamu b-jevih sosedov (tudi to strukturo moramo namreč pobrisati, ko pobrišemo povezavo med a in b).

Vzdrževati moramo tudi množico trenutnih listov, da bomo med njimi lahko izbrali najmanjšega. Spodnji program si pomaga z navadno dvojiško kopico, ki jo hrani v tabeli Listi in z njo dela prek podprogramov ZadnjiList in DodajList. Lahko bi uporabili tudi rdeče-črno drevo ali kaj podobnega. Pri majhnih drevesih, kakršna so tule, bi lahko imeli najbrž tudi navaden seznam listov in ga vsakič v celoti prečesali, da bi videli, kateri je največji.

```
program PruferjevKod;
const MaxN = 10000;
var Listi: array [1..MaxN] of integer; StListov: integer;

  function ZadnjiList: integer;
  var i, ci, x: integer;
  begin
    ZadnjiList := Listi[1]; x := Listi[StListov]; StListov := StListov - 1; i := 1;
    while 2 * i <= StListov do begin
      ci := 2 * i;
      if ci + 1 <= StListov then if Listi[ci + 1] > Listi[ci] then ci := ci + 1;
      if Listi[ci] <= x then break;
      Listi[i] := Listi[ci]; i := ci;
    end; {while}
    Listi[i] := x;
  end; {ZadnjiList}

  procedure DodajList(x: integer);
  var i, pi: integer;
  begin
    StListov := StListov + 1; i := StListov;
    while i > 1 do begin
      pi := i div 2; if Listi[pi] >= x then break;
      Listi[i] := Listi[pi]; i := pi;
    end; {while}
    Listi[i] := x;
  end; {DodajList}

type
  VozlisceT = record Stopnja, Sosedje: integer end;
  PovezavaT = record Sosed, Prejsnja, Naslednja, Nasprotna: integer end;
var
  T: text; N, i, u, w: integer;
  V: array [1..MaxN] of VozlisceT;
  E: array [1..2 * (MaxN - 1)] of PovezavaT;
begin
```

```
  { Preberimo število vozlišč. }
  Assign(T, 'produkt.in');
  Reset(T); ReadLn(T, N);
  for u := 1 to N do begin V[u].Stopnja := 0; V[u].Sosedje := 0 end;

  { Preberimo povezave. }
  for i := 1 to N − 1 do begin
    ReadLn(T, u, w);
    { w postane u-jev sosed. }
    with E[2 * i − 1] do begin Sosed := w; Prejsnja := 0;
                               Naslednja := V[u].Sosedje; Nasprotna := 2 * i end;
    if V[u].Sosedje > 0 then E[V[u].Sosedje].Prejsnja := 2 * i − 1;
    V[u].Sosedje := 2 * i − 1; V[u].Stopnja := V[u].Stopnja + 1;
    { u postane w-jev sosed. }
    with E[2 * i] do begin Sosed := u; Prejsnja := 0;
                           Naslednja := V[w].Sosedje; Nasprotna := 2 * i − 1 end;
    if V[w].Sosedje > 0 then E[V[w].Sosedje].Prejsnja := 2 * i;
    V[w].Sosedje := 2 * i; V[w].Stopnja := V[w].Stopnja + 1;
  end; {for i}

  { Dodajmo liste v kopico. }
  Close(T); StListov := 0;
  for u := 1 to N do if V[u].Stopnja = 1 then DodajList(u);

  { Izpišimo rezultat. }
  Assign(T, 'produkt.out');
  Rewrite(T);
  while N > 2 do begin
    u := ZadnjiList; N := N − 1;
    w := E[V[u].Sosedje].Sosed; WriteLn(T, w);

    { Zbrišimo zapis i, ki pravi, da je u sosed vozlišča w. }
    i := E[V[u].Sosedje].Nasprotna;
    if E[i].Prejsnja > 0 then E[E[i].Prejsnja].Naslednja := E[i].Naslednja
    else V[w].Sosedje := E[i].Naslednja;
    if E[i].Naslednja > 0 then E[E[i].Naslednja].Prejsnja := E[i].Prejsnja;

    { Če je w postal list, ga dodajmo v kopico. }
    V[w].Stopnja := V[w].Stopnja − 1;
    if V[w].Stopnja = 1 then DodajList(w);
  end; {while}
  Close(T);
end. {PruferjevKod}
```

Razmislimo še o algoritmu za dekodiranje, torej za rekonstrukcijo drevesa iz kodnega zaporedja. Vsako vozlišče s k sosedi se pojavlja v kodnem zaporedju točno $(k-1)$-krat (vsakič, ko eden od njegovih sosedov postane list in ga odtrgamo; ko ostane le en sosed, je naše vozlišče že samo postalo list in takrat ga ne bomo več izpisovali). Torej lahko iz števila pojavitev vozlišč v zaporedju ugo-

tovimo, kakšna je bila stopnja (število sosedov) vsakega vozlišča v prvotnem drevesu. Vemo tudi, da je število vseh vozlišč za 2 večje od dolžine kodnega zaporedja. Zdaj torej ni težko ugotoviti, kateri je bil v prvotnem drevesu list z največjo oznako, iz prvega števila v kodnem zaporedju pa zdaj tudi vemo, kdo je bil takrat njegov sosed. Potem lahko v mislih ta list vzamemo iz drevesa, stopnjo soseda zmanjšamo in zdaj s pomočjo drugega števila v zaporedju ugotovimo, kdo je bil drugi zbrisani list in na koga je bil povezan.

Za hranjenje listov in ugotavljanje, kateri ima trenutno najvišjo oznako, lahko uporabimo prav takšno kopico kot zgornji program za kodiranje. Spodnji postopek izpiše za vsako povezavo drevesa po eno vrstico z dvema številoma, ki predstavljata krajišči te povezave. Na koncu, ko smo obdelali že vseh $n-2$ členov kodnega zaporedja, nam ostaneta od drevesa le še dva lista, ki sta morala biti torej tudi povezana s povezavo, tako da na koncu izpišemo še to.

Algoritem *Refürp:*
Vhod: kodno zaporedje $\langle a_1, \ldots, a_{n-2} \rangle$
Recimo, da so oznake vozlišč $1, \ldots, n$.

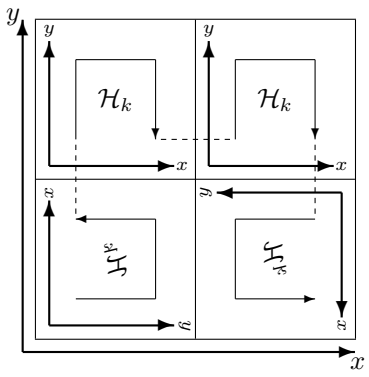
```
for u := 1 to n do Stopnja[u] := 1;
for i := 1 to n − 2 do Stopnja[a[i]] := Stopnja[a[i]] + 1;
for u := 1 to n do if Stopnja[[u]] = 1 then DodajList(u);
for i := 1 to n − 2 do
    u := ZadnjiList;
    WriteLn(u, ' ', a[i]);
    Stopnja[a[i]] := Stopnja[a[i]] - 1;
    if Stopnja[a[i]] = 0 then DodajList(u);
u := ZadnjiList; v := ZadnjiList;
WriteLn(u, ' ', v);
```

Pred tekmovanjem smo razmišljali o tem, da bi v nalogi opisali postopek za kodiranje, od tekmovalca pa zahtevali, naj napiše program za dekodiranje, potem pa smo to opustili, češ da bi bilo pretežko in smo v nalogi raje zahtevali izvedbo programa za kodiranje. Tule pa je pravzaprav videti, da bi bil program za dekodiranje znatno preprostejši (odpade zapletena podatkovna struktura za predstavitev drevesa); no, res pa bi bila naloga težja v tem smislu, da bi se morali tekmovalci postopka za dekodiranje šele domisliti, ker jim ne bi bil že podan.

R2002.3.7 Hilbertova krivulja

N: 497 Pomagamo si lahko z dejstvom, da je krivulja $\mathcal{H}_{k+1}$ sestavljena iz štirih kopij krivulje $\mathcal{H}_k$ in zato najprej obišče vse točke v spodnji levi četrtini svoje mreže, nato vse v zgornji levi, nato v zgornji desni in končno vse v spodnji desni. Za pretvorbo koordinat v številko točke moramo izračunati njeno številko znotraj

ustrezne kopije krivulje $\mathcal{H}_k$ in prišteti število točk v tistih delih mreže, ki jih $\mathcal{H}_{k+1}$ obišče še prej.[97]



Upoštevati moramo, da ima vsaka kopija krivulje $\mathcal{H}_k$ svoj posebni koordinatni sistem, v katerem je izhodišče drugje kot pri koordinatnem sistemu krivulje $\mathcal{H}_{k+1}$, pa tudi osi sta lahko obrnjeni drugače.

Spodnji program zna tudi pretvarjati številke točk v koordinate (če kot a dobi negativno število, si b razlaga kot številko točke in izpiše njene koordinate). Tu moramo ugotoviti, v kateri četrtini mreže leži točka (vsaka četrtina mreže $2^k \times 2^k$ ima 4^{k-1} točk), nato pa jo lahko pretvorimo v koordinate ustrezne kopije krivulje $\mathcal{H}_k$.

```
program Hilbert;

  function Kodiraj(k, x, y: longint): longint;
  var h, j: longint;
  begin
    if k = 0 then begin Kodiraj := 0; exit end;
    h := 1 shl (k - 1); j := h * h;
    if (x < h) and (y < h) then Kodiraj := Kodiraj(k - 1, y, x)
    else if (x < h) then Kodiraj := j + Kodiraj(k - 1, x, y - h)
    else if (y < h) then Kodiraj := 3*j + Kodiraj(k-1, h-1-y, 2*h-1-x)
    else Kodiraj := 2 * j + Kodiraj(k - 1, x - h, y - h);
  end; {Kodiraj}

  procedure Dekodiraj(k, i: longint; var x, y: longint);
  var h, j, xx, yy: longint;
  begin
    if k = 0 then begin x := 0; y := 0; exit end;
    h := 1 shl (k - 1); j := h * h; Dekodiraj(k - 1, i mod j, xx, yy);
    if i < j then begin x := yy; y := xx end
    else if i < 2 * j then begin x := xx; y := yy + h end
    else if i < 3 * j then begin x := xx + h; y := yy + h end
    else begin x := 2 * h - 1 - yy; y := h - 1 - xx end;
  end; {Dekodiraj}

var T: text; k, a, b: longint;
```

[97] Mimogrede, Hilbertovo krivuljo lahko smiselno posplošimo tudi na tri ali več dimenzij in sestavimo postopek, ki bo znal za poljuben d pretvarjati med indeksi in koordinatami na d-razsežnih Hilbertovih krivuljah. (T. Bially: *Space-filling curves: Their generation and their application to bandwidth reduction*, IEEE Trans. on Inf. Theory, IT-15(6):658–664, Nov. 1969; W. Gilbert: *A cube-filling Hilbert curve*, Math. Intelligencer 6(3):78, 1984, `http://www.math.uwaterloo.ca/~wgilbert/Research/HilbertCurve/HilbertCurve.html`).

```
begin
  Assign(T, 'produkt.in'); Reset(T); ReadLn(T, k, a, b); Close(T);
  Assign(T, 'produkt.out'); Rewrite(T);
  if a >= 0 then WriteLn(T, Kodiraj(k, a, b))
  else begin Dekodiraj(k, b, a, b); WriteLn(T, a, ' ', b) end;
  Close(T);
end. {Hilbert}
```

R2002.3.8 Slovar

N: 498

Slovar lahko predstavimo z neusmerjenim grafom, v katerem vsaki besedi ustreza ena točka, dve točki pa sta povezani, če se njuni besedi razlikujeta le v eni črki. Problem, ki ga rešujemo, se tako prevede v problem iskanja najkrajših poti (najkrajših po številu povezav) in ga lahko rešujemo na primer z iskanjem v širino.

Za gradnjo grafa imamo več možnosti: (1) Naivni pristop, ki primerja vsak par besed in preveri, če se razlikujeta le v enem mestu — če je besed veliko, bo trajalo to predolgo časa. (2) Za vsako dolžino besed, npr. d, in za vsak i od 1 do d, vzamemo vse besede dolžine d in jih uredimo, pri čemer se ob primerjanju besed delamo, kot da i-te črke ni. Tako pridejo skupaj tiste, ki se razlikujejo le v eni črki. (3) Za vsako dolžino besed d in za vsak i od 1 do d pripravimo razpršeno tabelo, v katero dodamo vse besede dolžine d, le da se pri tem delamo, kot da jim manjka i-ta črka. Tako spet lahko opazimo, če se kakšne besede razlikujejo le v eni črki.

```
program Slovar;
const
  MinDolz = 2; MaxDolz = 10;
  MaxN = 10000; MaxStopnja = 30;
  MaxSkupDolz = 55000;
  RazpMax = 10037; { primerno veliko praštevilo }
type
  SosedeT = packed array [1..MaxStopnja * MaxN] of integer;
  SosedeP = ↑SosedeT;
var
  T, TT: text; S, S1, S2: string;
  i, i2, j, k, d, dNasl, iOd, iDo, Izp, L, N, hc, Prehod, StPrimerov: integer;
  { Črke vseh besed, zbite skupaj v eno dolgo tabelo;
    besedi i pripadajo Crke[PrvaCrka[i]..PrvaCrka[i] + Dolzina[i] − 1]. }
  Crke: packed array [1..MaxSkupDolz + MaxDolz] of char;
  PrvaCrka, Dolzina: array [0..MaxN] of word;
  { Sosede[d] kaže na tabelo s seznami sosed za vse besede dolžine d;
    teh sosed je vsega skupaj SkupajSosed[d]. Sosede besede i so v celicah
    Sosede[Dolzine[i]]↑[PrvaSoseda[i]..PrvaSoseda[i] + StSosed[i] − 1]. }
  Sosede: array [MinDolz..MaxDolz] of SosedeP;
```

```
  SkupajSosed: array [MinDolz..MaxDolz] of word;
  StSosed, PrvaSoseda: array [0..MaxN] of word;
  { Razpršena tabela (za lažje odkrivanje enakih besed). Reze[hc]
    kaže na prvo besedo z razpršilno kodo hc; če je to beseda i,
    kaže potem Verige[i] na naslednjo, Verige[Verige[i]] na še
    naslednjo in tako naprej. Konec verige označuje indeks 0. }
  Reze: array [0..RazpMax - 1] of integer;
  Verige: array [1..MaxN] of integer;
  { Vrsta za iskanje v širino. Trenutna beseda je na oddaljenosti
    d od začetne besede, tiste, ki so v vrsti od indeksa dNasl naprej,
    pa so že na oddaljenosti d + 1. }
  Vrsta, Obiskana: array [1..MaxN] of integer; Glava, Rep: integer;

  { Izračuna razpršilno kodo dane besede (z eno izpuščeno črko).
    Takšno kodo se lahko uporabi kot indeks v tabelo Reze. }
  function Razprsi(StBesede, IzpuscenaCrka: integer): integer;
  var r: longint; i: integer;
  begin
    r := 0; for i := 1 to Dolzina[StBesede] do if i <> IzpuscenaCrka then
      r := (r * 256 + Ord(Crke[PrvaCrka[StBesede] + i - 1])) mod RazpMax;
    Razprsi := r;
  end; {Razprsi}

  function Enaki(StBesede1, StBesede2, IzpuscenaCrka: integer): boolean;
  var i: integer;
  begin
    Enaki := false; if Dolzina[StBesede1] <> Dolzina[StBesede2] then exit;
    for i := 1 to Dolzina[StBesede1] do if i <> IzpuscenaCrka then
      if Crke[PrvaCrka[StBesede1] + i - 1] <> Crke[PrvaCrka[StBesede2] + i - 1]
      then exit;
    Enaki := true;
  end; {Enaki}

begin
  Assign(T, 'slovar.in');
  Reset(T); ReadLn(T, N);
  { Preberimo slovar. }
  for i := 1 to N do begin
    ReadLn(T, S); L := Length(S); Dolzina[i] := L;
    if i = 1 then PrvaCrka[i] := 1
    else PrvaCrka[i] := PrvaCrka[i - 1] + Dolzina[i - 1];
    for j := 1 to L do Crke[PrvaCrka[i] + j - 1] := S[j];
  end; {for i}
  { Za vsako besedo pripravimo seznam sosed. To naredimo v dveh
    prehodih; najprej bomo sosede samo prešteli, v drugem prehodu
```

```
    pa jih bomo zapisali v tabele Sosede. }
  for Prehod := 1 to 2 do begin
    for i := 1 to N do StSosed[i] := 0;
    for d := MinDolz to MaxDolz do for Izp := 1 to d do begin

      { Začnimo s prazno razpršeno tabelo. }
      for i := 0 to RazpMax - 1 do Reze[i] := 0;
      for i := 1 to N do Verige[i] := 0;

      { Preglejmo vse besede dolžine d. }
      for i := 1 to N do if Dolzina[i] = d then begin

        { V razpršeni tabeli poglejmo, katere se ujemajo z i-to
          v vseh črkah razen na mestu Izp. }
        hc := Razprsi(i, Izp); i2 := Reze[hc];
        while i2 <> 0 do begin
          if Enaki(i, i2, Izp) then begin
            { i2 je res soseda besede 1. }
            if Prehod = 2 then begin { dodajmo ju v seznam sosed }
              Sosede[d]↑[PrvaSoseda[i] + StSosed[i]] := i2;
              Sosede[d]↑[PrvaSoseda[i2] + StSosed[i2]] := i;
            end; {if Prehod = 2}
            StSosed[i] := StSosed[i] + 1; StSosed[i2] := StSosed[i2] + 1;
          end; {if Enaki}
          i2 := Verige[i2]; { naprej po seznamu besed z razpršilno kodo hc }
        end; {while}

        { Dodajmo še besedo i v razpršeno tabelo. }
        Verige[i] := Reze[hc]; Reze[hc] := i;
      end; {for i}
    end; {for d, Izp}

    if Prehod = 1 then begin
      for d := MinDolz to MaxDolz do SkupajSosed[d] := 0;
      for i := 1 to N do begin
        PrvaSoseda[i] := SkupajSosed[Dolzina[i]] + 1;
        SkupajSosed[Dolzina[i]] := SkupajSosed[Dolzina[i]] + StSosed[i];
      end; {for i}
      for d := MinDolz to MaxDolz do if SkupajSosed[d] > 0 then
        GetMem(Sosede[d], SkupajSosed[d] * SizeOf(integer));
    end; {if Prehod = 1}

  end; {for Prehod}

  { Pomečimo zdaj vse besede v razpršeno tabelo,
    da jih bomo pri odgovarjanju na poizvedbe lažje našli. }
  for i := 0 to RazpMax - 1 do Reze[i] := 0;
  for i := 1 to N do begin Verige[i] := 0; Obiskana[i] := 0 end;
  for i := 1 to N do
    begin hc := Razprsi(i, 0); Verige[i] := Reze[hc]; Reze[hc] := i end;
```

```
  { Začnimo iskati najkrajše poti med danimi pari besed.
    Iskane besede bomo pomožno obravnavali pod številko 0. }
  PrvaCrka[0] := PrvaCrka[N] + Dolzina[N];
  Assign(TT, 'slovar.out');
  Rewrite(TT); ReadLn(T, StPrimerov);

  for j := 1 to StPrimerov do begin
    ReadLn(T, S); i := 1; while S[i] <> ' ' do i := i + 1;
    S1 := Copy(S, 1, i - 1); S2 := Copy(S, i + 1, Length(S) - i);

    { Niza S1 in S2 poiščimo v razpršeni tabeli. }
    Dolzina[0] := Length(S1);
    for i := 1 to Length(S1) do Crke[PrvaCrka[0] + i - 1] := S1[i];
    iOd := Reze[Razprsi(0, 0)]; while not Enaki(0, iOd, 0) do iOd := Verige[iOd];
    Dolzina[0] := Length(S2);
    for i := 1 to Length(S2) do Crke[PrvaCrka[0] + i - 1] := S2[i];
    iDo := Reze[Razprsi(0, 0)]; while not Enaki(0, iDo, 0) do iDo := Verige[iDo];

    { Z iskanjem v širino iščemo pot od besede iOd do besede iDo. }
    Obiskana[iOd] := j; Glava := 1; Rep := 2;
    Vrsta[Glava] := iOd; d := 0; dNasl := 2;
    while (Glava < Rep) and (Obiskana[iDo] < j) do begin
      if Glava = dNasl then begin d := d + 1; dNasl := Rep end;
      i := Vrsta[Glava]; Glava := Glava + 1;
      { Dodajmo v vrsto i-jeve sosede, če jih nismo že kdaj prej. }
      for k := 1 to StSosed[i] do begin
        i2 := Sosede[Dolzina[i]]↑[PrvaSoseda[i] + k - 1];
        if Obiskana[i2] = j then continue; { Točka i2 je bila že obiskana. }
        Vrsta[Rep] := i2; Rep := Rep + 1; Obiskana[i2] := j;
      end; {for k}
    end; {while}

    { Zdaj smo ali našli pot do iDo ali pa preiskali vse, do česar se je dalo
      iz iOd sploh priti (in ugotovili, da se do iDo ne da priti). }
    if Obiskana[iDo] = j then WriteLn(TT, d + 1)
    else WriteLn(TT, 'Ni prehoda');
  end; {for j}

  Close(T); Close(TT);
  for d := MinDolz to MaxDolz do if SkupajSosed[d] > 0 then
    FreeMem(Sosede[d], SkupajSosed[d] * SizeOf(integer));
end. {Slovar}
```

Gornji program si prizadeva biti varčen pri porabi pomnilnika in ima zato podatkovne strukture mogoče urejene malo bolj zapleteno, kot bi bilo nujno potrebno. V resnici bi bilo gotovo čisto sprejemljivo tudi, če bi pri razpršeni tabeli in seznamih sosedov uporabili dinamično alocirane zapise in jih povezovali s kazalci.

Viri nalog za leto 2002: limuzine — Matija Grabnar; najdaljši cikel, prijatelji in sovražniki, število vsot, Prüferjev kod — Jure Leskovec; TiVo — Mark Martinec; bencin, uvrstitve tekmovalcev — Marjan Šterk; razbijanje kode, razbijanje gesel — Miha Vuk; pristanišče, slovar — Anže Žagar; kodiranje, sestanki — Klemen Žagar; produkt števil — po zgledu ACM SEERC 1996, naloga C, #787 na `online-judge.uva.es`; Hilbertova krivulja — Janez Brank. TiVo je nek proizvajalec osebnih videorekorderjev (*www.tivo.com*). Zahvale ljudem, ki so implementirali rešitve tujih nalog za 3. skupino: Blažu Fortuni za produkt števil; Blažu Novaku za Prüferja; Marjanu Šterku za Hilberta.

27. državno tekmovanje v znanju računalništva (2003)

NALOGE ZA PRVO SKUPINO

2003.1.1 Dva kupa števil

Napiši podprogram R: 542

```
procedure Razdeli(N: integer);
```

ali, v C-ju:

```
void Razdeli(int N);
```

ki bo števila $1, 2, 3, \ldots, N-1, N$ (kjer je N vhodni parameter podprograma Razdeli) razdelil na dva kupa in to tako, da bosta vsoti števil na enem in na drugem kupu čim bolj podobni; če pa je možnih glede tega več enako dobrih razporedov, poišči tistega, ki ima na obeh kupih čim bolj enako število števil.

Podprogram naj izpiše vsebino vsakega od kupov in vsoto števil na njem. Možnih je več rešitev. Dovolj je, da najdeš eno izmed njih.

Primer: Razdeli(9) lahko izpiše

```
1. kup: 2, 5, 6, 9  Vsota: 22
2. kup: 1, 3, 4, 7, 8  Vsota: 23
```

2003.1.2 „Pet čevljev merim, palcev pet,“

... je tarnal mlad trgovec, ko je čez lužo odprl trgovino z blagom. Sprva se mu niti sanjalo ni, da mu utegne pretvarjanje enot povzročati toliko preglavic. R: 545

„Kako že gre? *1 liga* so *3 milje*, *1 milja* je *8 furlongov*, *1 furlong* je *220 jardov*, *1 jard* so *3 čevlji*, *1 čevelj* so *3 dlani*, *1 dlan* so *4 palci*,“ je drdral v mislih. Uh, saj to je še šlo, a ko je nekaj kupcev naročilo blago, se je pri seštevanju in naročanju blaga v tovarni krepko uštel.

Pomagaj mu **sestaviti program**, ki s standardnega vhoda prebere deset vrstic s po sedmimi števili (ki pomenijo zaporedoma število lig, milj, furlongov, jardov, čevljev, dlani in palcev blaga, ki so jih posamezne stranke naročile), nato pa še vrstico sedmih števil, ki povedo količino blaga, naročenega v tovarni. Na standardni izhod naj nato izpiše eno od naslednjih sporočil:

```
Naročil si *** preveč blaga.
Naročil si *** premalo blaga.
Naročil si ravno prav blaga.
```

Namesto zvezdic naj se izpiše količina preveč ali premalo naročenega blaga. Ta naj bo izražena tako, da uporabiš čim večje enote. Dolžino 5 000 jardov bi moral na primer napisati kot 2 milji, 6 furlongov in 160 jardov (ne pa kot 5 000 jardov ali pa kot 2 milji in 1 320 jardov ali pa kot 22 furlongov in 160 jardov ali pa celo kot 15 000 čevljev ali kaj podobnega). Količino zapiši kar s sedmimi števili, ločenimi s presledki, torej v enaki obliki, v kakršni so zapisane posamezne količine tudi v vhodnih podatkih. Dolžino 5 000 jardov bi tako zapisal kot `0 2 6 160 0 0 0`.

2003.1.3 Glasovanje

R: 547 Na šolah vsako leto v vsakem razredu učenci izvolijo predsednika razreda. Šola, na katero hodi tudi Miha, je to leto uvedla nov, bolj zaupen način volitev. Vsak učenec je na list napisal številko kandidata, za katerega je glasoval, nato pa je Miha števila iz vseh volilnih lističev pretipkal v računalnik. Sedaj pa potrebujejo tebe, da jim **sestaviš podprogram**, ki bo na podlagi teh števil izrisal histogram, iz katerega bo lepo razvidno, koliko glasov je dobil posamezen kandidat. Kandidati so oštevilčeni s številkami od 1 do StKandidatov, zagotovo pa je StKandidatov manjše ali enako MaxStKandidatov.

Tvoj podprogram naj bo takšne oblike:

```
const MaxStVolilcev = ...; MaxStKandidatov = ...;
type GlasoviT = array [1..MaxStVolilcev] of integer;

procedure Histogram(StKandidatov, StVolilcev: integer; Glasovi: GlasoviT);
begin
  ...
end;
```

oziroma, v C-ju:

```
#define MaxStKandidatov ...

void Histogram(int StKandidatov, int StVolilcev, int Glasovi[])
{
  ...
}
```

Primer: za 7 kandidatov, 10 volilcev in glasove $\langle 1, 3, 2, 4, 1, 4, 7, 6, 1, 2 \rangle$ naj podprogram izpiše:

```
1:***
2:**
3:*
4:**
```

```
5:
6:*
7:*
```

2003.1.4 Radar

Na cestnem odseku je postavljen radar, ki stalno beleži hitrosti mimovozečih vozil in jih zapisuje v datoteko, vsako meritev v svojo vrstico. Nabralo se je veliko število meritev, sedaj pa nas zanima dvajset najvišjih izmerjenih hitrosti. R: 548

Napiši program, ki prebere datoteko s podatki in izpiše dvajset največjih prebranih števil (ni nujno, da so v izpisu urejena po velikosti). Podatkov je preveč, da bi lahko vse shranili v pomnilnik.

NALOGE ZA DRUGO SKUPINO

2003.2.1 Križanka

Napiši podprogram, ki dobi kot vhod križanko, na izhod pa izpiše besede, ki se pojavljajo v križanki, skupaj z njihovimi položaji. R: 551

Križanka je sestavljena iz m vrstic in n stolpcev. V vsakem polju je ali velika tiskana črka angleške abecede ali pa znak `'*'`, ki pomeni, da to polje razmejuje besede. Za besedo šteje strnjeno zaporedje vsaj dveh črk, ki je napisano navpično ali vodoravno in je na začetku in koncu ločeno od ostalih črk v križanki z znakom `'*'` ali z robom križanke.

Vsa polja, na katerih se začnejo besede, oštevilčimo od 1 naprej. Štejemo po vrsticah od zgoraj navzdol, v vsaki vrstici pa od leve proti desni (glej primer).

Tvoj podprogram naj ustreza naslednjim deklaracijam:

```
const Visina = ...; Sirina = ...;
type KrizankaT = array [1..Visina, 1..Sirina] of char;
procedure IzpisiBesede(Krizanka: KrizankaT);
```

ali pa

```
#define Visina ...
#define Sirina ...
void IzpisiBesede(char Krizanka[Visina][Sirina]);
```

Primer: recimo, da imamo podano naslednjo križanko.

```
SOLAR
*LOK*
SIPON
IMATO
RPR*V
```

Najprej oštevilčimo vsa polja (od zgoraj navzdol, v vsaki vrstici od leve proti desni):

```
1234.
*5..*
6...7
8....
9....
```

Program mora v tem primeru izpisati:

```
1, vodoravno: SOLAR
1, navpično : -
2, vodoravno: -
2, navpično : OLIMP
3, vodoravno: -
3, navpično : LOPAR
4, vodoravno: -
4, navpično : AKOT
5, vodoravno: LOK
5, navpično : -
6, vodoravno: SIPON
6, navpično : SIR
7, vodoravno: -
7, navpično : NOV
8, vodoravno: IMATO
8, navpično : -
9, vodoravno: RPR
9, navpično : -
```

2003.2.2 Števke

R: 552

V danem naravnem številu je *zadnja neničelna števka* najbolj desna neničelna števka v desetiškem zapisu števila. Na primer, zadnja neničelna števka števila 123 je 3, pri številu 45600 je to 6, pri številu 100 pa 1.

Napiši podprogram

```
procedure Stevke(M, N: integer; var Rezultat: array [1..9] of integer);
```

ali, v C:

```
void Stevke(int M, int N, int Rezultat[9]);
```

ki za vsako števko od 1 do 9 ugotovi, kolikokrat se pojavi kot zadnja neničelna števka v zaporedju

$$M, M+1, M+2, \ldots, N-2, N-1, N.$$

Podprogram naj zapiše rezultat v tabelo Rezultat. Na primer, če je $M = 118$ in $N = 122$, pomeni, da obravnavamo zaporedje 118, 119, 120, 121, 122.

Zadnje neničelne števke so 8, 9, 2, 1 in 2. Torej mora tabela Rezultat vsebovati elemente 1, 2, 0, 0, 0, 0, 0, 1, 1, ker enica, osmica in devetka nastopajo kot zadnja neničelna števka enkrat, dvojka pa dvakrat.

Pozor: Razlika med številoma N in M je lahko tako velika, da je bolje, če ne obravnavamo vsakega števila med M in N posebej, ker bi podprogram predolgo tekel. Tvoj podprogram naj bo učinkovit!

2003.2.3 Različnost nizov

Pogosto je koristno definirati neko mero različnosti med nizi znakov. Takšne mere radi definirajo tako, da predpišejo nek nabor operacij, ki jih je nad nizi dovoljeno izvajati, vsaki operaciji pripišejo tudi neko ceno, nato pa definirajo razdaljo med dvema nizoma kot ceno najcenejšega zaporedja operacij, ki predela prvi niz v drugega. (Cena zaporedja operacij je definirana kar kot vsota cen vseh posameznih operacij v njem.) R: 554

Tako bomo storili tudi pri tej nalogi. Omejili se bomo na nize, ki jih sestavljajo same male črke angleške abecede. Nad njimi dovolimo tri operacije:

(1) Dodajanje znaka: poljubno črko vrinemo na poljubno mesto v nizu, lahko tudi na začetek ali na konec. Če hočemo na primer v *zxc* dodati *q*, lahko dobimo *qzxc*, *zqxc*, *zxqc* ali pa *zxcq*.

(2) Brisanje znaka: zbrišemo lahko poljuben znak niza. Iz *xyzzy* bi lahko na primer z enim brisanjem dobili *yzzy*, *xzzy*, *xyzy* ali pa *xyzz*.

(3) Premikanje znaka: en znak lahko premaknemo na poljubno drugo mesto v nizu; učinek je tak, kot da bi ga najprej zbrisali in nato dodali na neko drugo mesto v nizu. Iz *spqr* bi lahko s premikanjem znaka *s* dobili *psqr*, *pqsr* in *pqrs*.

Cena vsakega dodajanja in brisanja naj bo 1, cena premikanja pa 0. **Napiši funkcijo** Razdalja, ki za dana dva niza izračuna ceno najcenejšega zaporedja operacij, ki predela prvi niz v drugega. Funkcija naj bo takšne oblike:

```
function Razdalja(S, T: string): integer;
```

ali, v C-ju:

```
int Razdalja(const char *S, const char *T);
```

2003.2.4 Pošiljanje sporočil

R: 555 V računalniškem omrežju podjetja MiSmoSoft je veliko računalnikov. Sistemski inženir je dobil nalogo, naj čim hitreje pošlje sporočilo na vse računalnike v omrežju. Vsi računalniki v omrežju imajo sposobnost prejemanja in pošiljanja sporočil. Pomagaj sistemcu ter mu **opiši postopek**, ki z uporabo večjega števila računalnikov čim hitreje pošlje sporočilo do vseh računalnikov v omrežju. Pri tem upoštevaj, da pošiljanje sporočila iz računalnika popolnoma zaposli ta računalnik za 1 sekundo (pošiljanje N sporočil iz enega računalnika traja torej N sekund). Pošiljanje sporočil iz več računalnikov je popolnoma neodvisno in hkratno. Vsak računalnik v omrežju ima enolično določen naslov, ki je 32-bitna številka (naslov IP); vsak računalnik tudi hrani naslove vseh računalnikov v mreži. Trajanje procesiranja sporočil na računalniku lahko zanemarimo. Iščemo torej nakrajši čas, v katerem obvestimo vse računalnike v omrežju.

Začetek pošiljanja sporočil sproži sistemski inženir, ki pošlje sporočilo na en računalnik v omrežju (to označi v sporočilu). Tvoja naloga je napisati algoritem, ki čim hitreje pošlje sporočila do vseh računalnikov. Ta algoritem se bo v nespremenjeni obliki izvajal na vseh računalnikih v omrežju. Bodi pozoren na to, da se bo začel tvoj algoritem na posameznem računalniku izvajati takrat, ko ta računalnik prvič prejme kakšno sporočilo.

Zahtevani algoritem zapiši v telo funkcije, ki se pokliče ob prejemu sporočila. Bodi pozoren na temeljit opis algoritma in podatkov, ki jih pošiljaš. Računalnik velja za obveščenega, ko prejme kakršnokoli sporočilo. Prejemanje več sporočil je dovoljeno, vendar brezpomensko.

Na voljo imaš naslednje deklaracije in podprograme:

```
const StRacunalnikov = ...; { Število računalnikov v omrežju. }
type NaslovT = ...; { Naslov računalnika v omrežju (npr. IP-številka). }

type SporociloT = record
  { Pri sporočilu, ki ga bo poslal sistemski inženir prvemu
    računalniku, bo spodnja vrednost gotovo true. Pri ostalih
    sporočilih je pač taka, kakršno pripravi računalnik-pošiljatelj. }
  PrvoSporocilo: boolean;
  { Tu lahko dopolniš to strukturo še s svojimi polji. }
  ...
end;

{ Pošlje sporočilo S na računalnik z naslovom Prejemnik.
  Ta podprogram potrebuje za svojo izvršitev eno sekundo. }
procedure PosljiSporocilo(S: SporociloT; Prejemnik: NaslovT); external;

{ V tabelo Naslovi vpiše naslove vseh računalnikov v mreži
```

```
  (vključno z našim), v naraščajočem vrstnem redu. }
type NasloviT = array [1..StRacunalnikov] of NaslovT;
procedure NasloviVsehRacunalnikov(var Naslovi: NasloviT); external;

{ Vrne naslov našega računalnika. }
function NasNaslov: NaslovT; external;

{ Ko naš računalnik prejme kakšno sporočilo, bo sistem
  poklical naš spodnji podprogram. Kot parametra dobi sporočilo
  in pošiljateljev naslov. }
procedure ObPrejemuSporocila(S: SporociloT; Posiljatelj: NaslovT);
begin
  { Tu vpiši svoj postopek. }
  ...
end; {ObPrejemuSporocila}
```

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

[Na začetku tekmovanja smo tekmovalcem najprej razdelili naslednja navodila. Nekaj minut kasneje so dobili tudi besedilo nalog, za reševanje pa so imeli slabe tri ure časa. — *Op. ur.*]

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*`.in` in izpisujejo svoje rezultate v *imenaloge*`.out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil mapo s svojim uporabniškim imenom ter navodili, ki jih pravkar prebiraš. Ko boš sedel pred računalnik, boš dobil nadaljnja navodila za prijavo v sistem.

Na vsakem računalniku imaš na voljo enoto (disk) `U:`, na kateri lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku Pascal, C ali C++, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal in GNU C/C++. Za delo lahko uporabiš `turbo` (Turbo Pascal), `fp` oz. `ppc386` (FreePascal), `tc` (Turbo C), ali `gcc`/`g++` (GNU C/C++ — command line compiler). Ves potreben softver lahko najdeš na `c:\Programi` ter v meniju `Start` pod `Programs` in `Prevajalniki`.

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `rtk.exe`, ki ga lahko uporabiš za preverjanje svojih rešitev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti

svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

```
rtk imenaloge.pas
rtk imenaloge.c
rtk imenaloge.cpp
```

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Na spletni strani boš dobil obvestilo o tem, ali je program na testne primere odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Izjema je 1. naloga, kjer boste dobili vhodne datoteke že z nalogo, oddajati pa boste morali izhodne datoteke. Te se oddaja s klicem

```
rtk ImeIzhodneDatoteke
```

Imena datotek bodo podana v opisu naloge. Oddaja se vsako izhodno datoteko posebej in ni nujno, da oddaš datoteke za vse testne primere.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z drugimi datotekami kot z vhodno in izhodno. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov, prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi od 0 do 10 točk (praviloma 10, če je izpisal popolnoma pravilen odgovor, sicer pa 0; izjema je 2. naloga, ki dopuča tudi delno pravilne rešitve), nato pa se te točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N - 1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk.

Izjema pri opisanem načinu točkovanja je 1. naloga, pri kateri se oddaja po eno izhodno datoteko za vsak testni primer. Za vsako dobiš od 0 do 10 točk. Skupno število točk pri tej nalogi dobimo tako, da za vsak testni primer upoštevamo najboljšo od izhodnih datotek, ki jih je tekmovalec oddal za ta primer. Število oddaj pri tej nalogi ne zmanjšuje števila točk.

Skupno število točk tekmovalca je vsota po vseh nalogah. Tekmovalce razvrstimo po skupnem številu točk.

Vsak tekmovalec se mora sam zase odločiti o tem, katerim nalogam bo posvetil svoj čas, v kakšnem vrstnem redu jih bo reševal in podobno. Verjetno je priporočljivo najprej reševati lažje naloge.

Poskusna naloga (ne šteje k tekmovanju) `poskus.in`, `poskus.out`

Napiši program, ki iz vhodne datoteke prebere eno celo število (le-to je v prvi vrstici, okoli njega ni nobenih dodatnih presledkov ipd.) in izpiše njegov desetkratnik v izhodno datoteko.

Primer vhodne datoteke:	Ustrezna izhodna datoteka:
`123`	`1230`

Primer rešitve:

```
program PoskusnaNaloga;
var T: text; i: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * i); Close(T);
end. {PoskusnaNaloga}
```

```
#include <stdio.h>
int main() {
  FILE *f = fopen("poskus.in", "rt");
  int i; fscanf(f, "%d", &i); fclose(f);
  f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * i);
  fclose(f); return 0;
}
```

```
#include <fstream.h>
int main() {
  ifstream ifs("poskus.in"); int i; ifs >> i;
  ofstream ofs("poskus.out"); ofs << 10 * i;
  return 0;
}
```

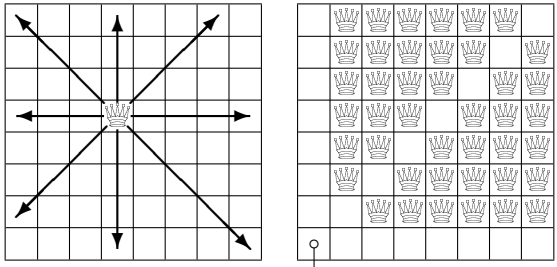
NALOGE ZA TRETJO SKUPINO

2003.3.1 Napadalne kraljice

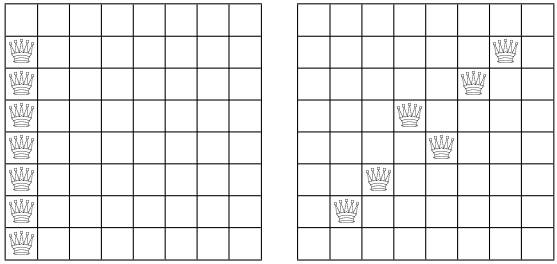
`dame01.out, dame02.out, ..., dame10.out`

Igra šah se igra na igralni plošči (šahovnici) z 8×8 polji. V igri nastopa več vrst figur, od katerih pa bo nas pri tej nalogi zanimala samo kraljica (dama). Kraljica se lahko premika po igralni plošči v osmih smereh (naravnost in po diagonalah) in sicer od svojega položaja pa vse do roba igralne plošče (glej zgornjo levo šahovnico na sliki, str. 536). Za polja, na katera bi se neka kraljica načeloma lahko premaknila, pravimo, da jih „napada“. Kraljica napada

R: 557



Kraljica in polja, ki jih napada.



Napadena so vsa polja.

tudi polje, na katerem že zdaj stoji. Če postavimo na šahovnico več kraljic, pravimo, da je neko polje napadeno, če ga napada vsaj ena od kraljic na šahovnici. Na šahovnici običajne velikosti, torej z 8×8 polji, lahko na primer s sedmimi ali celo s šestimi kraljicami že napademo vsa polja šahovnice — če te kraljice primerno razporedimo (glej spodnji dve šahovnici na sliki). Seveda tudi večje število kraljic samo po sebi še ne zagotavlja, da bodo napadena vsa polja (zgornja desna šahovnica na sliki).

Ta problem lahko še malo posplošimo, če se namesto običajne šahovnice z 8×8 polji zanimamo tudi za šahovnice drugih velikosti. Zanimalo nas bo naslednje: če imamo šahovnico velikosti $n \times n$ in bi radi nanjo postavili k kraljic, kako naj jih postavimo, da bo napadenih čim več polj?

Pri tej nalogi ne boš oddajal izvorne kode programa, pač pa boš dobil deset vhodnih datotek, za vsako od njih pa moraš oddati po eno izhodno datoteko. Vsaka od *vhodnih datotek* vsebuje dve števili, naprej n in potem k, ločeni s po enim presledkom. V *izhodni datoteki* podaj nek razpored k kraljic na šahovnico velikosti $n \times n$, pri katerem je napadenih čim več polj. Prva vrstica izhodne datoteke naj vsebuje tri števila, ločena s po enim presledkom: najprej n, nato k in nato še skupno število napadenih polj pri razporedu kraljic. Sledi naj še n vrstic, ki predstavljajo razpored kraljic na šahovnici. Vsaka od teh vrstic naj ima n znakov `'.'` ali `'#'`. Pike predstavljajo prazna polja, znaki `'#'` pa polja, na katerih stoji kakšna kraljica. Znakov `'#'` mora biti točno k, torej ni dovoljeno, da bi na kakšnem polju šahovnice stalo po več kraljic.

Primer vhodne datoteke:

```
11 5
```

Primer pripadajoče izhodne datoteke:

```
11 5 117
#..........
...........
...........
.......#...
...#........
...........
...........
........#..
....#......
...........
...........
```

Pozor: najti je mogoče tudi boljše rešitve. Z drugimi besedami, pet kraljic lahko na šahovnico 11×11 razporedimo tudi tako, da je napadenih več kot 117 polj.

Tule pa je deset vhodnih primerov, za katere moraš poiskati rešitve:

n	k	Oddaj datoteko s tem imenom
8	5	`dame01.out`
10	5	`dame02.out`
12	5	`dame03.out`
14	7	`dame04.out`
15	7	`dame05.out`
17	9	`dame06.out`
19	9	`dame07.out`
19	10	`dame08.out`
20	11	`dame09.out`
21	10	`dame10.out`

Posamezno rešitev oddaš tako, da pokličeš program `rtk` in mu podaš kot parameter ime izhodne datoteke:

`rtk dame`XX`.out`

kjer je XX eden od nizov `01`, `02`, ..., `10`.

Točkovanje. Za vsak testni primer, pri katerem si oddal kakšno izhodno datoteko, lahko dobiš od 0 do 10 točk. Če oblika izhodne datoteke ni takšna, kot je predpisano v nalogi (ali pa se ne ujema z vhodnimi podatki za ta testni primer), dobiš 0 točk. Sicer pa je število točk odvisno od tega, kako dobra je tvoja rešitev v primerjavi z najboljšo rešitvijo, ki jo je našla tekmovalna komisija. Naj bo t število napadenih polj v tvoji rešitvi, m pa v najboljši rešitvi, ki jo je uspela najti komisija.

Če je...	... dobiš toliko točk:
$t \geq m$	10
$t = m - 1$	9
$t = m - 2$	8
$m - 5 \leq t < m - 2$	7
$m - 10 \leq t < m - 5$	5
$m - 20 \leq t < m - 10$	4
$t < m - 20$	3

Za vsak testni primer lahko oddaš tudi več različnih izhodnih datotek; v tem primeru se šteje najboljša med njimi. Število oddaj pa sámo po sebi ne vpliva na to, koliko točk dobiš.

2003.3.2 Smučarji

`smucarji.in`, `smucarji.out`

R: 563 Na posamezni smučarski tekmi za svetovni pokal dobi vsak tekmovalec določeno število točk (0 ali več). Število prejetih točk je odvisno od njegove uvrstitve na tej tekmi. Mednarodna smučarska zveza vodi tudi razvrstitev v skupnem seštevku, kjer je vsak tekmovalec predstavljen z vsoto točk, ki jih je dobil na vseh tekmah trenutne sezone.

Ko manjka do konca sezone le še ena tekma, se opazovalci radi sprašujejo, katera je najslabša ali najboljša možna uvrstitev v skupnem seštevku, ki bi jo nek tekmovalec utegnil dobiti po zadnji tekmi. Recimo na primer, da se na posamezni tekmi dobi 100 točk za prvo mesto, 50 za drugo, 25 za tretje, za ostala pa še manj. Recimo še, da v skupnem seštevku trenutno vodi tekmovalec A, drugouvrščeni pa je tekmovalec B, ki za A-jem zaostaja za 60 točk. Iz tega že lahko sklepamo, da bo A po zadnji tekmi v skupnem seštevku ali prvi ali drugi, gotovo pa ne slabši. Da bi ga kdo v skupnem seštevku prehitel, bi namreč potreboval vsaj 60 točk, toliko pa lahko na eni tekmi dobi samo zmagovalec. Torej lahko A-ja v skupnem seštevku prehiti največ en tekmovalec (zmagovalec zadnje tekme, če se A na njej uvrsti dovolj slabo).

Tvoj program bo v vhodni datoteki dobil naslednje podatke. (Vsi podatki so cela števila, vsako je v samostojni vrstici, okoli njih ni presledkov, praznih vrstic ali česa podobnega.)

- V prvi vrstici je m, število smučarjev, ki na posamezni tekmi dobijo kaj točk. Tisti, ki se uvrstijo na $(m+1)$-vo ali slabše mesto, ne dobijo nič točk.

- V naslednjih m vrsticah je za vsako uvrstitev od prve do m-te podano število točk, ki jih dobi tekmovalec za to uvrstitev. Če število točk, ki jih prejme i-touvrščeni, označimo s t_i, lahko predpostaviš, da velja: $100\,000 \geq t_1 > t_2 > t_3 > \cdots > t_{m-1} > t_m > 0$.

- V naslednji vrstici je n, število smučarjev v skupnem seštevku svetovnega pokala.

- V naslednjih n vrsticah je podano za vsakega od teh smučarjev njegovo število točk v skupnem seštevku pred zadnjo tekmo sezone; najprej za vodilnega v skupnem seštevku, nato za drugega, itd., nazadnje pa za zadnjega. Če je a_i število točk, ki jih ima i-ti najboljši v skupnem seštevku, lahko predpostaviš, da velja: $100\,000 \geq a_1 \geq a_2 \geq a_3 \cdots \geq a_{n-1} \geq a_n \geq 0$.

Tako m kot n sta pozitivni celi števili, ki nista manjši od 1 in nista večji od 3 000.

V izhodno datoteko naj tvoj program zapiše n vrstic. V vsaki naj bosta dve pozitivni celi števili. Prvo število v i-ti vrstici naj bo najboljši možni položaj, ki ga bi utegnil imeti v skupnem seštevku po zadnji tekmi tisti tekmovalec, ki je v skupnem seštevku pred zadnjo tekmo na i-tem mestu. Drugo število v i-ti vrstici naj bo najslabši možni položaj, ki bi ga utegnil imeti ta tekmovalec v skupnem seštevku po zadnji tekmi.

Dogovorimo se še, da na zadnji tekmi sezone velja posebno pravilo: ne more se zgoditi, da bi si dva ali več tekmovalcev delilo isto mesto (torej tudi ne morejo dobiti enakega števila točk). Če bi imelo več tekmovalcev na stotinko enak čas, jih bodo pač razvrstili s pomočjo žreba in zakulisne kuhinje. (Lahko pa si več ljudi deli mesto v skupnem seštevku. Če sta dva sedma, je tisti takoj za njima deveti in podobno.) Mogoče pa je, da kak tekmovalec (lahko celo zelo veliko tekmovalcev) med tekmo odstopi (ali pa sploh ne štartajo ali pa so diskvalificirani) in takšni ne dobijo na tej tekmi nobenih točk.

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:
6	1 3
50	1 3
25	1 4
20	2 8
10	3 10
5	3 10
3	4 10
10	4 10
1000	4 10
980	4 10
971	
930	
925	
923	
920	
915	
912	
910	

Če pravilno izračunaš le najboljše možne uvrstitve, pri najslabših pa je kakšna napaka, dobiš pri tistem testnem primeru le tri točke; če pravilno izračunaš najslabše, ne pa najboljših, dobiš pri tistem testnem primeru sedem točk; če je pravilno oboje, dobiš pri tistem testnem primeru vseh deset točk.

2003.3.3 Vplivi

`vplivi.in`, `vplivi.out`

R: 566

Kajti če si je kdajkoli kdo zaslužil našo grmado, si to ti. Jutri te bom sežgal. Dixi.
Veliki inkvizitor v *Bratih Karamazovih*

Véliki inkvizitor vodi skupino n inkvizitorjev (v tem številu je zajet tudi sam). Ker vseh procesov proti krivovercem ne more neposredno nadzirati sam, obenem pa ne bi rad, da bi kak heretik ušel grmadi, bi rad občasno poslal do posameznega inkvizitorja kak nasvet, s katerim bi lahko vplival na njegovo razsodbo. Nerodno pa je, da je pripravljen vsak inkvizitor upoštevati nasvete le nekaterih drugih (veliki inkvizitor je lahko med njimi ali pa tudi ne), pa še pri tem ne vplivajo nanj vsi enako močno. Zato ga zanima, kako močan vpliv ima lahko na vsakega izmed preostalih inkvizitorjev, če si pametno izbere obveščevalne verige.

Oštevilčimo inkvizitorje s števili $1, 2, \ldots, n$. Veliki inkvizitor seveda dobi številko 1. Vpliv inkvizitorja i na inkvizitorja j označimo s $p(i, j)$. To je celo število, večje ali enako 0. (Vplivi niso nujno obojestranski: $p(i, j)$ ni nujno

enak $p(j,i)$.) Če vpliva inkvizitor i_0 na inkvizitorja i_1, ta na i_2, ta na i_3, ..., i_{k-1} pa vpliva na i_k, pravimo temu *pot* od i_0 do i_k in definiramo vpliv te poti kot $\min\{p(i_0,i_1), p(i_1,i_2), \ldots, p(i_{k-1},i_k)\}$. *Najvplivnejša pot* od i do j je taka pot od i do j, ki ima vsaj tolikšen vpliv kot vsaka druga pot od i do j.

Zagotovljeno je, da od velikega inkvizitorja do vsakega drugega obstaja vsaj ena pot z vplivom, večjim od 0. Veliki inkvizitor te prosi, da mu za vsakega inkvizitorja i ($i = 2, \ldots, n$) izračunaš vpliv najvplivnejše poti od 1 do i. (Mogoče sicer obstaja več različnih najvplivnejših poti od 1 do i, ampak že iz definicije je očitno, da imajo vse enak vpliv.)

Vhodna datoteka: v prvi vrstici je n (število vseh inkvizitorjev, vključno z velikim inkvizitorjem), v drugi neko pozitivno celo število m, sledi pa m vrstic, ki navajajo vse neničelne vplive. V vsaki od teh vrstic so tri števila: i, j in $p(i,j)$, ločena s po enim presledkom. Za vse te $p(i,j)$ v datoteki vedno velja: $1 \le p(i,j) \le 1\,000\,000\,000$. Če pa za nek par inkvizitorjev (i,j) v datoteki ni ustrezne vrstice, velja, da je $p(i,j) = 0$. Zagotovljeno je tudi, da je $p(i,i) = 0$ za vsak i. Predpostaviš lahko, da velja $1 \le n \le 1\,000$ in $1 \le m \le 200\,000$.

Izhodna datoteka naj ima $n-1$ vrstic. V prvi vrstici naj bo vpliv najvplivnejše poti od 1 do 2, v drugi vpliv najvplivnejše poti od 1 do 3 in tako naprej. V zadnji vrstici naj bo torej vpliv najvplivnejše poti od 1 do n.

Primer vhodne datoteke:

```
5
7
1 2 10
1 3 5
2 4 8
3 5 7
4 3 9
2 5 2
5 4 12
```

Pripadajoča izhodna datoteka:

```
10
8
8
7
```

2003.3.4 Vodenje projektov

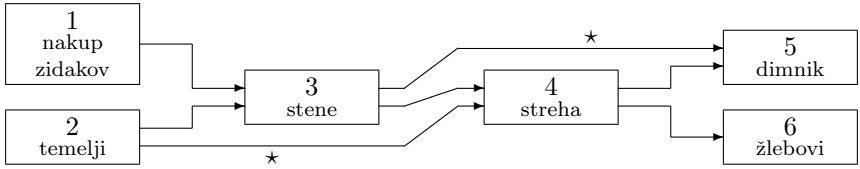
`projekti.in`, `projekti.out`

R: 572

Gradbeno podjetje Zidarstvo Polde, d. d., uporablja računalniško podprto načrtovanje projektov. Projekt opišejo kot množico aktivnosti, vsaka aktivnost pa je lahko odvisna od poljubno mnogo drugih aktivnosti, ki morajo biti končane, preden se lahko začne. Direktor Polde si zaradi lažje predstave rad nariše sliko celotnega projekta (glej primer), vendar njegovi podrejeni pri navajanju odvisnosti med aktivnostmi radi pretiravajo in navajajo odvečne podatke (kot na primer odvisnosti $\star$ na spodnji sliki), ki naredijo sliko nepregledno.

Polde te prosi za pomoč pri odstranjevanju odvečnih podatkov. Odvečne odvisnosti so vse, ki že izhajajo iz drugih navedenih odvisnosti. V podatkih je

odvisnost med dvema aktivnostima navedena *največ enkrat*. Nobena aktivnost tudi nikoli ni posredno ali neposredno odvisna od same sebe.



Oblika *vhodnih podatkov* je: v prvi vrstici je število aktivnosti, n; v drugi vrstici je število odvisnosti, m; sledi m vrstic, ki vsebujejo po dve števili: prvo je številka neke aktivnosti, drugo pa številka neke aktivnosti, ki je odvisna od prve. Veljalo bo $1 \leq n \leq 1000$, $0 \leq m \leq 10000$.

Tvoja naloga je izločiti največje možno število parov ⟨*aktivnost, odvisna aktivnost*⟩ tako, da bo zahtevano zaporedje aktivnosti še vedno enako. (Z drugimi besedami, če je bila neka aktivnost prvotno odvisna (posredno ali neposredno) od neke druge aktivnosti, mora biti tudi po končanem izločanju parov še vedno vsaj posredno odvisna od nje.) Na zgornjem primeru sta to odvisnosti, označeni z zvezdico (⋆). **Izločene** odvisnosti izpiši v *izhodno datoteko* kot pare števil (najprej številka aktivnosti, nato številka odvisne aktivnosti). Vrstni red, v katerem izpišeš izločene odvisnosti, ni pomemben, ne smeš pa nobene odvisnosti navesti več kot enkrat.

Primer vhodne datoteke (zgornja slika):

```
6
7
1 3
2 3
3 4
2 4
4 5
3 5
4 6
```

Ena od možnih pripadajočih izhodnih datotek:

```
2 4
3 5
```

2003.3.5 Knjižnica

`knjige.in`, `knjige.out`

V neki knjižnici imajo ogromno starih knjig, ki so zanimive predvsem zaradi zgodovinske vrednosti. Ker imajo zanje na voljo le en regal, morajo med njimi narediti nek izbor tako, da jih čimveč razvrstijo na police, preostale pa prepustijo muzeju za kulturno dediščino. Knjige so različno debele, potrebno pa jih je razvrstiti v kronološkem vrstnem redu izdaje — od leve proti desni in potem naprej na naslednji polici... Tako smejo biti na prvi polici na primer

R: 576

le knjige, izdane v letih od 1900 do 1903, na drugi le tiste od 1903 do 1908 itd. (letnice so tu podane le kot primer!). Z drugimi besedami, vse knjige na eni polici morajo imeti zgodnejši datum izdaje kot knjige na naslednji polici.

Tvoja naloga je, da najdeš največje možno število knjig, ki jih še lahko spravijo v regal. Podano imaš število knjig n, število polic m in širino regala d (to je v bistvu širina vsake posamezne police) ter seznam debelin knjig, podanih v centimetrih. Ta seznam je že urejen v kronološkem vrstnem redu izdaje, tako da datumov izdaj ne potrebuješ.

Vhod: najprej prebereš trojico celih števil N, M in d, za katere bo zagotovo veljalo $1 \leq N \leq 1000$, $1 \leq M \leq 100$ ter $1 \leq d \leq 100$, nato pa še zaporedje N celih števil d_i (za katera velja $1 \leq d_i \leq 10$), ki predstavlja debeline knjig, podane v kronološkem vrstnem redu izdaje. Nobena knjiga ni debelejša od širine police: za vsak i velja $d_i \leq d$.

Izhod: izpiši največje število knjig iz danega zaporedja knjig, ki jih pod danimi pogoji še lahko spravijo v regal.

Primer vhodne datoteke:

```
10 3 5
1 4 2 1 3 4 1 2 2 1
```

Pravilni odgovor za to vhodno datoteko:

```
8
```

Primer takšne razporeditve osmih knjig na tri police je:

$$\underline{1}\ 4\ \underline{2\ 1} \quad | \quad \underline{3}\ 4\ \underline{1} \quad | \quad \underline{2\ 2\ 1}.$$

Podčrtane so debeline tistih knjig, ki smo jih res uvrstili na police; navpični črti določata meji med policami.

REŠITVE NALOG ZA PRVO SKUPINO

R2003.1.1 Dva kupa števil

N: 527

Možnih je več rešitev. Začnemo lahko z dvema praznima kupoma in nato pregledujemo števila od večjih proti manjšim ter vsako število odložimo na tisti kup, ki ima trenutno manjšo vsoto (če imata oba enako, pa na prvi kup). Spodnji podprogram naredi dva prehoda po vseh številih in v prvem izpisuje, kaj je odložil na prvi kup, v drugem prehodu pa, kaj je odložil na drugi kup.

```
procedure Razdeli(N: integer);
var i, Kup, Kam: integer; Vsota: array [1..2] of integer;
begin
  for Kup := 1 to 2 do begin
    Write(Kup, '. kup:');
    Vsota[1] := 0; Vsota[2] := 0;
    for i := N downto 1 do begin
```

```
      if Vsota[1] <= Vsota[2] then Kam := 1 else Kam := 2;
      Vsota[Kam] := Vsota[Kam] + i;
      if Kam = Kup then Write(' ', i);
    end; {for i}
    WriteLn('  Vsota: ', Vsota[Kup]);
  end; {for Kup}
end; {Razdeli}
```

Recimo, da imata v nekem trenutku oba kupa enako vsoto; naslednje število (recimo k) bomo torej odložili na prvi kup. Ta ima zdaj večjo vsoto kot drugi, zato bomo naslednje število, $k-1$, odložili na drugi kup. Prvi ima še vedno večjo vsoto, zato tudi $k-2$ odložimo na drugi kup. Zdaj ima večjo vsoto drugi in bomo naslednje število, $k-3$, odložili spet na prvi kup. Zdaj pa, ker je $k+(k-3) = (k-1)+(k-2)$, imata oba kupa spet enako vsoto.

Torej bomo po vsakih štirih pregledanih številih imeli na obeh kupih enako vsoto in tudi enako mnogo števil. Če je N večkratnik števila 4, bo veljalo to tudi na koncu, po pregledu vseh števil, iz česar vidimo, da je ta rešitev najboljša možna. Če N ni večkratnik števila 4, pa nam po pregledu vseh četveric števil (spomnimo se, da imamo takrat dva kupa z enako vsoto in enako mnogo števili) ostane še število 1 ali pa števili 2 in 1 ali pa števila 3, 2 in 1, odvisno od tega, kakšen ostanek ima N pri deljenju s 4. Če nam ostane le 1 ali pa 2 in 1, vidimo, da mora biti vsota vseh števil od 1 do N liha (ker če smo vsa dosedanja števila razdelili na dva kupa z enako vsoto, mora biti njihova skupna vsota soda, tu pa smo ji prišteli še 1 ali pa $2+1$, kar je liho), torej se jih sploh ne da razdeliti na dva kupa z enako vsoto; naš podprogram bi dobil na enem kupu za 1 večjo vsoto kot na drugem, kar je v tem primeru torej najboljša rešitev. Če pa nam na koncu ostanejo števila 3, 2 in 1, bi naš algoritem na koncu dobil na obeh kupih enako vsoto, pri čemer bi bilo na enem kupu eno število več kot na drugem; boljše rešitve ni, saj mora biti N lih (če smo doslej gledali po štiri števila skupaj in so nam na koncu ostala tri) in se torej ne da dobiti dveh kupov z enako mnogo števili.

Oglejmo si primer za $N = 13$. Če razdelimo števila na skupine po štiri, dobimo:

$$13\ \ 12\ \ 11\ \ 10 \qquad 9\ \ 8\ \ 7\ \ 6 \qquad 5\ \ 4\ \ 3\ \ 2 \qquad 1.$$

Prvi kup bi torej dobil števila $13, 10, 9, 6, 5, 2$ in na koncu še 1, drugi kup pa bi dobil $12, 11, 8, 7, 4$ in 3. Tako ima prvi kup sedem števil z vsoto 46, drugi pa šest števil z vsoto 45.

Takšno obravnavanje števil v skupinah po štiri lahko zapišemo tudi bolj eksplicitno:

```
procedure Razdeli2(N: integer);
var i, Vsota: integer;
begin
```

```
  Write('1. kup: ', N); Vsota := N; i := N − 4;
  while i >= 0 do begin
    Write(' ', i + 1); if i > 0 then Write(' ', i);
    Vsota := Vsota + i + (i + 1); i := i − 4;
  end; {while}
  WriteLn('  Vsota: ', Vsota);
  Write('2. kup:'); Vsota := 0; i := N − 2;
  while i >= 0 do begin
    Write(' ', i + 1); if i > 0 then Write(' ', i);
    Vsota := Vsota + i + (i + 1); i := i − 4;
  end; {while}
  WriteLn('  Vsota: ', Vsota);
end; {Razdeli2}
```

Do enako dobrih razbitij na dva kupa pa pridemo tudi z naslednjim razmislekom. Recimo, da bi šli po vrsti od 1 do N in dajali števila izmenično na prvi in na drugi kup. Pri $N = 13$ bi dobili:

prvi kup:	1	3	5	7	9	11	13
drugi kup:	2	4	6	8	10	12.	

Če drugo vrstico malo zamaknemo,

prvi kup:	1	3	5	7	9	11	13
drugi kup:		2	4	6	8	10	12,

vidimo, da dobi pri vsakem paru števil (za 1 si mislimo, da je v paru z 0) prvi kup za eno večje število kot drugi. Parov je sedem, torej ima prvi kup za sedem večjo vsoto kot drugi. Če zdaj v treh parih zamenjamo števili (tisto, ki je bilo prej v prvem kupu, pride v drugega in obratno), se vsota prvemu zmanjša za tri, drugemu pa poveča za tri, torej se vsoti zdaj razlikujeta le še za 1.

prvi kup:		2	4	7	9	11	13
drugi kup:	1	3	5	6	8	10	12,

Prvi kup ima zdaj vsoto 46, drugi pa 45.

Hitro se lahko prepričamo, da bi lahko podoben razmislek opravili tudi v primerih, ko je N sod (takrat nam druge vrstice ne bi bilo treba zamikati, bi pa imel zato drugi kup večjo vsoto kot prvi; vsoti bi spet zbližali s pomočjo zamenjav, na enak način kot prej).

Za izvedbo s programom je ta rešitev malo bolj nerodna, ker je preračunavanje, na kateri kup gre katero število, malo bolj zapleteno. Po tisti prvi delitvi (števila izmenično na en in na drugi kup) ostane $p := (N+1) \operatorname{div} 2$ parov, torej ima en kup za p večjo vsoto kot drugi. Torej bo dovolj, če izvedemo zamenjavo v prvih $p \operatorname{div} 2$ parih.

```
procedure Razdeli2(N: integer);
var P, Vsota, i, j, Kup: integer;
begin
  P := (N + 1) div 2;
  for Kup := 1 to 2 do begin
    Write(Kup, '. kup:'); Vsota := 0;
    for i := 1 to P do begin
      { Prvi kup naj bo tisti, ki bi imel brez zamenjav
        večjo vsoto. Od vsakega para bi torej ta kup dobil
        j, drugi pa j − 1. Če je N lih, se moramo delati,
        da začnemo s parom (0, 1), sicer pa s parom (1, 2). }
      j := 2 * i − (N mod 2);
      { Število, ki je v paru z j, je j − 1. Tega moramo
        uporabiti, če smo na drugem kupu ali pa če smo
        izvedli pri tem paru zamenjavo (ne pa, če velja oboje!). }
      if (Kup = 1) = (i <= P div 2) then j := j − 1;
      { Izpišimo število in ga dodajmo v vsoto. }
      Vsota := Vsota + j; if j > 0 then Write(' ', j);
    end; {for i}
    WriteLn(' Vsota: ', Vsota);
  end; {for Kup}
end; {Razdeli3}
```

R2003.1.2 „Pet čevljev merim, palcev pet“

Za lažje računanje z merskimi enotami (seštevanje in odštevanje) lahko vse količine najprej preračunamo v najmanjšo mersko enoto, torej v palce. Da ne bomo pisali vsake reči za vseh sedem enot posebej, jih kar oštevilčimo od 1 (liga) do 7 (palec). Vrednost Enota[i] nam bo povedala, kolikokrat je enota $i-1$ daljša od enote i. Ta preračun lahko opravimo od večjih enot proti manjšim: število lig pomnožimo s tri in prištejemo število milj; to pomnožimo z osem in prištejemo število furlongov; itd. Po vsakem takem koraku smo vse enote, večje od trenutne, pretvorili v trenutno, tako da na koncu dobimo prvotno količino izraženo v palcih. Potem ni težko seštevati in odštevati, na koncu pa moramo rezultat pretvoriti nazaj, da ga bomo lahko izpisali; to je obratna operacija od pretvorbe v palce. Kjer smo prej množili in prištevali, moramo zdaj deliti, računati ostanke in odštevati. Količino 5 000 jardov bi na primer delili z 220 (ker je 220 jardov en furlong) in ker je $5\,000 = 22 \cdot 220 + 160$, vemo, da je 5 000 jardov enako 22 furlongom in 160 jardom. Furlonge bi potem na enak način pretvorili v milje in furlonge in tako naprej.

N: 527

```
program MerskeEnote;

{ Enote so oštevilčene od 1 (lige) do 7 (palci).
  Enota[i] pove, kolikokrat je enota i − 1 daljša od enote i. }
```

```
const Enota: array [1..7] of integer= (1, 3, 8, 220, 3, 3, 4);

function Preberi: integer; { vrne prebrano dolžino v palcih }
var i, e, Dolzina: integer;
begin
  Dolzina := 0;
  for i := 1 to 7 do begin
    { Vrednost Dolzina je zdajle v enoti i − 1. Pretvorimo jo v enoto i. }
    Dolzina := Dolzina * Enota[i];
    { Preberimo količino enote i in jo prištejmo dolžini. }
    Read(e); Dolzina := Dolzina + e;
  end; {for}
  Preberi := Dolzina; ReadLn;
end; {Preberi}

{ Dolzina naj bo v palcih, Zapisi pa jo zapiše, kot zahteva naloga. }
procedure Zapisi(Dolzina: integer; S: string);
var i: integer; e: array [1..7] of integer;
begin
  for i := 7 downto 2 do begin
    { Vrednost Dolzina je zdajle v enoti i.
      Poglejmo, koliko se ne bo dalo izraziti z večjimi enotami. }
    e[i] := Dolzina mod Enota[i];
    { Preostanek pretvorimo v enoto i − 1. }
    Dolzina := Dolzina div Enota[i];
  end; {for}
  e[1] := Dolzina;
  for i := 1 to 7 do Write(e[i], ' ');
  WriteLn(S);
end; {Zapisi}

var Stranke, Tovarna, i: integer;
begin
  Stranke := 0; for i := 1 to 10 do Stranke := Stranke + Preberi;
  Tovarna := Preberi; Write('Naročil si ');
  if Stranke < Tovarna then Zapisi(Tovarna − Stranke, 'preveč blaga.')
  else if Stranke > Tovarna then Zapisi(Stranke − Tovarna, 'premalo blaga.')
  else WriteLn('ravno prav blaga.');
end. {MerskeEnote}
```

Druga možnost bi bila, da bi seštevali in odštevali kar v predstavitvi, razbiti na posamezne enote. Ta postopek bi bil tak kot pisno seštevanje ali odštevanje, le meja za prenos naprej je od mesta do mesta različna. Na primer, pri običajnem pisnem seštevanju pride do prenosa na naslednje mesto, če je bila vsota na prejšnjem večja ali enaka 10; tu pa mora priti do prenosa, če je bila vsota na prejšnjem mestu dovolj velika, da bi iz tega dobili že vsaj eno večjo enoto. Podobno bi razmišljali tudi pri odštevanju. Vendar pa bi imeli z vsem tem po

vsej verjetnosti več dela kot z gornjo rešitvijo.

R2003.1.3 Glasovanje

Pomagali si bomo s tabelo (StGlasov v spodnjem podprogramu), v kateri bomo za vsakega kandidata hranili število volilcev, ki so glasovali zanj. Na začetku postavimo vse elemente te tabele na 0, nato pa se sprehodimo po vseh glasovih in pri vsakem povečajmo števec pri tistem kandidatu, na katerega se ta glas nanaša. Na koncu ta števila glasov uporabimo, da vemo, koliko zvezdic izpisati pri posameznem kandidatu.

N: 528

```
program Glasovanje;

const MaxStVolilcev = 10; MaxStKandidatov = 10;
type TabelaT = array [1..MaxStVolilcev] of integer;

procedure Histogram(StKandidatov, StVolilcev: integer; Glasovi: TabelaT);
var
  StGlasov: array [1..MaxStKandidatov] of integer;
  i, j: integer;
begin
  for i := 1 to StKandidatov do StGlasov[i] := 0;
  for i := 1 to StVolilcev do
    StGlasov[Glasovi[i]] := StGlasov[Glasovi[i]] + 1;
  for i := 1 to StKandidatov do begin
    Write(i, ':');
    for j := 1 to StGlasov[i] do Write('*');
    WriteLn;
  end; {for}
end; {Histogram}

const Glasovi: TabelaT = (1, 3, 2, 4, 1, 4, 7, 6, 1, 2);
begin
  Histogram(7, 10, Glasovi);
end. {Glasovanje}
```

Še primer enovrstične rešitve v jeziku perl:

```
perl -ne '$k[$_]++; END { printf("%d:%s\n", $_, "*" x $k[$_]) for (1..$#k) }'
```

Stikalo `-ne` pove, da je program naveden kot naslednji parameter v ukazni vrstici (`e`) in da naj ga interpreter izvede po enkrat za vsako vrstico vhodne datoteke (`n`). Program predpostavi, da je v vsaki vrstici naveden en glas; vsebino trenutne vrstice dobimo v spremenljivki `$_` in jo uporabimo kot indeks v tabelo `k`, kjer ustreznemu elementu povečamo vrednost za 1. Preostanek programa, „`END { ... }`“ je podprogram po imenu `END`; če obstaja tak podprogram, ga interpreter pokliče na koncu, po tistem, ko je že obdelal celo vhodno

datoteko. Takrat v tabeli k že piše, koliko glasov je prejel posamezni kandidat, zato se lahko lotimo risanja histograma. Izraz $#k pomeni število elementov v tabeli k. „**for** (1..$#k)“ za stavkom, ki kliče printf, pomeni, da se bo ta stavek izvedel po enkrat za vsako število od 1 do $#k (trenutno število pa bomo videli v spremenljivki $_). Operator x pomeni ponavljanje niza: „"*" x $k[$_]“ je torej niz $k[$_] zvezdic, to pa je ravno toliko, kolikor glasov je dobil kandidat številka $_.

R2003.1.4 Radar

N: 529 Naloga pravi, da je meritev preveč, da bi lahko vse shranili v pomnilnik; če bi jih bilo manj, bi lahko vse prebrali v pomnilnik, jih uredili in tako ugotovili, katere so največje. (Pravzaprav bi bilo to časovno potratno, tudi če bi imeli dovolj pomnilnika za vsa števila.) Ker je meritev veliko, tudi ne bi bilo pametno iti dvajsetkrat skozi celotno datoteko (da bi npr. pri prvem prehodu poiskali največje število, pri drugem drugo največje in tako naprej).

Raje si med branjem vzdržujmo tabelo dvajsetih največjih izmed doslej prebranih števil. Ko preberemo novo število (recimo x), ga primerjamo s tistim, ki je bilo doslej dvajseto največje (recimo mu y); če je x večje, lahko y pozabimo in si namesto njega zapomnimo x.

Dvajset največjih doslej znanih števil lahko hranimo urejena po velikosti ali pa tudi ne. Vsaka od teh dveh različic ima svoje dobre in slabe strani. Če jih hranimo urejena po velikosti, bomo imeli vedno pri roki dvajseto največje doslej znano, vendar pa bo zato vstavljanje novega števila v tabelo malo zahtevnejše (ker ga bo treba včasih vriniti nekam na sredo tabele in ostala števila zato zamakniti za eno mesto). Če jih hranimo neurejena, je vstavljanje enostavno (preprosto vpišemo x v tisto celico, kjer je bil prej y), vendar pa bi morali načeloma vsakič prečesati celo tabelo dvajsetih števil, da bi ugotovili, katero je najmanjše med njimi. Temu se lahko izognemo, če si v neki spremenljivki zapomnimo, katero je najmanjše; ta podatek je treba potem popraviti le, ko vpišemo v tabelo novo število.

```
program RadarZUrejenoTabelo;
const N = 20;
var Tabela: array [1..N + 1] of real; i: integer;
    T: text; x: real;
begin
  Assign(T, 'podatki.txt'); Reset(T);
  for i := 1 to N do Tabela[i] := 0;
  while not Eof(T) do begin
    ReadLn(T, x);
    { Radi bi imeli manjše elemente na koncu tabele. Torej, kolikor je na koncu
      tabele elementov, ki so manjši od x, jih premaknimo za eno mesto naprej.
      Prav zato je tabela nalašč za eno celico daljša (ima N + 1 namesto N celic). }
```

```
      i := N;
      while i > 0 do
        if Tabela[i] >= x then break
        else begin Tabela[i + 1] := Tabela[i]; i := i − 1 end;
      { Zdaj vemo, da je Tabela[i] >= x, tako da moramo
        x postaviti eno mesto za njim. }
      Tabela[i + 1] := x;
   end; {while}
   Close(T);
   for i := 1 to N do WriteLn(Tabela[i]:0:2);
end. {RadarZUrejenoTabelo}

program RadarZNeurejenoTabelo;
const N = 20;
var Tabela: array [1..N] of real; i, KjeNajmanjsi: integer;
    T: text; x: real;
begin
   Assign(T, 'podatki.txt'); Reset(T);
   KjeNajmanjsi := 1; for i := 1 to N do Tabela[i] := 0;
   while not Eof(T) do begin
      ReadLn(T, x);
      if x > Tabela[KjeNajmanjsi] then begin
         Tabela[KjeNajmanjsi] := x;
         for i := 1 to N do
            if Tabela[i] < Tabela[KjeNajmanjsi] then KjeNajmanjsi := i;
      end; {if}
   end; {while}
   Close(T);
   for i := 1 to N do WriteLn(Tabela[i]:0:2);
end. {RadarZNeurejenoTabelo}
```

Če bi naloga zahtevala največjih n meritev za nek večji n, ne pa le $n = 20$, bi bilo koristno namesto urejene tabele uporabiti kakšno od podatkovnih struktur za prioritetno vrsto, npr. dvojiško kopico. To bi bilo podobno rešitvi z neurejeno tabelo, le da bi imeli v primerih, ko med n največjih pride neka nova meritev, le $O(\lg n)$ dela, ne pa $O(n)$.

V vsakem primeru je za opisane postopke najhujši tak scenarij, pri katerem pride ob vsaki novi meritvi do spremembe v množici n največjih meritev (na primer: če so meritve v vhodni datoteki že urejene naraščajoče). (Kajti v primerih, ko preberemo novo meritev, pa vidimo, da je manjša od n-te doslej največje, ni treba z njo narediti ničesar več, tako da imamo s tako meritvijo le $O(1)$ dela, neodvisno od n.) Če je v vhodni datoteki m meritev, imata prikazani rešitvi v takem najslabšem primeru časovno zahtevnost $O(mn)$, rešitev s kopico pa $O(m \lg n)$. Če bi si lahko privoščili prebrati vse meritve naenkrat v pomnilnik, bi jih lahko tam uredili in tako videli, katere so največje, vendar pa

bi urejanje zahtevalo $O(m \lg m)$ časa, kar torej ni nič bolje od rešitve s kopico (saj je m večji od n, verjetno celo precej večji); pač pa bi lahko (če bi imeli vse meritve v pomnilniku) uporabili znani postopek z mediano median, ki bi znal izbrati n-ti največji element v času $O(m)$, ne glede na n (glej npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 10.3 v prvi izdaji, 9.3 v drugi).

Na srečo pa, če so meritve naključno premešane, do sprememb v množici n največjih doslej prebranih meritev prihaja precej bolj poredko: čim več meritev smo že prebrali, tem manjša je verjetnost, da bo naslednja meritev prišla med n največjih; zato prihaja do sprememb med n največjimi vse bolj poredko in pri večini meritev bomo porabili le $O(1)$ časa za vsako meritev. Na primer: recimo, da so naše meritve naključne spremenljivke, porazdeljene neodvisno in enakomerno na intervalu $[0, 1]$. Recimo, da smo prebrali že m meritev (in $m \geq n$); naj bo $X_{(n)}$ n-ta največja med njimi. Potem za vsak $x \in [0, 1]$ velja

$$P(X_{(n)} < x) = \sum_{k=0}^{n-1} P(\text{točno } k \text{ meritev je} \geq x) = \sum_{k=0}^{n-1} \binom{m}{k} (1-x)^k x^{m-k}.$$

Označimo naslednjo prebrano meritev z X; ker je porazdeljena tako kot ostale, je njena gostota verjetnosti kar $f_X(x) = 1$ za $x \in [0, 1]$ in $f_X(x) = 0$ drugod. Zato je

$$\begin{aligned} P(X_{(n)} < X) &= \int_0^1 dx P(X_{(n)} < x) f_X(x) \\ &= \int_0^1 dx \sum_{k=0}^{n-1} \binom{m}{k} (1-x)^k x^{m-k} \\ &= \sum_{k=0}^{n-1} \binom{m}{k} \int_0^1 (1-x)^k x^{m-k} dx. \end{aligned}$$

Zadnji integral je poseben primer funkcije beta; pokazati je mogoče, da je enak $k!(m-k)!/(m+1)!$. Tako dobimo

$$\begin{aligned} P(X_{(n)} < X) &= \sum_{k=0}^{n-1} m!/(k!(m-k)!) \cdot k!(m-k)!/(m+1)! \\ &= \sum_{k=0}^{n-1} 1/(m+1) = n/(m+1). \end{aligned}$$

Ta verjetnost je torej res vse manjša, čim več meritev smo prebrali (čim večji je m). Pričakovano število sprememb med n največjimi števili je zato do časa, ko bomo prebrali M števil, enako $\sum_{m=n}^{M-1} n/(m+1)$. Pri tej vsoti si lahko pomagamo s harmoničnimi števili: $H_k = \sum_{i=1}^{k} 1/i$, za katera je znano, da je $H_k = \ln k + \gamma + 1/(2k) + O(k^{-2})$, konstanta γ pa je približno 0,577. Naša vsota je zato

$$\sum_{m=n}^{M-1} n/(m+1) = n \sum_{m=n+1}^{M} 1/m = n(H_M - H_{n+1}) \approx n \ln(M/n).$$

Torej lahko pričakujemo, da se pri branju M meritev spremembe med največjimi n meritvami zgodijo le v približno $n \ln(M/n)$ primerih. To pa ni le precej manjše od M, ampak tudi narašča precej počasneje.

V gornjem odstavku smo razmišljali o primeru, ko prihajajo meritve iz verjetnostne porazdelitve, porazdeljene enakomerno na $[0, 1]$. Vendar, če meritve

pretransformiramo s poljubno strogo naraščajočo funkcijo, je jasno, da do spremembe med največjimi n meritvami zdaj prihaja v natanko istih primerih kot prej (saj če je bila ena meritev npr. manjša od druge pred transformacijo, bo po njej tudi, če je uporabljena funkcija res strogo naraščajoča). Torej, če prihajajo naše meritve iz neke porazdelitve X in je verjetnostna funkcija te porazdelitve, torej $F_X(x) := P(X < x)$, strogo naraščajoča, lahko meritve poženemo skozi funkcijo F_X in dobimo vrednosti, porazdeljene enakomerno po intervalu $[0, 1]$: res, kajti $P(F_X(X) < y) = P(X < F_X^{-1}(y)) = F_X(F_X^{-1}(y)) = y$ (prvi enačaj velja, ker je F_X strogo naraščajoča, drugi velja po definiciji funkcije F_X, tretji pa zaradi definicije inverza). Zato lahko razmislek iz prejšnjega odstavka uporabimo tudi v tem primeru. Dobljena posplošitev pride prav, če so merive na primer porazdeljene normalno (Gaussova porazdelitev), kar je v praksi najbrž bližje resnici kot pa začetna predpostavka, da so porazdeljene enakomerno po nekem intervalu.

REŠITVE NALOG ZA DRUGO SKUPINO

R2003.2.1 Križanka

Križanko bomo pregledovali po vrsticah od zgoraj navzdol, vsako vrstico od leve proti desni, in pri vsakem polju preverili, če se tu začenja nova beseda. Vodoravna beseda se začenja, če na trenutnem polju in na tistem desno ob njem ni zvezdice, na tistem levo ob trenutnem pa je zvezdica. Podobno je za navpične besede, le da gledamo poleg trenutnega polja še tisto nad in tisto pod njim. Primere, ko je trenutno polje na robu križanke, bi morali obravnavati posebej; vodoravna beseda se lahko začne na levem robu križanke, ne pa na desnem (ker potem ne bi bila dolga vsaj dve črki), podobno pa velja tudi na navpične besede. Opazimo lahko, da je učinek tega pravila tak, kot da bi bila zunaj križanke tudi polja, na njih pa same zvezdice. To upošteva spodnji podprogram Zvezdica in nam s tem malo poenostavi preverjanje, ali se na trenutnem polju začne nova beseda. Kakorkoli že, če moramo potem neko besedo tudi res izpisati, se moramo le premikati od trenutnega polja v pravi smeri (desno za vodoravne besede, dol za navpične) in izpisovati črke, dokler ne naletimo na zvezdico (ali na rob križanke, vendar za to poskrbi že podprogram Zvezdica).

N: 529

```
const Visina = 5; Sirina = 10;
type KrizankaT = array [1..Visina, 1..Sirina] of char;

procedure IzpisiBesede(var Krizanka: KrizankaT);

  { Delali se bomo, kot da so zunaj križanke same zvezdice. }
  function Zvezdica(i, j: integer): boolean;
  begin
```

```
    if (i < 1) or (j < 1) or (i > Visina) or (j > Sirina)
    then Zvezdica := true else Zvezdica := Krizanka[i, j] = '*';
  end; {Zvezdica}

var i, j, k, Stevilka: integer; Vod, Nav: boolean;
begin
  Stevilka := 0;
  for i := 1 to Visina do for j := 1 to Sirina do
    if not Zvezdica(i, j) then begin

      { Preverimo, če se v tem polju začenja kakšna beseda. }
      Vod := Zvezdica(i, j - 1) and not Zvezdica(i, j + 1);
      Nav := Zvezdica(i - 1, j) and not Zvezdica(i + 1, j);
      if not (Vod or Nav) then continue;
      Stevilka := Stevilka + 1;

      { Izpišimo vodoravno besedo. }
      Write(Stevilka, ', vodoravno: '); k := j;
      if not Vod then Write('-')
      else while not Zvezdica(i, k) do
        begin Write(Krizanka[i, k]); k := k +1  end;

      { Izpišimo navpično besedo. }
      WriteLn; Write(Stevilka, ', navpično: '); k := i;
      if not Nav then Write('-')
      else while not Zvezdica(k, j) do
        begin Write(Krizanka[k, j]); k := k + 1 end;
      WriteLn;

    end; {if}
end; {IzpisiBesede}

const Krizanka: KrizankaT = (
  'LOREM*IP*S',
  'UM*DOL*ORS',
  'ITAMET*CON',
  'SETE*TUR*S',
  'ADI*PSCING');
begin
  IzpisiBesede(Krizanka);
end.
```

R2003.2.2 Števke

N: 530

Naivna rešitev bi bila, da bi preprosto našteli vsa števila od M do N, pri vsakem pogledali, katera je zadnja neničelna števka (to lahko naredimo tako, da ga delimo z deset, dokler ni njegov ostanek po deljenju z deset različen od 0; ta ostanek je ravno zadnja neničelna števka), ter povečali ustrezno celico

tabele Rezultat. Ta rešitev je spodaj v podprogramu Stevke2. Vendar pa, kot pravi že besedilo naloge, sta lahko M in N velika in bi takšna rešitev tekla predolgo.

Lahko pa bi razmišljali takole: če gledamo po deset zaporednih števil, $10k$, $10k+1, 10k+2, \ldots, 10k+9$, je za vsako števko od 1 do 9 tu natanko eno število, ki ima to števko kot zadnjo števko. Če je torej med M in N veliko takih deseteric števil, ni nobene potrebe, da bi jih vse naštevali posebej; lahko preprosto izračunamo, koliko jih je, in ustrezno povečamo vse celice tabele Rezultat. S števili oblike $10k$ pa je tako, da je njihova zadnja števka 0, tako da se jim zadnja *neničelna* števka nič ne spremeni, če vsa ta števila delimo z deset. Tako lahko torej namesto števil $10k, 10k+10, 10k+20, \ldots$ gledamo kar števila $k, k+1, k+2, \ldots$.

Spodnji podprogram Stevke najprej poveča M in zmanjša N do prvega večkratnika števila 10. Števila, ki smo jih zaradi tega preskočili, imajo vsa zadnjo števko različno od 0 in jih torej lahko upoštevamo tako, da povečamo ustrezno celico tabele Rezultat za 1.

Ko sta enkrat M in N večkratnika števila 10, vemo, da je med njima $(N-M)/10$ skupin po deset števil (to so števila od M do vključno $N-1$), za povrhu pa še število N. Kot smo ugotovili zgoraj, lahko izmed teh števil vsa, ki niso večkratniki 10, upoštevamo v rezultatih tako, da vse celice tabele Rezultat povečamo za $(N-M)/10$. Števila $M, M+10, M+20, \ldots, N$ pa lahko vsa delimo z 10 in jih obdelamo tako, da kličemo Rezultat s parametroma M div 10 in N div 10.

```
type RezultatT = array [1..9] of integer;

procedure Stevke(M, N: integer; var Rezultat: RezultatT);
var i, d: integer; R: RezultatT;
begin
  for i := 1 to 9 do Rezultat[i] := 0;
  while (M mod 10 <> 0) and (M <= N) do
    begin Rezultat[M mod 10] := Rezultat[M mod 10] + 1; M := M + 1 end;
  while (N mod 10 <> 0) and (M <= N) do
    begin Rezultat[N mod 10] := Rezultat[N mod 10] + 1; N := N - 1 end;
  { Zdaj sta M in N večkratnika 10. Naj bo d := (N - M)/10. Na intervalu
    M..N - 1 se torej pojavi d števil z zadnjo števko i, in to za vsak i od 0 do 9.
    Za i = 1, ..., 9 jih lahko torej kar prištejemo v Rezultat, za i = 0 pa imajo
    tista števila, pa tudi N sam, isto zadnjo neničelno števko, kot če bi jih vsa
    delili z 10, to pa nam da interval (M div 10)..(N div 10). }
  if M <= N then begin
    Stevke(M div 10, N div 10, R);
    for i := 1 to 9 do Rezultat[i] := Rezultat[i] + R[i];
    d := (N - M) div 10;
    for i := 1 to 9 do Rezultat[i] := Rezultat[i] + d;
  end; {if}
```

```
end; {Stevke}

procedure Stevke2(M, N: integer; var Rezultat: RezultatT);
var i, j: integer;
begin
  for i := 1 to 9 do Rezultat[i] := 0;
  for i := M to N do begin
    j := i; while j mod 10 = 0 do j := j div 10;
    Rezultat[j mod 10] := Rezultat[j mod 10] + 1;
  end; {for}
end; {Stevke2}

var M, N, i: integer; R: RezultatT;
begin
  ReadLn(M, N); Stevke(M, N, R);
  for i := 1 to 9 do if R[i] > 0 then Write(i, ':', R[i], ' ');
  WriteLn; Stevke2(M, N, R);
  for i := 1 to 9 do if R[i] > 0 then Write(i, ':', R[i], ' ');
  WriteLn; { Upajmo, da se rezultata ujemata. }
end.
```

R2003.2.3 Različnost nizov

N: 531 S premikanjem znakov, ki je zastonj, lahko poljubno spremenimo vrstni red znakov v nizu, ne moremo pa brisati odvečnih pojavitev neke črke ali dodajati primerkov črke, ki se je dotlej pojavljala v premalo izvodih. Seveda ne bi imelo smisla, če bi neko črko najprej dodali in kasneje zbrisali ali pa obratno. Najcenejše zaporedje operacij bo zato tisto, ki le doda manjkajoče ali zbriše odvečne črke in jih nato preuredi v pravi vrstni red.

Zato za vsako črko abecede poglejmo, kolikokrat se pojavlja v prvem in kolikokrat v drugem nizu. Če se pojavlja v prvem večkrat kot v drugem, bo treba nekaj pojavitev zbrisati, če v drugem večkrat kot v prvem, pa bo treba nekaj pojavitev dodati. V vsakem primeru je cena tega kar absolutna vrednost razlike števila pojavitev. Vsota teh cen nam zadostuje, da prvi niz predelamo v nekaj, kar ima enake črke kot drugi niz (in v enakem številu izvodov), nato pa jih moramo le še preurediti, kar pa je zastonj.

```
function Razdalja(S, T: string): integer;
var NS, NT: array ['a'..'z'] of integer; c: char; i: integer;
begin
  for c := 'a' to 'z' do begin NS[c] := 0; NT[c] := 0 end;
  for i := 1 to Length(S) do NS[S[i]] := NS[S[i]] + 1;
  for i := 1 to Length(T) do NT[T[i]] := NT[T[i]] + 1;
  i := 0; for c := 'a' to 'z' do i := i + Abs(NS[c] - NT[c]);
  Razdalja := i;
end; {Razdalja}
```

R2003.2.4 Pošiljanje sporočil

N: 532

Ker so naslovi (številke IP) enolični, lahko računalnike uredimo po naslovih v urejen seznam. Seznam nato predelamo tako, da je računalnik, ki je prvi prejel sporočilo (od uporabnika), tudi prvi v seznamu (računalnike z nižjim IP-jem dajmo na konec seznama). Nato ta računalnik pošlje sporočilo do naslednjega v seznamu. Nato 1. in 2. računalnik pošljeta sporočilo do 3. in 4. Vsi skupaj pošljejo sporočilo do 5., 6., 7. in 8. Vsi računalniki pošiljajo sporočila, dokler ne dosežejo zadnjega v seznamu.

Primer za 9 računalnikov (oštevilčeni z 0–8):

1. sekunda: $0 \to 1$
2. sekunda: $0 \to 2$, $1 \to 3$
3. sekunda: $0 \to 4$, $1 \to 5$, $2 \to 6$, $3 \to 7$
4. sekunda: $0 \to 8$

Potrebovali smo torej 4 sekunde. Pomembno je, da uporabljajo vsi računalniki enak seznam naslovov, česar pa ni težko zagotoviti, saj vsi poznajo vse naslove, s prej opisanim urejanjem pa lahko tudi vsak razporedi naslove v enak vrstni red.

Kako bi to delovalo v splošnem? Če je v neki sekundi m računalnikov poslalo sporočilo, bodo v naslednji sekundi poslali sporočilo vsi ti računalniki, pa tudi vsi tisti, ki so v prejšnji sekundi šele prejeli obvestilo: to pa je ravno tistih m prejemnikov, ki so jim pošiljatelji v prejšnji sekundi poslali sporočila. V naslednji sekundi bo torej pošiljalo sporočila $2m$ računalnikov. Ta razmislek nam pove, da v k-ti sekundi pošilja sporočila 2^{k-1} računalnikov. Če računalnike oštevilčimo z indeksi od 0 naprej, vidimo, da v k-ti sekundi pošiljajo računalniki $0, \ldots, 2^{k-1}-1$ sporočila računalnikom $2^{k-1}, \ldots, 2^k-1$, in sicer tako, da vsak računalnik i pošlje obvestilo računalniku $i+2^{k-1}$.

Ko torej naš računalnik prvič dobi obvestilo, mora vedeti le, v kateri sekundi pošiljanja se je to zgodilo, pa bo lahko ugotovil, koga mora začeti sam obveščati v naslednji sekundi. Dovolj je že, če računalnik ugotovi svoj indeks, kar lahko stori tako, da poišče svoj naslov v primerno urejeni tabeli naslovov. Ko najde svoj indeks i, lahko poišče tak k, za katerega je $2^{k-1} \le i < 2^k$; za tega potem velja, da je naš računalnik dobil obvestilo v k-ti sekundi. (Lahko pa bi ta k prenašali tudi skupaj s sporočilom. Kasneje, ko bi naš računalnik pošiljal sporočila drugim, bi k pred vsakim pošiljanjem povečal za 1.) Zdaj torej vemo, s kakšnim k-jem moramo nadaljevati, ko bomo sami pošiljali sporočila. Svoje prvo sporočilo bomo poslali v okviru $(k+1)$-ve sekunde, tako da ga bomo morali poslati računalniku $i+2^k$. Sekundo zatem bomo obvestili računalnik $i+2^{k+1}$ in tako naprej. Ko nam ti indeksi padejo čez število računalnikov, moramo seveda nehati; takrat vemo, da bodo najkasneje do konca trenutne sekunde obveščeni že vsi računalniki.

```
type SporociloT = record
                    PrvoSporocilo: boolean;
                    { Naslov prvega računalnika (tistega, ki je prvi dobil
                      sporočilo od uporabnika). Tega potrebujemo, da lahko
                      vsak računalnik sestavi enak vrstni red naslovov in to
                      takega, v katerem je prvi računalnik na začetku tabele. }
                    NaslovPrvega: NaslovT;
                  end; {SporociloT}

procedure ObPrejemuSporocila(S: SporociloT; Posiljatelj: NaslovT);

  procedure PrerazporediNaslove(var Naslovi: NasloviT; NaslovPrvega: NaslovT);
  var Naslovi2: NasloviT; i, j: integer;
  begin
    { Naslove, ki so manjši kot NaslovPrvega, odložimo v pomožno tabelo. }
    i := 0; while Naslovi[i + 1] <> NaslovPrvega do
      begin i := i + 1; Naslovi2[i] := Naslovi[i] end;
    { Naslove, ki so večji ali enaki NaslovPrvega, premaknemo na začetek tabele. }
    for j := i + 1 do StRacunalnikov do Naslovi[j - i] := Naslovi[j];
    { Na konec tabele zapišemo naslove, manjše od NaslovPrvega. }
    for j := 1 to i do Naslovi[StRacunalnikov - i + j] := Naslovi2[j];
  end; {PrerazporediNaslove}

  function IndeksNaslovaVTabeli(Naslov: NaslovT; var Naslovi: NasloviT): integer;
  var i: integer;
  begin
    i := 1; while Naslovi[i] <> Naslov do i := i + 1;
    IndeksNaslovaVTabeli := i;
  end; {IndeksNaslovaVTabeli}

var
  Naslovi: NasloviT;
  Indeks, k: integer;
begin
  { Če je tole uporabnikovo sporočilo, vemo, da je
    naš računalnik prvi, ki je bil obveščen. }
  if S.PrvoSporocilo then begin
    S.PrvoSporocilo := false;
    S.NaslovPrvega := NasNaslov;
  end; {if}

  { Spomnimo se, da nam podprogram NasloviVsehRacunalnikov
    vrne naslove, urejene naraščajoče. Torej bo vsak računalnik
    po klicu te funkcije videl enak vrstni red. }
  NasloviVsehRacunalnikov(Naslovi);

  { PrerazporediNaslove premakne naslove, manjše od S.NaslovPrvega, na konec
    tabele, drugače pa njihovega medsebojnega vrstnega reda ne spreminja. }
```

```
  PrerazporediNaslove(Naslovi, S.NaslovPrvega);

  { Kateri po vrsti (0..StRacunalnikov − 1) v tabeli naslovov je naš naslov? }
  Indeks := IndeksNaslovaVTabeli(NasNaslov, Naslovi) − 1;

  { Izračunajmo k, torej v kateri sekundi smo mi prejeli tole sporočilo. }
  k := 0; while not (Indeks < 1 shl k) do k := k + 1;

  { V naslednji, torej (k + 1)-vi sekundi, bomo morali
    obvestiti računalnik Indeks + 1 shl k. }
  while Indeks + 1 shl k < StRacunalnikov do begin
    k := k + 1; { Zdaj smo v novi sekundi. }
    { V k-ti sekundi obvestimo računalnik Indeks + 1 shl (k − 1).
      Vendar pa ne pozabimo, da mi štejemo indekse od 0 naprej,
      v tabeli Naslovi pa so od 1 naprej. }
    PosljiSporocilo(S, Naslovi[Indeks + 1 shl (k − 1) + 1]);
  end; {while}

end; {ObPrejemuSporocila}
```

REŠITVE NALOG ZA TRETJO SKUPINO

R2003.3.1 Napadalne kraljice

Za začetek si oglejmo nekaj rezultatov, dobljenih z metodo „razveji in omeji“ (*branch and bound*) in s simuliranim ohlajanjem. Naslednja tabela kaže najmanjše število kraljic, potrebnih, da napademo vsa polja šahovnice $n \times n$:[98] N: 535

[98] Glej tudi: *The On-Line Encyclopedia of Integer Sequences*, A075458 (minimalno potrebno število kraljic, ki napadejo vsa polja), A002563 (z dodatnim pogojem, da ne smejo napadati druga druge); Matthew D. Kearse, Peter B. Gibbons: *Computational methods and new results for chessboard problems*, Australasian Journal of Combinatorics, 23:253–284, March 2001 (tudi: Tech. Rept. CDMTCS-133, Centre for Disc. Math. and Theoretical Comp. Sci., CS Dept., Univ. of Auckland, NZ, May 2000); Patric R. J. Östergård, William Douglas Weakley: *Values of domination numbers in the queen's graph*, The Electronic Journal of Combinatorics, 8(1):R29, 2001. Naj bo a_n minimalno število kraljic, potrebnih, da napademo vsa polja šahovnice $n \times n$. Za vse šahovnice $n \times n$ do $n = 122$ (in še za nekaj večjih n) je znano, da je $a_n \in \{\lceil n/2 \rceil, \lceil n/2 \rceil + 1\}$ (edini izjemi sta $n = 3$ in $n = 11$, kjer je dovolj že $\lfloor n/2 \rfloor$ kraljic); za 53 izmed teh n-jev je znano že tudi, kakšna je res prava vrednost a_n (največkrat je to $\lceil n/2 \rceil$, med drugim tudi za vse n-je, ki so oblike $4t + 1$, vse do vključno $n = 129$). Za vsak n pa velja $a_n \geq \lfloor n/2 \rfloor$.

Za šahovnice do vključno $n = 13$ smo se s pregledom vseh možnih razporedov prepričali, da z manj kot v gornji tabeli navedenim številom kraljic ne gre; za nadaljnje n pa takega preizkusa nismo naredili, saj bi trajal predolgo (zato so v gornji tabeli znaki $\leq$). Vendar pa iz članka Östergårda in Weakleya sledi, da so vse tudi vse nadaljnje meje, navedene v gornji tabeli, dejansko tesne (z manj kraljicami ne moremo napasti vseh polj), le pri $n = 20$ še ni znano, če ni morebiti dovolj že tudi samo deset kraljic. Vendar pa, če razpored z desetimi kraljicami za šahovnico 20×20 obstaja, ga je presneto težko najti, saj ga tudi po večdnevnem iskanju s simuliranim ohlajanjem nismo našli. Ker pri drugih tu preizkušenih n večinoma ni tako težko priti do razporeda z minimalnim številom kraljic, se nagibamo k

n	1	2	3	4	5	6	7	8	9	10	
št. kraljic	1	1	1	2	3	3	4	5	5	5	
n	11	12	13	14	15	16	17	18	19	20	21
št. kraljic	5	6	7	≤ 8	≤ 9	≤ 9	≤ 9	≤ 9	≤ 10	≤ 11	≤ 11

Na šahovnici $n \times n$ se da s k kraljicami napasti vsaj toliko polj:

n	$k=$	5	6	7	8	9	10	11
12		134	144					
13		153	165	169				
14		172	186	194	196			
15		193	209	221	224	225		
16		212	231	242	252	256		
17		233	255	269	282	289		
18		252	277	294	310	324		
19		273	301	321	341	357	361	
20		292	323	348	370	385	398	400
21		313	347	377	401	419	435	441

Rezultati, ki jih je bilo malo težje najti[99] (ostale rezultate gornje tabele najde simulirano ohlajanje zelo hitro): (14, 7, 194); (15, 7, 221); (17, 8, 282); (17, 9, 289); (18, 9, 324); (19, 9, 357); (19, 10, 361); (20, 10, 398); (20, 11, 400); (21, 10, 435); (21, 11, 441) (že (21, 11, 439) se ne najde zelo hitro; pač pa ni težko dobiti (21, 12, 441)).

Ker učinkovitega algoritma, ki bi pri danem številu kraljic in velikosti šahovnice zagotovljeno poiskal najboljši razpored, najbrž sploh ni, nam preostane le to, da poskušamo s kakšnimi hevristikami preizkusiti veliko razporedov, se osredotočati na čim bolj obetavne razporede in končno odkriti nek čim boljši razpored.

Preprost postopek bi bil kar ta, da kraljice na šahovnico **razporedimo naključno**; pazimo le na to, da jih ne bi po več pristalo na istem polju. Tak naključni razpored sicer običajno najbrž ne bo pretirano dober, ker pa lahko tak razpored sestavimo in ocenimo zelo hitro, si lahko privoščimo preizkusiti veliko naključnih razporedov. Na koncu izpišimo najboljšega od vseh preizkušenih razporedov; z malo sreče ta vendarle ne bo tako zelo slab. Seveda pa je z naključnim razporejanjem težko priti do res vrhunskih rezultatov; na primer, pri 6 kraljicah in šahovnici 13×13 je mogoče napasti največ 165 polj, vendar to doseže le 72 razporedov, vseh pa je $N = \binom{13\cdot 13}{6} = 29\,581\,203\,652$. Verjetnost, da je nek naključni razpored slabši, je torej $1 - 72/N$; verjetnost, da je slabši vsak od c preizkušenih razporedov, je zato $(1-72/N)^c$; verjetnost, da je vsaj eden najboljši, je torej $1-(1-72/N)^c$. Pri $c = 10^6$ poskusih nam ta

mnenju, da je pri $n = 20$ najbrž vendarle potrebnih enajst kraljic.

[99] Npr. ker je moralo simulirano ohlajanje teči nekaj deset sekund, preden je našlo tak razpored. Lahko pa praktično vedno pri istih n in k hitro najdemo razpored z le malo manj napadenimi polji.

		Naključno razporejanje								Naključno razporejanje + 2-opt							
n	$k=$	5	6	7	8	9	10	11	12	5	6	7	8	9	10	11	12
12		2	6							0	2						
13		0	8	7						0	0	4					
14		7	9	9	6					0	1	8	0				
15		8	13	14	10	6				0	3	5	2	2			
16		13	18	14	16	12				0	0	2	2	2			
17		15	15	18	17	17				0	0	1	11	11			
18		17	18	17	20	25				0	9	0	0	13			
19		19	24	20	25	29	24			0	3	10	3	4	6		
20		19	32	21	29	30	30	23		0	1	2	1	17	16	8	
21		24	24	31	35	36	33	30	23	0	3	2	6	6	7	6	5

Za vsak par (n, k) smo preizkusili milijon naključnih razporedov, kar je za vsak (n, k) vzelo povprečno po deset sekund (na računalniku z 800-megaherčnim procesorjem). Levi del tabele kaže, za koliko je bil najboljši med temi naključnimi razporedi slabši od najboljšega znanega razporeda. Potem smo na najboljšem naključnem razporedu poskusili izvajati še 2-opt, dokler se je s tem razpored kaj izboljševal; desni del tabele kaže, koliko je bil končni razpored slabši od najboljšega znanega. (Ti rezultati so seveda odvisni od vrednosti, ki jih je vračal generator naključnih števil. Če bi pognali program še enkrat z drugačnim semenom, bi mogoče odkrili še kak boljši razpored.)

---

formula pove, da imamo le 0,24 % možnosti, da bi odkrili enega od najboljših razporedov (v povprečju pa lahko, kot se izkaže, pričakujemo, da bo imel najboljši izmed milijona preizkušenih razporedov napadenih okoli 157 polj). Če bi hoteli naše možnosti dvigniti na 90 %, bi morali preizkusiti skoraj milijardo razporedov.

Po tistem, ko smo s preizkušanjem naključnih razporedov našli nek netako-zelo-slab razpored, ga lahko poskušamo še malo izboljšati. Lahko na primer poskusimo na vse možne načine premakniti dve kraljici (ostalih $k-2$ pa pustimo pri miru). (Tovrstni lokalni optimizaciji včasih pravijo **2-opt**.) Dve kraljici izmed k si lahko izberemo na $\binom{k}{2} = k(k-1)/2$ načinov, njuna položaja (pri čemer nočemo, da bi bili na istih poljih kot ostale kraljice) pa na $\binom{n^2-(k-2)}{2}$ načinov. Vsega skupaj moramo torej preizkusiti približno $k^2n^4/4$ novih razporedov; pri vrednostih k in n, s kakršnimi imamo opravka mi, je to še sprejemljivo, čeprav ne več bliskovito hitro. Če je kakšen od teh novih razporedov boljši od začetnega, vzemimo najboljšega izmed novih razporedov in na njem isti postopek ponovimo: mogoče se da dobiti še kaj boljšega, če premikamo zdaj še kakšni drugi kraljici. Pri naših poskusih tega postopka običajno ni bilo treba izvesti več kot trikrat ali štirikrat. Izkaže se, da s tem pridemo do že kar precej dobrih rezultatov, sploh če je kraljic bolj malo (npr. le pet ali šest).

Sestavljanja razporeda pa se lahko lotimo tudi drugače. Poskusimo postavljati kraljice na šahovnico eno za drugo; ker si želimo, da bi bilo napadenih čim več polj, lahko poskusimo vsako naslednjo postaviti na tako mesto, da bo na-

		Požrešni algoritem ($p = 1$)								Rekurzija ($p = 3$ in $p = \lceil (10^6)^{1/k} \rceil$)							
n	$k =$	5	6	7	8	9	10	11	12	5	6	7	8	9	10	11	12
12		1	4							–	2						
13		2	5	4						–	–	–					
14		–	1	2	–					–	–	2	–				
15		2	4	2	1	–				–	–	3	–	–			
16		–	1	–	1	1				–	–	–	–	–			
17		2	4	2	3	3				–	–	–	1	–			
18		–	1	–	1	7				–	–	–	1*	4			
19		2	4	2	4	8	4			–	–	–	–	4	2		
20		–	1	4	8	8	9	5		–	–	2*	3*	1*	4	–	
21		2	3	6	7	6	9	7	2	–	–	–	–	–	4	2	–

Ta tabela kaže, za koliko se razlikujejo od najboljših znanih rezultatov rešitve, ki smo jih našli s požrešnim algoritmom (levi del tabele) in z rekurzijo (desni del). (Požrešni algoritem je pravzaprav poseben primer rekurzivnega: kot da bi vzeli $p = 1$.) Pri rekurziji smo p izbirali na dva načina: vedno smo poskusili vzeti $p = 3$, kar pomeni, da se rekurzija izteče že po nekaj sekundah; kot drugo možnost pa smo p prilagodili številu kraljic po formuli $p = \lceil (10^6)^{1/k} \rceil$, tako da je število preizkušenih razporedov vedno nekaj čez milijon. Ti dve možnosti sta dali skoraj vedno enako dobre rezultate in zato desna tabela velja za obe; razlikujeta se le v nekaj primerih, ko daje večji p rezultate, enakovredne najboljšim znanim, $p = 3$ pa nekaj slabše. Ti primeri so v tabeli označeni z zvezdico in številka poleg nje se nanaša na rezultat za $p = 3$. Ker so rezultati rekurzije zelo pogosto enakovredni najboljšim znanim razporedom, smo namesto ničel pisali znak $-$, da je neničelne razlike lažje opaziti.

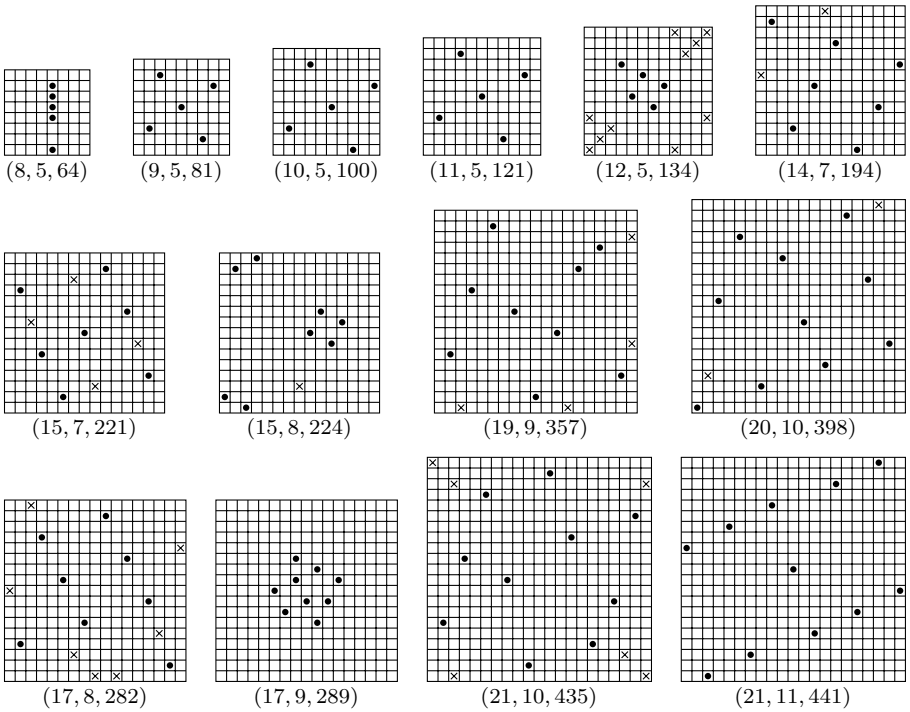
---

padla čim več takih polj, ki jih prejšnje kraljice niso napadale. Tak **požrešen postopek** je zelo hiter in tudi ne daje slabih rezultatov.

Požrešni postopek lahko dopolnimo s **sestopanjem**. Po tistem, ko razpostavimo vse kraljice in ocenimo dobljeni razpored, lahko poskusimo zadnjo kraljico vzeti s šahovnice in nato predzadnjo kraljico postaviti na kakšno drugo polje; potem bi spet vzeli najboljši položaj zadnje kraljice. Ko preizkusimo več položajev predzadnje kraljice, poskusimo spremeniti tudi položaj predpredzadnje kraljice in nato spet preizkusimo več položajev predzadnje kraljice; itd. Ta postopek je pravzaprav rekurzija, ki poskusi dodati trenutno kraljico na razna mesta na šahovnici in pri vsakem izvede še en rekurzivni klic, da bi razmestila še preostale kraljice.

Če hočemo za vsako kraljico preizkusiti p položajev, kraljic pa je k, bomo vsega skupaj preizkusili p^k razporedov. Če imamo na primer deset kraljic, je 3^{10} še čisto sprejemljivo, 10^{10} pa najbrž že ne več. Paziti moramo torej, da ne vzamemo prevelikega p. Vsako kraljico poskusimo postaviti le na nekaj najobetavnejših položajev. To, kako obetaven je nek položaj, lahko ocenjujemo enako kot pri požrešnem postopku: položaj nove kraljice je tem bolj obetaven, čim več doslej nenapadenih polj lahko tja postavljena kraljica napade.

Če sta šahovnica in število kraljic dovolj majhni, lahko pri tej rekurzivni



Nekaj dobrih razporedov kraljic na šahovnice. Številke pod vsako šahovnico pomenijo velikost šahovnice (n), število kraljic (k) in število napadenih polj. Črne pike so kraljice, znak $\times$ pa pomeni nenapadeno polje. Še nekaj zanimivih razporedov lahko dobimo, če tu prikazanim dodamo ali odvzamemo kakšno kraljico: $(13, 7, 169)$ in $(12, 6, 144)$ iz $(11, 5, 121)$; $(14, 8, 196)$ iz $(14, 7, 194)$; $(15, 9, 225)$ in $(16, 9, 256)$ iz $(17, 9, 289)$; $(18, 9, 324)$ in $(19, 10, 361)$ iz $(19, 9, 357)$; $(20, 11, 400)$ iz $(20, 10, 398)$.

---

rešitvi preizkusimo tudi vse možne položaje naslednje kraljice in s tem tudi vse razporede cele skupine k kraljic. S tem se lahko prepričamo, da pri šahovnici 8×8 z manj kot petimi kraljicami ne moremo napasti vseh polj, pa da pri 12×12 s petimi kraljicami ne moremo napasti več kot 134 od 144 polj ipd. Večjih problemov od tega slednjega pa na ta način najbrž že ne bi več mogli rešiti v sprejemljivem času.[100] Mogoče pa bi rekurzija hitro našla neko precej

[100] Preizkušanje vseh razporedov 6 kraljic na šahovnici 13×13 (ki jih je dobrih 29,5 milijard) je trajalo na računalniku z dvema 2400-MHz procesorjema približno 13 ur in pol (program je bil dvoniten, tako da sta bila ves čas zasedena oba procesorja). No, s tem smo se vsaj prepričali, da s šestimi kraljicami na tolikšni šahovnici ne moremo napasti več kot 165 polj. Pri tej hitrosti (približno 600 000 razporedov na sekundo) bi trajal pregled vseh razporedov sedmih kraljic na šahovnico 14×14 (ki jih je slaba dva bilijona) skoraj 38 dni.

Res pa je, da lahko takšen rekurzivni postopek še precej izboljšamo z raznimi hevristikami, ki poskušajo čim prej odkriti, če je trenutni razpored neobetaven (ne bo vodil do rešitve),

dobro rešitev in bi jo lahko po nekaj časa preprosto prekinili in izpisali najboljšo dotlej najdeno rešitev.

Še en optimizacijski postopek, ki se pri tej nalogi kar dobro odreže, je **simulirano ohlajanje**. Pri tem postopku poskušamo trenutni razpored naključno spremeniti (na primer tako, da spremenimo položaj ene od kraljic). Če je novi razpored boljši, to spremembo obdržimo; če pa bi bil novi razpored slabši, ga z neko verjetnostjo vendarle sprejmemo, z neko verjetnostjo pa ga zavrnemo. To verjetnost običajno izberemo tako, da pada eksponentno z razliko med razporedoma: če napade novi razpored le m' polj, prvotni pa m polj, ga obržimo z verjetnostjo $c^{m-m'}$ za nek $c < 1$, na primer $c = 1/2$. (Če smo blizu kakšne zelo dobre rešitve (npr. napadamo že skoraj vsa polja), lahko c še zmanjšamo, da ne bi program prehitro obupoval in si poslabšal razporeda.) Namen tega pravila za sprejemanje sprememb na slabše je, da bi programu, če se zapleza v nek lokalni optimum, omogočili tudi, da se izvleče iz njega; obenem pa mu hočemo preprečiti, da bi po nemarnosti sprejemal neumne spremembe (zato, če je novi razpored res precej slabši od prvotnega, bo verjetnost sprejema tako majhna, da ga bomo skoraj gotovo zavrnili). Naključno spreminjanje razporeda pa programu omogoča, da raziskuje prostor možnih razporedov in, upajmo, sčasoma najde predele z obetavnimi razporedi. (Da se program ne bi takoj po premiku na slabše povrnil nazaj v isti razpored, iz katerega je malo prej prišel (saj bi bil to zdaj zanj premik na bolje in se mu načeloma ne bi mogel upreti), si je koristno zadnjih nekaj (npr. zadnjih trideset) razporedov zapomniti in se vanje ne premikati. Temu običajno pravijo, da smo jih razglasili za tabu.) Simulirano ohlajanje je koristno na vsake toliko časa pognati spet od začetka iz nekega naključno sestavljenega razporeda. Dlje ko ga pustimo teči, več je možnosti, da se bo našla kakšna dobra rešitev. Od tu opisanih postopkov je dajalo simulirano ohlajanje še najboljše razporede. Praktično vedno je že po nekaj sekundah našlo kakšen zelo dober razpored; v nekaj primerih je po minuti ali dveh našlo še kaj boljšega, kasneje pa skoraj nikoli nič (razen pri 10 in 11 kraljicah na plošči 21×21), tudi če smo ga pustili teči po pol ure.

Nekaj pa lahko naredimo tudi ročno. Pri naših poskusih smo morali pustiti simulirano ohlajanje teči približno sedem minut, preden je našlo razpored, ki napade vsa polja šahovnice 19×19 z desetimi kraljicami; pač pa je v manj kot minuti našlo razpored, ki z devetimi kraljicami napade 357 od $19^2 = 361$ polj. Izkazalo se je, da so pri tem razporedu nenapadena polja ležala tako, da bi jih lahko vsa štiri napadli z eno samo dodatno kraljico; tako pridemo do razporeda, ki napade vsa polja šahovnice z desetimi kraljicami. Če šahovnico povečamo na 20×20 in dodamo še eno kraljico v kot, pa dobimo tudi razpored,

---

tako da se z njim ni treba ukvarjati še naprej. Več takih tehnik opisujeta Kearse in Gibbons (razdelek 3), ki sta z njimi pregledala še nekaj naslednjih šahovnic (do 18×18, za katero sta se na ta način prepričala, da je ni mogoče v celoti napasti z 8 ali manj kraljicami).

ki z enajstimi kraljicami napade vsa polja šahovnice 20×20.

R2003.3.2 Smučarji

Do najboljše uvrstitve nekega tekmovalca v skupnem seštevku ni težko priti. To uvrstitev doseže, če zmaga, obenem pa vsi ostali odstopijo. Po novem ima torej namesto a_i točk v skupnem seštevku $a_i + t_1$ točk, ostali pa toliko, kot so jih imeli prej zadnjo tekmo. Treba je le še pogledati, koliko bi jih s tem prehitel. Ker bomo hoteli to izračunati za vse smučarje, si lahko pomagamo s tem, da najboljša možna končna uvrstitev smučarja, ki je bil prej $(s+1)$-vi v skupnem seštevku, prav gotovo ni boljša od najboljše možne končne uvrstitve smučarja, ki je bil prej s-ti. Zato se ta uvrstitev le povečuje, ko gledamo vse slabše smučarje; vsega skupaj je torej s tem le $O(n)$ dela.

N: 538

Če bi dovolili, da se več tekmovalcev uvrsti na isto mesto, bi bilo tudi do najslabše uvrstitve lahko priti: naš tekmovalec naj odstopi, vsi ostali pa naj si delijo prvo mesto. Vendar pa naloga takšnega izida tekme ne dopušča.

Mislimo si tisti izid zadnje tekme, po katerem je naš tekmovalec na najslabšem možnem mestu v skupnem seštevku. V njem lahko po vrsti izvedemo naslednje spremembe, pa se položaj našega tekmovalca v skupnem seštevku gotovo ne bo nič izboljšal:

- Naš tekmovalec lahko odstopi. Zaradi tega dobijo drugi vsaj toliko točk kot prej, naš pa največ toliko kot prej, tako da ga vsi, ki so ga prej v skupnem seštevku prehiteli, prehitijo tudi zdaj. Zato njegova uvrstitev ni nič boljša.

- Odstopijo lahko vsi, ki so bili pred zadnjo tekmo v skupnem seštevku pred našim. Ker naš v zadnji tekmi ne dobi nič točk, bodo vsi ti še vedno pred njim; vsi ostali pa dobijo zaradi te spremembe vsaj toliko točk kot prej in torej tisti, ki bi našega sicer prehiteli, to storijo tudi zdaj.

- Odstopijo lahko vsi, ki našega tekmovalca po tej zadnji tekmi v skupnem seštevku ne prehitijo. Vsi ostali, namreč tisti, ki ga prehitijo, dobijo zaradi tega vsaj toliko točk kot prej in ga zato še vedno prehitijo.

- Označimo zdaj s S_i smučarja, ki je bil v skupnem seštevku pred zadnjo tekmo i mest za našim. Po vseh spremembah, ki smo jih doslej izvedli v izidu zadnje tekme, je ta zdaj tak, da pride do cilja le še nekaj smučarjev S_i za $i > 0$ in vsi ti tudi prehitijo našega smučarja v skupnem seštevku. Recimo, da pridejo na cilj $S_{i_1}, \ldots, S_{i_k}$ (ne nujno v tem vrstnem redu) za neke $1 \le i_1 < i_2 < \cdots < i_k$. Potem očitno velja $i_1 \ge 1$, $i_2 \ge 2$, itd., $i_k \ge k$. Če bi zdaj smučarja S_{i_j} zamenjali s S_j (za vsak $j = 1, \ldots, k$), bi tudi ta uspel prehiteti našega (saj je imel zaradi $j \le i_j$ smučar S_j pred zadnjo tekmo v skupnem seštevku vsaj toliko točk kot S_{i_j}).

- Zdaj smo torej prišli do takega izida, pri katerem pridejo na cilj le smučarji $S_1, \ldots, S_k$ in vsi prehitijo našega v skupnem seštevku. Recimo, da za nek i doseže i-to mesto nek smučar S_j, $(i+1)$-vo pa nek smučar S_r za $r > j$. Če bi se zdaj vrstni red teh dveh ravno zamenjal, bi S_r dobil vsaj toliko točk, kot jih je prej (saj je zdaj uvrščen eno mesto više), tako da bi gotovo še vedno prehitel našega; po drugi strani pa bi S_j dobil toliko točk, kot jih je prej S_r, in ker je $r > j$, je imel S_r od prej v skupnem seštevku kvečjemu toliko točk kot S_j (mogoče pa še manj), tako da, če je S_r-ju uvrstitev na $(i+1)$-vo mesto zadostovala za to, da je prehitel našega, bo S_j-ju še toliko lažje.

- S ponavljanjem prejšnje točke lahko preuredimo izid zadnje tekme tako, da pride na prvo mesto smučar S_k, na drugo S_{k-1}, itd., na predzadnje smučar S_2 in na zadnje smučar S_1.

Videli smo: vsak izid zadnje tekme lahko predelamo v izid take oblike, kot ga opisuje zadnja točka, pa se pri tem uvrstitev, kot jo naš tekmovalec doseže po zadnji tekmi, ne bo nič poslabšala. Torej je dovolj, če se pri iskanju najslabše možne uvrstitve v skupnem seštevku omejimo na izide tega tipa.

Za vsakega smučarja S_i lahko ugotovimo, katera je najslabša uvrstitev, pri kateri še dobi dovolj točk, da bo našega prehitel v skupnem seštevku; recimo, da je to uvrstitev u_i. Po drugi strani pri izidu gornje oblike, če pride v cilj k smučarjev, smučar S_i doseže $(k+1-i)$-to mesto. Če hočemo, da res prehiti našega, mora torej veljati $k+1-i \le u_i$ oziroma $k \le u_i + i - 1$. To mora veljati za vse smučarje, torej za vse i od 1 do k. Veljati mora torej $k \le \min\{u_i + i - 1 : i = 1, \ldots, k\}$. Jasno je, da, če pri nekem k to ne velja več, tudi pri nobenem večjem ne bo (ker se leva stran neenačbe le povečuje, desna pa lahko ostane enaka ali pa se celo zmanjša). Čim naletimo na tak k, lahko nehamo z iskanjem. Drugi možni ustavitveni pogoj je to, da nam zmanjka smučarjev; če je bil naš smučar prej v skupnem seštevku s-ti, ga lahko pač na novo prehiti le $n - s$ smučarjev.

Prav nam bo prišlo tudi dejstvo, da tekmovalec, ki od prej v skupnem seštevku za našim zaostaja bolj kot nek drug, potrebuje v zadnji tekmi vsaj tako dobro uvrstitev kot ta, če naj prehiti našega. Z drugimi besedami, $u_i \ge u_{i+1}$.

Časovna zahtevnost tega postopka za posameznega smučarja je $O(n+\lg m)$. $O(\lg m)$ časa potrebujemo, da z bisekcijo poiščemo u_1; če je ta večji od $n-s$, ga lahko postavimo na $n-s$, saj k nikoli ne bo večji od $n-s$ in zato tudi ne bo potrebe po tem, da bi bil kdo na zadnji tekmi uvrščen na slabše mesto od tega. Za vsakega naslednjega smučarja potem ugotovimo u_{i+1} tako, da začnemo pri u_i in ga po potrebi zmanjšujemo, dokler ne dosežemo uvrstitve, s katero lahko S_{i+1} prehiti našega smučarja. Zato je vseh zmanjševanj pri računanju vrednosti u_i lahko največ u_1 (potem pademo na 0, kar pomeni, da

smo prišli do smučarjev, ki ne morejo niti z zmago prehiteti našega v skupnem seštevku), ta pa je $\leq n - s = O(n)$. Zato imamo tu tako z iskanjem nadaljnjih vrednosti u_i kot s preverjanjem, če še velja zgoraj omenjena neenačba, le $O(n)$ dela. Ker moramo to ponoviti za vsakega smučarja (s gre od 1 do n), je skupna časovna zahtevnost $O(n^2 + n \lg m)$.

```
program Smucarji;
const MaxM = 10000; MaxN = 10000; MaxT = 100000;
var f: text;
    s, ui, i, r, n, m, Najboljsa: integer;
    t: array [0..MaxM] of integer;
    a: array [1..MaxN + 1] of integer;
begin
  Assign(f, 'smucarji.in'); Reset(f);
  { Na konec tabele a dajmo kot stražarja enega čisto brezupnega smučarja, ki ne
    more prehiteti nikogar. Na začetek tabele t dajmo kot stražarja odlično uvrstitev,
    s katero nas lahko prehiti vsakdo, celo tisti brezupni smučar. }
  ReadLn(f, m); t[0] := 2 * MaxT + 2; for i := 1 to m do ReadLn(f, t[i]);
  ReadLn(f, n); a[n] := - MaxT - 1; for i := 1 to n do ReadLn(f, a[i]);
  Close(f); Assign(f, 'smucarji.out'); Rewrite(f);

  Najboljsa := 1;
  for s := 1 to n do begin

    { Poiščimo najboljši možni končni položaj s-tega smučarja. }
    while a[s] + t[1] < a[Najboljsa] do Najboljsa := Najboljsa + 1;

    { V nadaljevanju se ukvarjamo z najslabšim končnim položajem. }
    { Katero mesto mora smučar s + 1 doseči, da nas bo prehitel? }
    ui := 0; if n - s > m then r := m + 1 else r := n - s + 1;
    while ui + 1 < r do begin
      { Invarianta: če doseže tekmovalec s + 1 ui-to mesto,
        bo tekmovalca s v skupnem seštevku prehitel, če
        doseže r-to mesto, pa ne. }
      i := (ui + r) div 2;
      if t[i] + a[s + 1] > a[s] then ui := i else r := i;
    end; {while}

    { Tekmovalec s + 1 mora doseči ui-to ali boljše mesto.
      Če nas prehiti k tekmovalcev, bo dosegel k-to.
      Torej mora biti k ≤ ui. }
    i := 1; r := ui;
    while i <= r do begin
      { Invarianta: r = min(u_j + j - 1 : j = 1, ..., i).
        Zaradi i ≤ r že vemo, da nas lahko prehiti i smučarjev. }
      i := i + 1; { Nas lahko prehiti i + 1 smučarjev? }
      while ui > 0 do
```

```
        if t[ui] + a[s + i] <= a[s] then ui := ui - 1 else break;
      { Tekmovalec s + i potrebuje ui-to mesto, da nas bo prehitel.
        Če nas prehiti k tekmovalcev, bo dosegel (k + 1 - i)-to.
        Torej mora biti k + 1 - i ≤ ui oz. k ≤ ui + i - 1. }
      if r > ui + i - 1 then r := ui + i - 1;
    end; {while i <= r}

    { Lahko nas prehiti i - 1 smučarjev, ne pa tudi i smučarjev. }
    WriteLn(f, Najboljsa, ' ', s + i - 1);
  end; {for s}
  Close(f);
end. {Smucarji}
```

R2003.3.3 Vplivi

N: 539

Odnose med inkvizitorji lahko predstavimo z grafom G, v katerem je za vsakega inkvizitorja po ena točka in če inkvizitor u vpliva na inkvizitorja v, naj bo v grafu povezava „$u \to v$“ od točke u do točke v z vplivom $p(u, v)$. Množico vseh točk označimo z $V = \{1, \ldots, n\}$, množico vseh povezav pa z E. Pot od u_0 do u_k je vsako tako zaporedje točk $\pi = \langle u_0, u_1, \ldots, u_k \rangle$, za katerega so $(u_{i-1} \to u_i) \in E$ za vse $i = 1, \ldots, k$. Vpliv poti je $p(\pi) = \min\{p(u_{i-1}, u_i) : i = 1, \ldots, k\}$. Označimo začetno točko s s (pri tej nalogi je $s = 1$, ker ima veliki inkvizitor številko 1); naj bo $p_M(u)$ vpliv najvplivnejše poti od s do u. Za prazno pot $\pi = \langle u_0 \rangle$ je smiselno definirati $p(\pi) = \infty$ (to sledi iz želje, naj vedno velja $\min(A \cup \{a\}) = \min\{\min A, a\}$ in zato $\min \emptyset = \infty$); zato je $p_M(s) = \infty$.

Ko za neko točko u odkrijemo neko novo najboljšo pot od s do nje, je vsekakor pametno pregledati še njene naslednice, torej tiste v, za katere obstaja povezava $u \to v$. Možne poti od s do v so namreč tudi tiste, ki vodijo skozi u, tako da, če smo odkrili do u neko novo najboljšo pot, se jo bo mogoče dalo podaljšati s korakom $u \to v$ v novo najboljšo pot do v. Naš program bo vzdrževal neko množico Q vseh točk, ki jih bo treba na ta način še pregledati. V vsakem koraku vzame neko točko u iz Q, pregleda njene naslednice in če za katero od njih pri tem odkrije novo najboljšo pot, doda to naslednico v Q. Postopek se ustavi, ko se množica Q izprazni. V tabeli $p_M[\cdot]$ bomo hranili vpliv najboljše doslej znane poti do u; na koncu hočemo seveda priti do tega, da bo za vsak u veljalo $p_M[u] = p_M(u)$ (torej: da bomo poznali res prave najboljše poti). Na začetku, ko za noben u (razen $u = s$) ne poznamo še nobene poti od s do u, postavimo $p_M[u]$ na $-\infty$, saj je vpliv vsake poti od s do u enak vplivu neke povezave, ta pa je $> -\infty$, tako da nam ni treba skrbeti, da bi zaradi prevelike začetne vrednosti $p_M[u]$ kakšno pot spregledali.

1 za vsako točko $u \in V$ naj bo $p_M[u] := -\infty$;
2 $Q := \{s\}$; $p_M[s] := \infty$;

```
3   ponavljaj, dokler množica Q ni prazna:
4     naj bo u nek element Q; vzemi u iz Q;
5     za vsako u-jevo neposredno naslednico v
      (torej: za vsako (u → v) ∈ E):
6       c := min{p_M[u], p(u, v)};
7       če je c > p_M[v],
8         p_M[v] := c;
9         če v ∉ Q, dodaj v v Q;
```

Kaj lahko povemo o **časovni zahtevnosti** tega postopka? Glavna zanka (vrstice 3–9) vsakič vzame neko točko iz Q, tako da se lahko izvede le tolikokrat, kolikorkrat se v Q kaj doda. Neko točko pa dodamo v Q le takrat, kadar se ji poveča $p_M[u]$ (in še to le v primeru, če u tisti hip še ni bil v vrsti). Kot smo videli, je $p_M[u]$ vedno enak vplivu neke povezave našega grafa; če označimo število povezav v grafu z m, vidimo, da ima $p_M[u]$ le m možnih vrednosti (pa še začetno vrednost $-\infty$), torej se lahko poveča le m-krat. Tako torej vidimo, da se lahko vsaka točka znajde v množici Q le m-krat. Notranja zanka (vrstice 5–9) pregleda vse naslednice točke u in če mora pregledati po enkrat vsako $u \in V$, pregleda s tem ravno vse povezave v grafu; torej se pri tem izvede skupaj m-krat. Če se vsaka u znajde v množici Q največ m-krat, bo morala notranja zanka pregledati vse povezave največ m-krat, torej se bo skupaj izvedla največ m^2-krat. Ta razmislek nam torej pove, da se zunanja zanka izvede največ nm-krat, notranja pa največ m^2-krat, tako da je časovna zahtevnost našega algoritma $O(m(n+m))$. Ker je rečeno, da obstaja do vsake točke u vsaj ena pot od s, mora v vsako u (razen mogoče v s) kazati vsaj ena povezava, tako da mora biti povezav vsaj $n-1$; zato lahko $O(m(n+m))$ poenostavimo kar v $O(m^2)$. Po drugi strani je število povezav, m, navzgor omejeno s tem, koliko je vseh parov točk (pri tem se še spomnimo, da nobena točka ne kaže sama nase, saj pravi naloga, da je $p(u,u) = 0$); torej je $m \leq n(n-1)$. Zato je časovna zahtevnost našega postopka v najslabšem primeru $O(n^4)$ (če je graf dovolj gost — se pravi, če ima veliko povezav), znala pa bi biti tudi le $O(n^2)$, če je graf dovolj redek (torej če ima dovolj malo povezav).

Dokaz pravilnosti. Prepričajmo se še, da naš postopek res vedno poišče prave rešitve, torej da na koncu izvajanja za vsak u velja $p_M[u] = p_M(u)$. Definirajmo v mislih nov graf G' z istimi točkami V kot doslej, le množica povezav E' bo malo manjša:

$$(u \to v) \in E' \Leftrightarrow (u \to v) \in E \wedge p_M(v) = \min\{p_M(u), p(u,v)\}.$$

Za prvotni graf G nam že opis naloge zagotavlja, da so v njem vse točke dosegljive iz s. Prepričajmo se zdaj, da velja to tudi za G'. Pa recimo, da neka v v grafu G' ni dosegljiva iz s. Naj bo W množica vseh točk, do katerih lahko v G' pridemo, če začnemo iz v in gremo v nasprotni smeri povezav.

Očitno $s \notin W$ (sicer bi bila v dosegljiva iz s tudi v G'). Naj bo W' množica vseh tistih točk iz W, ki imajo maksimalno $p_M(\cdot)$ (torej: vsaj tolikšno kot katerakoli druga točka iz W). Naj bo $(u \to w)$ poljubna povezava (v G) med neko $u \notin W'$ in $w \in W'$. (Neka taka povezava gotovo obstaja, saj W ni prazna, torej tudi W' ni prazna in ker je v G vsaka $w \in W'$ dosegljiva iz s, s pa ni v W', mora biti na vsaki poti od s do w prej ali slej neka povezava iz $V - W'$ v W'.) V E' te povezave ni, saj za $u \notin W$ to ne gre že po definiciji W, za $u \in W - W'$ pa ne gre zato, ker bi bilo sicer (po definiciji E') $p_M(u) \geq p_M(w)$ in bi bil tedaj po definiciji W' tudi u v W'. Ker povezave $(u \to w)$ ni v E', je pa v E, mora biti (po definiciji E') $p_M(w)$ strogo večji od $\min\{p_M(u), p(u,w)\}$. (Po definiciji E' sledi pravzaprav, da mora biti različen, toda manjši ne more biti, kar je jasno že iz definicije funkcije p_M.) — Naj bo π najboljša pot od s do neke $w' \in W'$. Ker se začne v $s \notin W'$, mora vsaj enkrat prestopiti iz $V - W'$ v W', torej mora vsebovati neko povezavo $(u \to w)$ za nek $u \notin W'$ in nek $w \in W'$. Pot π lahko v mislih predelamo tako, da tisti del poti do u zamenjamo z najboljšo potjo do u; nastali poti, ki gotovo ni slabša od prvotne, recimo π'. Torej je $p_M(w') = p(\pi) \leq p(\pi') \leq \min\{p_M(u), p(u,w)\}$; kot pa smo ugotovili malo prej, je to naprej $< p_M(w)$, kar je po definiciji W' enako $p_M(w')$. Zdaj smo prišli v protislovje, češ da je $p_M(w') < p_M(w')$. Tako vidimo, da je nemogoče, da kakšna v v grafu G' ne bi bila dosegljiva iz s.

Vrednosti, ki jih program vpisuje v $p_M[u]$, so vedno enake $p(\pi)$ za neko pot π od s do u. Zmotimo se torej lahko le tako, da je za nek u vrednost $p_M[u]$ tudi na koncu izvajanja programa manjša od prave vrednosti $p_M(u)$, ne more pa biti $p_M[u]$ večja od prave vrednosti. Recimo, da za nek u naš program ne najde prave rešitve. Naj bo π najkrajša pot (najkrajša po številu povezav) od s do u v grafu G' (ki gotovo obstaja, saj smo se malo prej prepričali, da so tudi v G' vse točke dosegljive iz s). Naj bo v prva točka na tej poti, za katero naš program ne najde prave rešitve; naj bo w njena neposredna predhodnica na tej poti (kajti v gotovo ima predhodnico, saj je na tej poti le s brez predhodnice, za s pa poznamo pravo rešitev že na začetku izvajanja programa, tako da $v \neq s$). Ker je w dosegljiva iz s, je $p_M(w) > -\infty$, in ker naš program odkrije pravilno rešitev za w, mora med njegovim izvajanjem $p_M[w]$ vsaj enkrat narasti, da od svoje začetne vrednosti $-\infty$ pride do $p_M(w)$. Torej dodamo w vsaj enkrat v Q; torej ga tudi vsaj enkrat vzamemo iz Q. Ko ga zadnjič vzamemo iz Q, je $p_M[w]$ že enako $p_M(w)$. Takrat pogledamo vse w-jeve naslednike, torej tudi v; pri slednjem izračunamo $c := \min\{p_M[w], p(w,v)\}$. Po predpostavki, da imamo že pravi rezultat za w, je c enak $\min\{p_M(w), p(w,v)\}$; in ker smo w in v izbrali tako, da je $(w \to v) \in E'$, je to naprej enako $p_M(v)$. Torej, tudi če je bil $p_M[v]$ doslej še manjši od $p_M(v)$, bi ga zdaj postavili na to vrednost. Ker se vrednosti tabele $p_M[\cdot]$ nikoli ne zmanjšujejo, bo tudi na koncu $p_M[v] = p_M(v)$, torej bo naš program našel pravilno rešitev za v. Prišli smo v protislovje, saj smo na začetku rekli, da se za v naš program zmoti.

Implementacija množice Q. Postopek torej načeloma deluje in daje pravilne rezultate ne glede na to, kako jemljemo točke iz množice Q. Vendar pa se izkaže, da vrstni red jemanja pomembno vpliva na časovno zahtevnost postopka. Če organiziramo Q kot **sklad**, torej vzamemo iz nje vedno tisto točko, ki smo jo nazadnje dodali vanjo, in če pregledujemo naslednike (v notranji zanki) v nekem dovolj nesrečno izbranem vrstnem redu, se pri nekaterih grafih res lahko zgodi, da ima naš postopek zahtevnost $O(n^4)$.

Če organiziramo Q kot **vrsto**, torej vzamemo iz Q vedno tisto točko, ki je že najdlje v njej, pa se da dokazati, da ne bo nobena točka prišla v vrsto več kot n-krat.[101] Zato mora notranja zanka pregledati vse povezave iz E največ n-krat in časovna zahtevnost našega postopka je tedaj le še $O(nm)$, kar je v najslabšem primeru $O(n^3)$, nikoli pa ne $O(n^4)$. Tako dobljen postopek je pravzaprav različica Bellman-Fordovega algoritma za iskanje najkrajših poti v grafih (le da namesto najkrajših išče najvplivnejše poti).

Še bolje je, če vzamemo iz Q vedno tisto točko u, ki ima med vsemi v Q največjo vrednost $p_M[u]$. Temu pravimo, da smo Q organizirali v **prioritetno vrsto**; naš postopek postane različica znanega Dijkstrovega algoritma za iskanje najkrajših poti. Zdaj se da pokazati, da vsako točko vzamemo iz vrste največ enkrat[102] (pravzaprav natanko enkrat, saj je vsaka dosegljiva iz s), tako da se notranja zanka izvede le $O(m)$-krat, zunanja pa $O(n)$-krat. Prioritetno vrsto se običajno implementira z dvojiško kopico, pri kateri dodajanje in brisanje točk zahteva $O(\lg n)$ časa; zahtevnost našega algoritma bi bila v tem primeru $O(m \lg n)$, kar je v najslabšem primeru $O(n^2 \lg n)$.[103]

[101] Načrt dokaza: mislimo si, da za vsako točko u vzdržujemo še neko vrednost $h[u]$, ki je na začetku 0; vsakič, ko pri pregledovanju naslednikov nekega u dodamo nek v v vrsto, pa postavimo $h[v] := h[u]+1$. Potem lahko z indukcijo pokažemo, da na začetku vsake iteracije zunanje zanke velja: obstaja nek k, tako da za prvih nekaj točk v vrsti velja $h[u] = k$, za preostale (nič ali več točk) pa $h[u] = k+1$. Iz te ugotovitve sledi, da ko točke jemljemo iz vrste, dobivamo take z nepadajočimi vrednostmi h. Naj bo $d(v)$ dolžina najkrajše poti od s do v v grafu G'. Potem lahko z indukcijo po $d(v)$ pokažemo, da po zadnji postavitvi točke v v vrsto prav gotovo velja $h[v] \le d(v)$. Na koncu še pokažimo, da se, če neko v večkrat dodamo v vrsto, po vsakem dodajanju njena $h[v]$ strogo poveča. Iz vsega tega že sledi, da nobene v ne moremo dodati v vrsto več kot $d(v)$-krat, $d(v)$ pa je seveda vedno $< n$.

[102] Pri dokazu si je koristno pomagati z naslednjo invarianto, ki velja na koncu vsake iteracije zunanje zanke. Naj bo A množica vseh točk, ki smo jih že kdaj vzeli iz Q. Naj bo B množica vseh tistih neposrednih naslednic točk iz A, ki same niso v A. Naj bo C množica vseh ostalih točk (torej $V - A - B$). (1a) Za vsako $u \in A$ je $p_M[u]$ zdajle že enak vplivu najvplivnejše poti od s do u; (1b) nadalje je med potmi od s do u s tem vplivom tudi vsaj ena taka, ki gre ves čas le po točkah iz A. (2a) Za vsako $v \in B$ je $p_M[v]$ zdajle enak vplivu najvplivnejše take poti od s do v, ki gre ves čas le po točkah iz A, razen v zadnjem koraku, ko stopi v v. (2b) Množica Q vsebuje vse točke iz B in nobene točke iz A ali C. (3) Za vsako $w \in C$ je $p_M[w] = -\infty$.

[103] S kakšno drugo vrsto kopic, npr. Fibonaccijevimi, bi šlo asimptotično tudi hitreje, ker bi dodajanje v Q in povečevanje vrednosti p_M vzelo v povprečju le $O(1)$ časa, tako da bi bila časovna zahtevnost celega algoritma le še $O(m + n \lg n)$. Vendar so te kopice bolj zapletene in se jih v praksi najbrž ne uporablja kaj dosti. — Še ena alternativa kopici bi bila, da bi vsakič pregledali kar vse $p_M[u]$ za vse $u \in Q$ in na ta način poiskali največjo. To bi

Ko smo merili časovno zahtevnost na konkretnih testnih primerih pri tej nalogi, se ni različica s prioritetno vrsto odrezala čisto nič hitreje kot tista z navadno vrsto, razen pri tistem testnem primeru, ki je bil nalašč sestavljen tako, da ima vrsta pri njem res zahtevnost $O(n^3)$ (notranja zanka se izvede približno $\frac{1}{2}n^3$-krat[104]). Eden od ostalih testnih primerov je sestavljen tako, da je zelo neugoden za različico s skladom,[105] ostali testni primeri pa so bili pripravljeni naključno. Na vseh teh sta porabili različici s prioritetno in z navadno vrsto praktično enako veliko časa. Zato objavljamo kar rešitev z navadno vrsto, saj je krajša in preprostejša.

```
program Vplivi;
const MaxN = 1000; MaxM = 200000; MaxVpliv = 1000000000;
type TabelaN = array [1..MaxN] of integer;
     TabelaM = array [1..MaxM] of integer;
     TabelaB = array [1..MaxN] of boolean;
var T: text;
    i, u, v, m, n, Glava, Dolz, p, Kand: integer;
    Kdo, NaKoga1, Vpliv1, NaKoga, Vpliv, Vrsta: ↑TabelaM;
    NaKoliko, Prvi, NajVpliv: ↑TabelaN; VVrsti: ↑TabelaB;
```

---

nam vzelo vsakič $O(n)$ časa, torej skupno $O(n^2)$; zato pa je vsak popravek kakšne vrednosti $p_M[u]$ le še $O(1)$, tako da je zahtevnost celega algoritma zdaj $O(n^2+m)$. To je lahko boljše ali pa tudi slabše od $O(m \lg n)$, odvisno od tega, ali je naš graf gost ($m = O(n^2)$) ali redek ($m = O(n)$).

[104]Primer takega grafa: n točk, oštevilčenih z $1, \ldots, n$; od vsake točke u obstajajo povezave do vseh ostalih točk s težo u, razen povezave do $u+1$, ki ima težo $2n-u$. Ta povezava (torej $u \to u+1$) naj bo navedena kot zadnja med vsemi povezavami, ki kažejo iz u. Naš algoritem bi na začetku vzel iz vrste točko 1 (s $p_M[1] = \infty$) in nato dodal v vrsto točke $3, \ldots, n$ s $p_M[u] = 1$ ter za njimi še točko 2 s $p_M[2] = 2n-1$. Ko bi nato jemal točke $3, \ldots, n$ iz vrste, se ne bi spremenilo nič; nato pa bi vzel iz vrste točko 2 in dodal v vrsto točke $4, \ldots, n$ s $p_M[u] = 2$ ter še točko 3 s $p_M[3] = 2n-2$. Tako bi šlo naprej; točko 1 dodamo v vrsto enkrat, ostale točke u pa po $(u-1)$-krat. Tako se izvede $1 + \sum_{u=2}^{n}(u-1) = 1 + n(n-1)/2$ jemanj iz vrste, ob vsakem od njih pa je treba pregledati vseh $n-1$ naslednic trenutne točke, tako da je število izvajanj notranje zanke kar $(1+n(n-1)/2)(n-1) = (n^3-2n^2+3n-2)/2$, skratka, približno $n^3/2$.

[105]Primer takega grafa: imejmo točke $u, v_1, \ldots, v_a, w_1, \ldots, w_b, x_1, \ldots, x_c$; u vpliva na vse v_i s težo ib; vsaka v_i vpliva na vsako w_j s težo $(i-1)b+j$; vsaka w_j vpliva na vsako x_k s težo ab; vsaka x_k vpliva na vse ostale točke s težo 1. Povezave, ki izhajajo iz posamezne točke, naj bodo navedene po padajoči teži. Naš algoritem bi najprej vzel s sklada točko u in naložil nanj $v_a, v_{a-1}, \ldots, v_2, v_1$ z utežmi $p_M[v_i] = ib$. Nato bi vzel s sklada v_1 in naložil nanj $w_b, w_{b-1}, \ldots, w_2, w_1$ z utežmi $p_M[w_j] = j$. Nato bi vsako w_j vzel s sklada, naložil nanj vsakič vse x_k (s $p_M[x_k] = p_M[w_j]$) ter jih takoj spet pobral s sklada. Po vsem tem bi vzel s sklada v_2 in naložil nanj spet $w_b, w_{b-1}, \ldots, w_2, w_1$, vendar zdaj z večjimi utežmi: $p_M[w_j] = j+b$. Zato bi pri jemanju vsake w_j s sklada zdaj na novo dodal na sklad tudi vse x_k (spet s $p_M[x_k] = p_M[w_j]$, le da je $p_M[w_j]$ zdaj večja kot prej). Podobno bi se potem zgodilo pri v_3, v_4 in tako naprej. To pomeni, da se u znajde na skladu enkrat, vsaka v_i tudi enkrat, vsaka w_j se pojavi na skladu a-krat, vsaka x_k pa ab-krat. Če upoštevamo še izhodne stopnje teh točk, vidimo, da se mora notranja zanka izvesti $((1+a)b+abc+abcd)$-krat, če je $d = a+b+c$ izhodna stopnja točk x_k. Če vzamemo $a = b = c = (n-1)/3$, imamo kar $(n^4-3n^3+6n^2+2n-6)/27 \approx n^4/27$ izvajanj notranje zanke.

```
begin
  Assign(T, 'vplivi.in'); Reset(T); ReadLn(T, n); ReadLn(T, m);

  { Preberimo vse povezave (u, v) v tabeli Kdo in NaKoga1.
    V Vpliv1 shranimo vplive povezav, v NaKoliko pa za
    vsakega inkvizitorja zapišemo, na koliko drugih vpliva. }
  New(Kdo); New(NaKoga1); New(Vpliv1); New(NaKoliko);
  for u := 1 to n do NaKoliko↑[u] := 0;
  for i := 1 to m do begin
    ReadLn(T, u, v, p); NaKoliko↑[u] := NaKoliko↑[u] + 1;
    Kdo↑[i] := u; NaKoga1↑[i] := v; Vpliv1↑[i] := p;
  end;
  Close(T); New(Prvi); New(NaKoga); New(Vpliv);

  { Povezave uredimo tako, da bomo imeli za vsakega inkvizitorja
    pri roki seznam vseh, na katere vpliva: za inkvizitorja u
    so to inkvizitorji NaKoga↑[Prvi↑[u]..Prvi↑[u] + NaKoliko↑[u] − 1]. }
  Prvi↑[1] := 1; for u := 2 to n do Prvi↑[u] := Prvi↑[u − 1] + NaKoliko↑[u − 1];
  for u := 1 to n do NaKoliko↑[u] := 0;
  for i := 1 to m do begin
    u := Kdo↑[i];
    NaKoga↑[Prvi↑[u] + NaKoliko↑[u]] := NaKoga1↑[i];
    Vpliv↑[Prvi↑[u] + NaKoliko↑[u]] := Vpliv1↑[i];
    NaKoliko↑[u] := NaKoliko↑[u] + 1;
  end; {for}
  Dispose(Kdo); Dispose(NaKoga1); Dispose(Vpliv1); New(NajVpliv);

  { Glavni del našega postopka. V vrsti (tabela Vrsta) je Dolz
    točk, prva je na indeksu Glava. (Če padejo indeksi čez rob,
    nadaljujemo pri indeksu 1.) VVrsti↑[u] pove, če je u v vrsti. }
  NajVpliv↑[1] := MaxVpliv + 1; for u := 2 to n do NajVpliv↑[u] := −1;
  Glava := 1; Dolz := 1; New(Vrsta); New(VVrsti); Vrsta↑[Glava] := 1;
  VVrsti↑[1] := true; for u := 2 to n do VVrsti↑[u] := false;
  while Dolz > 0 do begin
    u := Vrsta↑[Glava]; { Vzamemo nek u iz vrste. }
    VVrsti↑[u] := false; Dolz := Dolz − 1;
    for i := 1 to NaKoliko↑[u] do begin { Na koga vse lahko u vpliva? }
      v := NaKoga↑[Prvi↑[u] + i − 1]; p := Vpliv↑[Prvi↑[u] + i − 1];
      if p < NajVpliv↑[u] then Kand := p else Kand := NajVpliv↑[u];
      if Kand > NajVpliv↑[v] then begin { Našli smo novo najboljšo pot do v. }
        NajVpliv↑[v] := Kand;
        if not VVrsti↑[v] then begin { Dodajmo v v vrsto. }
          Vrsta↑[(Glava + Dolz) mod n + 1] := v;
          Dolz := Dolz + 1; VVrsti↑[v] := true;
        end; {if}
      end; {if}
    end; {for}
```

```
    Glava := Glava + 1; if Glava > n then Glava := 1;
  end; {while}
  Dispose(Vrsta); Dispose(VVrsti); Dispose(Vpliv);
  Dispose(NaKoga); Dispose(NaKoliko); Dispose(Prvi);

  { Izpis rezultatov. }
  Assign(T, 'vplivi.out'); Rewrite(T);
  for u := 2 to n do WriteLn(T, NajVpliv↑[u]);
  Close(T); Dispose(NajVpliv);
end. {Vplivi}
```

R2003.3.4 Vodenje projektov

N: 540

Podobno kot pri prejšnji nalogi si lahko tudi tu pomagamo z grafom. Vsaka točka ustreza eni od aktivnosti, povezava od u do v (označimo jo „$u \to v$“) pa naj obstaja natanko v primeru, ko je v vhodnih podatkih v navedena kot odvisna od u. To, da je neka aktivnost v posredno ali neposredno odvisna od u, pa pomeni, da je v grafu neka pot (zaporedje povezav) od u do v.

Naloga zdaj pravzaprav zahteva, naj zbrišemo iz grafa čim več povezav (to ustreza brisanju odvisnosti), ne da bi pri tem spremenili relacijo dosegljivosti v njem. (V teoriji grafov pravijo temu problemu *tranzitivna redukcija* grafa.) Z drugimi besedami, če je neka točka dosegljiva iz neke druge v prvotnem grafu, mora ostati dosegljiva iz nje tudi po brisanju. Preden zbrišemo neko povezavo $u \to v$, moramo torej preveriti, da je v dosegljiva iz u tudi po kakšni drugačni (daljši) poti. To lahko učinkovito preverimo takole: naj bo $R(u)$ množica vseh točk, dosegljivih iz u (v enem ali več korakih). Naj bo $P(v)$ množica vseh neposrednih predhodnic točke v (torej takih, iz katerih kaže v v neka povezava). Potem, če je presek $R(u) \cap P(v)$ neprazen, pomeni, da je neka v-jeva predhodnica (recimo ji w) tudi dosegljiva iz u (v enem ali več korakih), zato pa lahko povezavo $u \to v$ zbrišemo, pa se bo od u do v še vedno dalo priti skozi w. (Poti od u do w pa z brisanjem povezave $u \to v$ prav gotovo ne pretrgamo, saj če bi tista pot vsebovala tudi to povezavo, bi pomenilo, da se da iz v iti naprej po tej poti do w in nato (ker je w predhodnica točke v) v v, torej obstaja v grafu cikel — nam pa naloga zagotavlja, da cikličnih odvisnosti ne bo.)

```
1  ponovi za vsako točko u ∈ V:
2    naj bo R(u) množica vseh točk, dosegljivih iz u v enem ali več korakih;
3    za vsako u-jevo neposredno naslednico v (torej: za vsako (u → v) ∈ E):
4      naj bo P(v) množica v-jevih neposrednih predhodnic;
5      če je R(u) ∩ P(v) ≠ ∅,
6        izpiši, da se lahko pobriše povezava u → v;
```

Dokaz pravilnosti. Naš program torej predlaga brisanje neke povezave $u \to v$ natanko tedaj, ko obstaja v prvotnem grafu neka daljša pot $u \rightsquigarrow w \to v$.

Prepričajmo se, da množica tistih povezav iz E, za katere naš postopek ne predlaga brisanja (recimo tej množici F) res ohrani celotno relacijo dosegljivosti prvotne množice E.

Recimo, da bi obstajala v E neka pot

$$u_1 \rightsquigarrow u_2 \rightsquigarrow \ldots \rightsquigarrow u_k \rightsquigarrow v_k \rightsquigarrow \ldots \rightsquigarrow v_2 \rightsquigarrow v_1,$$

kjer za vsak i velja $(u_i \neq u_{i-1}) \vee (v_i \neq v_{i-1})$ in v_i v F ni dosegljiv iz u_i. (Iz slednjega tudi sledi $u_i \neq v_i$.) Recimo še, da je del med u_k in v_k dolg vsaj dva koraka (dve povezavi). (Opazimo, da je dolžina cele poti vsaj $k-1$ korakov.)

Naj bo π del gornje poti med u_k in v_k. Če bi za vsako povezavo $u \to v$ iz π veljalo, da je v v F dosegljiv iz u, bi bil tudi v_k v F dosegljiv iz u_k. Torej obstaja med temi povezavami neka $u_{k+1} \to v_{k+1}$, da v_{k+1} v F ni dosegljiv iz u_{k+1}. Ker je π po predpostavki dolga vsaj dva koraka (in ker v našem grafu ni ciklov), to seveda pomeni, da ne more biti obenem $u_{k+1} = u_k$ in $v_{k+1} = v_k$ — vsaj nekaj od tega dvojega ni res. Poleg tega, ker v F točka v_{k+1} ni dosegljiva iz u_{k+1}, je moral naš program med drugim pobrisati tudi povezavo $u_{k+1} \to v_{k+1}$, torej obstaja v E neka daljša (vsaj dva koraka dolga) pot π' od u_{k+1} do v_{k+1}. Vidimo torej: če velja predpostavka iz prejšnjega odstavka (torej da obstaja pot z navedenimi lastnostmi) za k, potem velja tudi za $k+1$.

Čim torej obstaja v E neka pot $u_1 \rightsquigarrow v_1$, dolga vsaj dva koraka, in v_1 v F ni dosegljiv iz u_1, bi lahko n-krat uporabili gornji razmislek in videli, da mora v E obstajati neka pot od u_1 do v_1, dolga vsaj n korakov; toda tako dolge poti v grafu na samo n točkah ne more biti, ne da bi vsebovala cikel, ciklov pa naš graf ne vsebuje.

Recimo torej, da je nek v_1 dosegljiv iz nekega u_1 v E, ne pa v F. Prejšnji odstavek je pokazal, da v E ne obstaja nobena pot od u_1 do v_1, dolga vsaj dva koraka. Ker pa je v E točka v_1 dosegljiva iz u_1, pomeni, da v E obstaja povezava $u_1 \to v_1$; in ker v F točka v_1 ni dosegljiva iz u_1, pomeni, da je naš postopek tisto povezavo $u_1 \to v_1$ pobrisal; to pa pomeni, da je videl v E še neko daljšo (vsaj dva koraka dolgo) pot od u_1 do v_1, mi pa smo rekli, da take ni. Prišli smo v protislovje, torej takega neugodnega para u_1 in v_1 ni.

Dosedanji razmislek je pokazal, da nismo z brisanjem izgubili nobene dosegljivosti; obenem pa z brisanjem seveda tudi ni mogoče nobene dosegljivosti pridobiti, tako da zdaj vemo, da v množici povezav, ki jih po brisanju pusti naš postopek, res veljajo natanko iste dosegljivosti kot v prvotni množici povezav E. Torej rešitev, ki jo vrne naš program, ustreza zahtevam naloge; prepričajmo se še o tem, da je tudi najboljša možna.

Recimo, da imamo dve množici povezav, F_1 in F_2, obe dobljeni iz E z brisanjem nekaj povezav; in recimo, da je tako pri F_1 kot pri F_2 dosegljivost v grafu enaka kot pri celi množici E. Recimo, da obstaja v F_1 neka povezava $u \to v$, ki je v F_2 ni. Ker F_1 ima to povezavo, je v v prvotnem grafu dosegljiv iz u, torej mora biti v F_2 tudi, na primer po neki poti $u_0 \to u_1 \to \ldots \to$

$u_{k-1} \to u_k$ za $u_0 = u$, $u_k = v$. Ker so vse te povezave $u_{i-1} \to u_i$ v F_2, so morale biti že v E, torej mora biti vsaka u_i tudi v F_1 dosegljiva iz u_{i-1}. Če se v F_1 od u_{i-1} do u_i ne bi dalo priti drugače kot tako, da obiščemo tudi povezavo $u \to v$, bi to pomenilo, da obstaja cikel $u = u_0 \rightsquigarrow u_1 \rightsquigarrow \ldots \rightsquigarrow u_i \rightsquigarrow u$ ali pa cikel $v \rightsquigarrow u_{i+1} \rightsquigarrow u_{i+2} \rightsquigarrow \ldots \rightsquigarrow u_{k-1} \rightsquigarrow u_k = v$;[106] torej bi bil tak cikel že v prvotnem grafu, ta pa je, kot zagotavlja naloga, acikličen, tako da se to ne more zgoditi. Torej se da v F_1 od vsake u_{i-1} priti do u_i brez obiska povezave $u \to v$; torej se da tudi od u_0 $(= u)$ priti do u_k $(= v)$, ne da bi obiskali povezavo $u \to v$; torej lahko to povezavo brez škode pobrišemo iz F_1, pa se dosegljivost v grafu ne bo nič spremenila.

Prejšnji odstavek nam med drugim pove, da ne moreta obstajati dve različni najboljši rešitvi (iz gornjega razmisleka bi takoj sledilo, da lahko eno od njiju še zmanjšamo in da prej torej ni bila najboljša). Pove pa nam tudi, da je rešitev, ki jo predlaga naš program, res najboljša. Kajti če ni najboljša, je moralo ostati po brisanju več povezav kot pri najboljši, to pa zagotovo pomeni, da ima naša rešitev kakšno tako povezavo $u \to v$, ki je najboljša nima. Ker obenem ohranja dosegljivost, lahko uporabimo razmislek iz prejšnjega odstavka in vidimo, da bi lahko iz naše rešitve F kakšno povezavo $u \to v$ še pobrisali, ker obstaja v F namesto nje še neka daljša pot od u do v. Toda ta daljša pot obstaja potemtakem tudi v E in bi jo zato naš postopek moral opaziti in $u \to v$ pobrisati in je potem sploh ne bi bilo v F.

Pri **implementaciji** postopka je koristno, če si za vsako točko pripravimo seznam predhodnic in seznam naslednic. Seznami naslednic bodo prišli prav pri ugotavljanju, kaj vse je dosegljivo iz neke u. Ko za neko točko na novo izvemo, da je dosegljiva iz u, jo dodajmo v neko vrsto, da bomo kasneje pregledali še njene naslednice (če jih kaj ima), saj so one potemtakem tudi dosegljive iz u. Takemu načinu pregledovanja grafa pravimo tudi *iskanje v širino*. To, ali je neka točka v dosegljiva iz u, si spodnji program zapomni v celici Dosegljiva[v]. Da ni treba za vsak u posebej postavljati cele tabele Dosegljiva na false, si pomagamo tako, da namesto vrednosti true uporabimo kar vrednost u, katerakoli manjša vrednost pa velja kot false; na začetku postavimo vse celice te tabele na 0. Kakorkoli že, tabelo Dosegljiva lahko potem uporabimo ob pregledovanju v-jevih predhodnic, da vidimo, če je katera od njih tudi dosegljiva iz u.

```
program VodenjeProjektov;
const MaxN = 1000; MaxM = 10000;
type TabelaN = array [1..MaxN] of integer;
     TabelaM = array [1..MaxM] of integer;
var T: text; Zbrisi: boolean;
```

[106] Oba tadva sprehoda sta možna; stvar je le v tem, da nista nujno oba cikla — če je $i = 0$ in $u = u_i = u_0 = u$, potem prvi od njiju sploh ni cikel; podobno pa, če je $i = k - 1$ in $v = u_{i+1} = u_k = v$, drugi od njiju ni cikel; oboje naenkrat pa ne more biti res, ker bi to pomenilo $k = 1$ in F_2 bi vseboval kar povezavo $u \to v$, to pa smo predpostavili, da je ne.

```
    m, n, i, j, u, v, w, Glava, Rep: integer;
    InDeg, OutDeg, PrviPred, PrviNasl, Vrsta, Dosegljiva: ↑TabelaN;
    Zac, Kon, Pred, Nasl: ↑TabelaM;
begin
  Assign(T, 'projekti.in'); Reset(T); ReadLn(T, n); ReadLn(T, m);

  { Preberimo seznam povezav. }
  New(InDeg); New(OutDeg); New(Zac); New(Kon);
  for u := 1 to n do begin InDeg↑[u] := 0; OutDeg↑[u] := 0 end;
  for i := 1 to m do begin
    ReadLn(T, u, v); Zac↑[i] := u; Kon↑[i] := v;
    OutDeg↑[u] := OutDeg↑[u] + 1; InDeg↑[v] := InDeg↑[v] + 1;
  end; {for}
  Close(T); New(PrviPred); New(PrviNasl); New(Pred); New(Nasl);

  { Pripravimo sezname predhodnic in naslednic.
    u-jeve predhodnice bodo Pred↑[PrviPred↑[u]..PrviPred↑[u] + InDeg↑[u] - 1],
    naslednice pa Nasl↑[PrviNasl↑[u]..PrviNasl↑[u] + OutDeg↑[u] - 1]. }
  PrviPred↑[1] := 1; PrviNasl↑[1] := 1;
  for u := 2 to n do PrviPred↑[u] := PrviPred↑[u - 1] + InDeg↑[u - 1];
  for u := 2 to n do PrviNasl↑[u] := PrviNasl↑[u - 1] + OutDeg↑[u - 1];
  for u := 1 to n do begin InDeg↑[u] := 0; OutDeg↑[u] := 0 end;
  for i := 1 to m do begin
    u := Zac↑[i]; v := Kon↑[i];
    Nasl↑[PrviNasl↑[u] + OutDeg↑[u]] := v; OutDeg↑[u] := OutDeg↑[u] + 1;
    Pred↑[PrviPred↑[v] + InDeg↑[v]] := u; InDeg↑[v] := InDeg↑[v] + 1;
  end; {for}
  Dispose(Zac); Dispose(Kon); New(Vrsta); New(Dosegljiva);

  { Preglejmo vsako točko. }
  Assign(T, 'projekti.out'); Rewrite(T);
  for u := 1 to n do Dosegljiva↑[u] := 0;
  for u := 1 to n do begin

    { Kaj vse je dosegljivo iz u? }
    Glava := 1; Rep := 1; Vrsta↑[1] := u; Dosegljiva↑[u] := u;
    while Glava <= Rep do begin
      v := Vrsta↑[Glava]; Glava := Glava + 1;
      for i := 1 to OutDeg↑[v] do begin { Preglejmo v-jeve naslednice. }
        w := Nasl↑[PrviNasl↑[v] + i - 1];
        if Dosegljiva↑[w] <> u then { Dodajmo w v vrsto. }
          begin Rep := Rep + 1; Vrsta↑[Rep] := w; Dosegljiva↑[w] := u end;
      end; {for}
    end; {while}

    { Preglejmo vse povezave u → v, ki izhajajo iz u. }
    for i := 1 to OutDeg↑[u] do begin
      v := Nasl↑[PrviNasl↑[u] + i - 1]; Zbrisi := false;
```

```
        { Ali je kakšna v-jeva predhodnica tudi dosegljiva iz u? }
        for j := 1 to InDeg↑[v] do begin
          w := Pred↑[PrviPred↑[v] + j − 1];
          if (Dosegljiva↑[w] = u) and (w <> u) then
              begin Zbrisi := true; break end;
        end; {for j}
        if Zbrisi then WriteLn(T, u, ' ', v);
      end; {for i}
    end; {for u}

    Dispose(Vrsta); Dispose(Dosegljiva); Dispose(Nasl); Dispose(Pred);
    Dispose(PrviNasl); Dispose(PrviPred); Close(T);
end. {VodenjeProjektov}
```

Naj bo n število točk in m število povezav našega grafa. Ugotavljanje, kaj je dosegljivo iz posamezne točke, nam vzame le $O(m)$ časa — bilo bi $O(m+n)$, če bi morali vsakič sproti inicializirati celo tabelo Dosegljiva, vendar se temu, kot smo videli, lahko izognemo. Ker je treba ta postopek pognati po enkrat za vsako točko grafa, je skupna zahtevnost $O(mn)$ in poleg tega še $O(n)$ za začetno inicializacijo tabele Dosegljiva. Notranja zanka, ki pregleduje naslednice trenutnega u, se izvede vsega skupaj m-krat (po enkrat za vsako povezavo) in ima vsakič v najslabšem primeru $O(n)$ dela (kolikor ima pač trenutni v neposrednih predhodnic). Branje vhodnih podatkov in priprava seznamov predhodnic in naslednic vzameta še $O(n+m)$ časa. Časovna zahtevnost celotnega postopka je tako $O(mn+n+m) = O(mn)$.

R2003.3.5 Knjižnica

N: 541

Naloga je podobna znanemu problemu polnjenja nahrbtnika, le da imamo tu več polic (kot da bi imeli več nahrbtnikov); iskanje rešitve nam olajša zahteva, da moramo pri razporejanju knjig na police spoštovati prvotni vrstni red (po datumu izida). Nalogo bomo rešili z dinamičnim programiranjem, s podobnim algoritmom kot pri nahrbtniku. Tam razmišljamo o tem, kaj storiti, če pride nov predmet (ali ga vzeti ali ne), tu pa poleg tega razmišljamo še o možnosti, če pride nova polica.

Koristno si je zastaviti podprobleme takšne oblike: $f(n,m,g)$ naj bo vrednost najboljše take rešitve za prvih n knjig in prvih m polic, ki ima na zadnji polici zasedenega g ali manj prostora. Potem je $f(0,m,g) = 0$ (za vsak m in g), za $n \geq 1$ pa je

$$f(n,m,g) = \max\{\quad \begin{array}{l} f(n-1,m,g), \\ (c_n + f(n-1,m,g-d_n)) \text{ [če } g \geq d_n\text{]}, \\ (c_n + f(n-1,m-1,d)) \text{ [če } m > 1 \text{ in } g \geq d_n\text{]} \quad \}. \end{array}$$

Pri tem moramo vzeti $c_n = 1$, če nas zanima le največ knjig, ali pa $c_n = d_n$, če nas zanima največja skupna debelina knjig. Prva od treh vrednosti,

katerih max iščemo, predstavlja možnost, da n-te knjige sploh ne damo na nobeno polico; druga predstavlja možnost, da n-to knjigo damo na m-to polico, pri čemer je bila na tej polici od prej mogoče že kakšna druga knjiga; tretja vrednost pa predstavlja možnost, da damo n-to knjigo kot prvo na m-to polico: vse knjige pred njo so torej šle na prvih $m-1$ polic (ali pa jih sploh nismo dali na nobeno polico). Rešitev našega prvotnega problema, torej rezultat, ki ga pravzaprav iščemo, pa je $f(N, M, d)$ — kar ustreza podproblemu, ki ima na razpolago vse knjige, vse police in nobene posebne omejitve glede tega, koliko prostora sme biti največ zasedenega na zadnji polici.

Gornja formula deluje tudi v primeru, ko je polic več kot knjig ($m > n$); ne more sicer najti razporedov, pri katerih je zadnjih nekaj polic praznih, ampak saj pri takih bi lahko knjige vedno premestili tako, da bi bilo praznih le prvih nekaj polic, vse od tam naprej pa bi vsebovale vsaj eno knjigo: do takih rešitev lahko gornja formula pride, če začne pri $f(0, m, g)$ za nek $m \geq 1$.

Za vsak konkreten par (n, m) je funkcija $f_{n,m}(g) := f(n, m, g)$ „stopničasta“, torej nepadajoča (če dovolimo bolj zasedeno zadnjo polico, lahko spravimo nanjo vsaj toliko knjig kot prej) in odsekoma konstantna (ko se spremeni, se spremeni zato, ker je mogoče spraviti v regal vsaj eno knjigo več kot prej, torej se vrednost f poveča vsaj za 1; dokler pa števila knjig ni mogoče povečati, ostaja f nespremenjena); torej jo lahko predstavimo tako, da navedemo g-je, pri katerih se njena vrednost poveča, in za vsak g navedemo še novo vrednost funkcije. Operacijo max med funkcijami lahko izvedemo z zlivanjem teh seznamov. Pri naši nalogi pa je stvar še toliko lažja, ker so možne vrednosti g le cela števila $\{0, \ldots, d\}$ (in $d \leq 100$), tako da lahko predstavimo vsako tako funkcijo kar s tabelo sto elementov.

Kot ponavadi pri dinamičnem programiranju je tudi tu zelo pomembno, da si rešitve podproblemov, torej funkcije $f_{n,m}(g)$ za razne pare (n, m), ko jih enkrat izračunamo, nekje zapomnimo, ker bodo kasneje prišle še prav in bi bilo škoda, če bi jih morali takrat računati ponovno. Iz gornje formule lahko opazimo, da pri izračunu $f_{n,m}$ potrebujemo le $f_{n-1,m}$ in $f_{n-1,m-1}$; zato je koristno računati te funkcije po naraščajočih n in pri vsakem n-ju po naraščajočih m. Tako imamo vedno pri roki rešitve podproblemov, ki jih potrebujemo, rezultate za $n-2$ in manj knjig pa lahko sproti pozabljamo.

```
program Knjige;
  procedure Max(var a: integer; b: integer);
    begin if b > a then a := b end;
const MaxN = 1000; MaxM = 100; MaxD = 100;
var T: text;
    n, m, d, ni, mi, g, b, bb: integer;
    di: array [1..MaxN] of integer;
    f: array [1..MaxM, 0..MaxD] of integer;
begin
```

```
  Assign(T, 'knjige.in'); Reset(T); ReadLn(T, n, m, d);
  for ni := 1 to n do Read(T, di[ni]);
  Close(T);

  { 0 knjig. }
  for mi := 1 to m do for g := 0 to d do f[mi, g] := 0;

  { Dodajajmo knjige. }
  for ni := 1 to n do begin
    b := 0; { Najboljši rezultat za ni − 1 knjig in 0 polic. }
    for mi := 1 to m do begin

      { ni knjig in mi polic. Ko računamo za nek g, bomo potrebovali rezultat
        za ni − 1 knjig, mi polic in razne manjše g, tako da je koristno
        iti od večjih g proti manjšim, da si ne bomo sproti kvarili starih
        rezultatov, ki jih bomo še potrebovali. Zapomnimo si tudi najboljši
        rezultat za ni − 1 knjig in mi polic. }

      bb := f[mi, d];
      for g := d downto di[ni] do begin
        { Ena možnost je, da knjige ni sploh ne vzamemo.
          Potem je rezultat za ni knjig in mi polic enak kot za ni − 1 knjig
          in mi polic, ta pa je že v f[mi, g]. }
        { Lahko dodamo knjigo ni na polico mi, pri čemer je prej na njej
          že kakšna knjiga. Rešitve za ni − 1 knjig in mi polic
          in majhne g so trenutno še v f[mi]. }
        Max(f[mi, g], 1 + f[mi, g − di[ni]]);
        { Lahko dodamo knjigo ni na polico mi kot prvo.
          Najboljši rezultat za ni − 1 knjig in mi − 1 polic je trenutno v b. }
        Max(f[mi, g], 1 + b);
      end; {for g}
      b := bb; { Najboljši rezultat za ni − 1 knjig in mi polic. }

    end; {for mi}
  end; {for ni}

  Assign(T, 'knjige.out'); Rewrite(T); WriteLn(T, f[mi, d]); Close(T);
end. {Knjige}
```

Viri nalog za leto 2003: pošiljanje sporočil — Andraž Bežek; vplivi — Uroš Jovanovič; napadalne kraljice — Mitja Lasič; dva kupa števil — Jure Leskovec; radar — Mark Martinec; „pet čevljev merim, palcev pet“ — Mojca Miklavec; vodenje projektov — Blaž Novak; glasovanje, knjižnica — Anže Žagar; križanka, števke — Klemen Žagar; različnost nizov, smučarji — Janez Brank.

Hvala Juretu Leskovcu za implementacijo rešitve naloge s smučarji in Andreju Bauerju za pripombe k besedilu nalog. Citat pri tretji nalogi za tretjo skupino je iz 5. poglavja pete knjige *Bratov Karamazovih*.

28. državno tekmovanje v znanju računalništva (2004)

NALOGE ZA PRVO SKUPINO

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, jo obvezno tudi komentiraj in v nekaj stavkih z besedami opiši, na kakšni ideji temelji tvoja rešitev.

2004.1.1 SMS

Pri pisanju sporočil na prenosnem telefonu moramo vsakič, ko dve sosednji črki pripadata isti tipki prenosnega telefona, malce počakati. Če želimo na primer natipkati besedo „bacil“, moramo pritisniti tipke R: 596

2 2 (za b) 2 (za a) 2 2 2 (za c) 4 4 4 (za i) 5 5 5 (za l),

kar pomeni, da moramo dvakrat malce počakati — preden natipkamo „a“ in preden natipkamo „c“.

Napiši program, ki za prebrani stavek izračuna, kolikokrat bomo pri pisanju sporočila na prenosnem telefonu morali počakati. Predpostavi, da stavek vsebuje le male črke angleške abecede in presledke.

Razporeditev črk po tipkah:

1	2	3	4	5	6	7	8	9
(presledek)	abc	def	ghi	jkl	mno	pqrs	tuv	wxyz

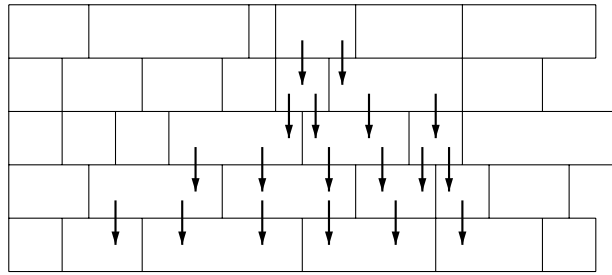
2004.1.2 Ploščice

Po tleh smo v nekaj vrstic položili ploščice, ki so vse enako visoke, vendar različno široke. Ploščice so zložene stikoma (ena ob drugi) tako, da v posamezni vrstici ni „lukenj“ in da so levi robovi vrstic lepo poravnani. R: 596

Po ploščicah se smemo premikati na naslednji način: začnemo v eni od ploščic prve vrstice; nato pa v vsakem koraku stopimo s trenutne ploščice na eno od tistih, ki so v naslednji vrstici tik pot trenutno ploščico in imajo z njo skupen nek del stranice (samo oglišče ni dovolj!). S premikanjem končamo, ko pridemo v zadnjo vrstico. (Glej primer na sliki na str. 580.)

Napiši program, ki ugotovi, katera je najbolj leva in katera najbolj desna ploščica v zadnji vrstici, ki ju lahko pri opisanem načinu premikanja dosežemo. Predpostavi, da so ti na voljo naslednji podprogrami:

- **function** StVrstic: integer; **external**; — Vrne število vrstic.



Ilustracija k nalogi 2004.1.2. Na sliki je Primer razporeditve ploščic v pet vrstic. Puščice kažejo, kako se lahko premikamo med ploščicami, če začnemo pri četrti ploščici prve vrstice. V zadnji (peti) vrstici so dosegljive ploščice od vključno druge do vključno pete.

- **function** StPloscic(StVrstice: integer): integer; **external**;
  Vrne število ploščic v dani vrstici (večje ali enako 1). Vrstice so oštevilčene od 1 do StVrstic.

- **function** SirinaPloscice(StVrstice, StPloscice: integer): integer; **external**;
  Vrne širino dane ploščice (v centimetrih). Ploščice so v vsaki vrstici oštevilčene od leve proti desni (skrajno leva ploščica ima številko 1, skrajno desna pa ima številko StPloscic(StVrstice)).

- **function** ZacetnaPloscica: integer; **external**;
  Številka ploščice (v prvi vrstici), v kateri se začne naše gibanje. To je število med vključno 1 in vključno StPloscic(1).

Predpostaviš lahko, da so ploščice razporejene tako, da bo v zadnji vrstici dosegljiva vsaj ena ploščica. Skupna širina vseh ploščic ne bo presegla največje vrednosti, ki jo še lahko hrani tip integer oz. **int**.

Še deklaracije v C/C++:

```
extern int StVrstic();
extern int StPloscic(int StVrstice);
extern int SirinaPloscice(int StVrstice, int StPloscice);
extern int ZacetnaPloscica();
```

2004.1.3 Ruleta

R: 597 Ruleta je igra na srečo, pri kateri se žreba številke od 0 do 36 (vržemo kroglico in pogledamo, pri kateri številki se ustavi). Igralci pred žrebanjem stavijo in če njihove stave zadenejo, se jim stavljeni znesek večkratno povrne.

Na voljo imaš naslednje funkcije, ki ti sporočajo podatke o stavah:

- **function** StStav: integer; **external**; — Vrne število stav.

- **function** AliZadene(Stava, Stevilka: integer): boolean; **external**;
  Ali dana stava zadene, če je izžrebana številka Stevilka? Stave so oštevilčene od 1 do StStav.

- **function** VisinaStave(Stava: integer): integer; **external**;
  Vrne znesek, ki ga je igralec stavil pri tej stavi.

- **function** Kolicnik(Stava: integer): integer; **external**;
  Vrne količnik, ki pove, kolikokratno se stavljeni znesek povrne, če stava zadene. V praksi so ti količniki odvisni od tega, kolikšna je verjetnost zadetka. (Na primer: če je stava taka, da zadene le pri eni številki, je ta količnik 36; če zadene, igralnica vrne igralcu znesek, ki ga je ta stavil, in mu za povrhu plača še 35-krat tolikšno vsoto. Če pa stava ne zadene, igralnica položeni denar pobere in igralec ne dobi ničesar. Pri drugačnih stavah je količnik drugačen; primer: stava, ki zadene pri vsaki lihi številki, ima količnik 2. Kakorkoli že, tebi se s tem ni treba ukvarjati, ampak predpostavi, da je funkcija Kolicnik že napisana.)

Pohlepni lastniki igralnice od tebe želijo, da **napišeš program**, ki pregleda podatke o stavah in ugotovi, katera številka mora biti izžrebana, da bo igralnica izplačala čim manj denarja. Če je več enako dobrih rešitev, je dovolj, če izpišeš samo eno od njih (katerokoli).

Še deklaracije v C/C++:

```
extern int StStav();
extern int AliZadene(int Stava, int Stevilka);
extern int VisinaStave(int Stava);
extern int Kolicnik(int Stava);
```

2004.1.4 Tekoče stopnice

Končno! V Ljubljano prihaja podzemna železnica. Na nekaterih vmesnih postajah, kjer bo vstopalo in izstopalo manj ljudi, se bolj kot dvoje tekočih stopnic splača zgraditi ene same, ki bodo smer prilagajale potnikom (razen v konicah, ko bodo nekateri morali po stopnicah peš, medtem ko bodo stopnice vozile v nasprotno smer). R: 598

Prosili so nas za pomoč pri programu za krmiljenje stopnic, ki se morajo zagnati (če so prej mirovale) vedno, ko kdo stopi na enega od obeh senzorjev pred stopnicami (na vsaki strani stopnic je po en senzor, ki ti zna povedati, če kdo trenutno stoji na njem ali ne; ne vemo pa, v katero smer se tisti potnik giblje), in ustaviti, ko smo prepričani, da na stopnicah ne stoji noben potnik več (samo predstavljaj si, da bi te stopnice pripeljale do sredine, se ustavile in

takoj spet obrnile v drugo smer, ti pa bi moral sredi hitenja v šolo po minuti prevažanja nazadnje še peš po navadnih stopnicah).

Pomagaj si z naslednjimi deklaracijami:

```
const CasVoznje = ...; { Toliko sekund je treba, da se pripelje potnik po stopnicah
                         od enega konca stopnišča do drugega. }

type SenzorT = (Zgoraj, Spodaj);
     SmerT = (Gor, Dol, Ustavi);

procedure PozeniStopnice(Smer: SmerT); external;
  { Ta podprogram pokliči, ko je treba pognati ali ustaviti stopnice. }
function PotnikNaSenzorju(Senzor: SenzorT): boolean; external;
  { Ta podprogram ti pove, ali na danem senzorju trenutno kdo stoji. }
```

Ti pa **napiši** naslednji **podprogram**:

```
procedure EnkratNaMilisekundo;
  { Sistem ga pokliče enkrat na milisekundo (tisočkrat na sekundo). }
```

Predpostavljaš lahko, da stopnice na začetku mirujejo. V svoji rešitvi lahko definiraš dodatne globalne spremenljivke, ki jih še potrebuješ, in jim tudi predpišeš začetne vrednosti.

Še deklaracije v C/C++:

```
const int CasVoznje = ...;
typedef enum { Zgoraj, Spodaj } SenzorT;
typedef enum { Gor, Dol, Ustavi } SmerT;
extern void PozeniStopnice(SmerT Smer);
extern bool PotnikNaSenzorju(SenzorT Senzor);

void EnkratNaMilisekundo();
```

NALOGE ZA DRUGO SKUPINO

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, jo obvezno tudi komentiraj in v nekaj stavkih z besedami opiši, na kakšni ideji temelji tvoja rešitev.

2004.2.1 Naraščajoča števila

R: 599

Včasih lahko pri nekaterih nalogah naletimo na podproblem, kako zamenjati števila v zaporedju, tako da imamo na koncu sama različna števila.

Opiši postopek, ki za izbran N prebere N celih števil, večjih od 0 in manjših ali enakih N (običajno niso vsa različna, ampak se nekatera števila ponavljajo). Za vsako od teh števil naj postopek izpiše najmanjše enako ali večje število, ki pa še ni bilo izpisano.

Pri tem pa izpisano število ne sme biti večje od N. Če so bila vsa števila od trenutnega vhodnega števila do N že izpisana, izpiši najmanjše število, ki je večje od 0 in še ni bilo izpisano.

Če upoštevaš vsa navodila, boš na koncu izpisal vsa števila od 1 do N, vsako natanko po enkrat. Za boljše razumevanje glej spodnja primera.

Pazi pa tudi na to, da bo tvoj postopek čim hitrejši.

Primer vhodnega zaporedja ($N = 10$):	6	6	6	6	5	3	3	1	1	1
Pripadajoče izhodno zaporedje:	6	7	8	9	5	3	4	1	2	10

Drugo število vhodnega zaporedja je 6, torej moramo na drugem mestu izhodnega števila izpisati število, večje ali enako 6. Ker smo 6 že izpisali, je najmanjše tako število 7.

Še en primer ($N = 5$):	5	5	5	5	5
Pripadajoče izhodno zaporedje:	5	1	2	3	4

Na drugem mestu bi morali izpisati število, večje ali enako 5; ker je $N = 5$, je edino tako število 5, ki pa je bilo že izpisano, zato izpišemo najmanjše še neizpisano število (v tem primeru torej 1).

2004.2.2 Topovske krogle

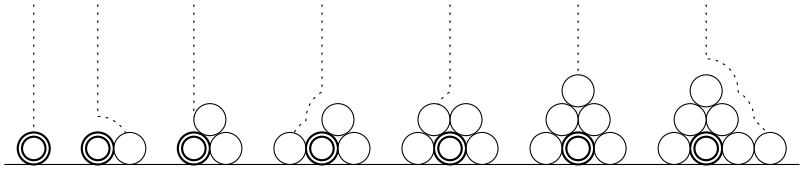
R: 600

Cesarsko topništvo se je dolgo ukvarjalo s problemom, kako polniti zaboje s topovskimi kroglami, saj so želeli ustaviti polnjenje zabojev prej, preden so začele težke krogle padati čez rob zabojev. Le kdo jih bo še enkrat nalagal; ni časa ne denarja. Pametni cesarski uradniki so ugotovili, da je to odvisno od kalibra topovske krogle (premer krogle), dimenzij zaboja in položaja polnilne odprtine, zato so predpisali, da mora biti velikost zaboja, ki ga dostavijo topovske garnizije, nek večkratnik kalibra topovske krogle. To so naredili, za ostalo jim je pa zmanjkalo pameti. Na pomoč so poklicali našega velikega matematika Jurija Vego, ki jim je v kratkem času narisal kopico čudnih slik in na koncu napisal še algoritem za izračun tega, katera krogla po vrsti po padla čez rob.

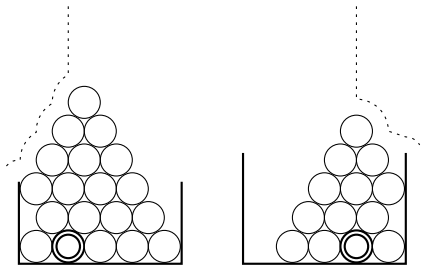
Žal so čas in vojne opravile svoje in tako danes ne moremo več najti tega algoritma in te naprošamo, da ga napišeš ti — **napiši** torej **algoritem ali**

podprogram, ki bo pri danih podatkih o višini, širini in položaju polnilne šobe izpisal, katera krogla po vrsti bo padla čez rob zaboja. Pri tem upoštevaj naslednje:

- Zaboj je ravno dovolj širok za s krogel in dovolj visok za v plasti krogel. Pri tem sta s in v neki celi števili (večji od 0).
- Položaj polnilne odprtine je določen s celim številom p (med vključno 1 in vključno s), ki pove, da je odprtina naravnost nad p-tim prostorom za kroglo (z drugimi besedami, ko pade prva krogla skozi odprtino v prazen zaboj, pristane tako, da je levo od nje prostora za natanko $p-1$ krogel, desno pa za $s-p$ krogel).
- Zaboj nima tretje dimenzije oz. je v tej dimenziji dovolj velik ravno za en premer krogle. Zato lahko pri zlaganju krogel tretjo dimenzijo zanemarimo.
- Če krogla pade natančno na vrh druge in ima prostor na obeh straneh, vedno pade na desno stran. Krogle so brez inercije — nikoli se ne gibljejo vodoravno ali navzgor.



Primer, kako bi se razporedilo prvih nekaj krogel (ob predpostavki, da se ne zaletavajo v stene zaboja). Prva krogla je označena z dvojnim robom. Črtkane črtice označujejo, kako se je gibala zadnja krogla.



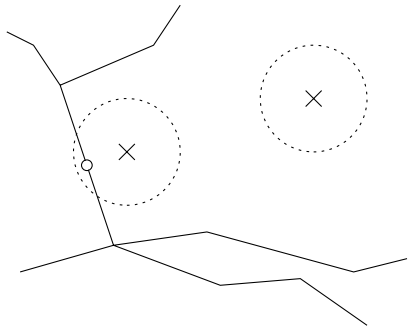
Črtkane črtice kažejo, kako se giblje tista krogla, ki prva pade čez rob.

Levo je primer za $s = 5$, $p = 2$, $v = 3$; devetnajst krogel gre v zaboj, dvajseta pade čez rob.

Desno je primer za $s = 5$, $p = 4$, $v = 4$; trinajst krogel gre v zaboj, štirinajsta pade čez rob.

2004.2.3 GPS

GPS je sistem lociranja v naravi z natančnostjo (radijem) okoli 16 metrov. R: 602
Omejena natančnost nas lahko pri orientaciji v naravi oziroma še bolj na cesti precej moti. Natančnost se v resničnem svetu lahko spreminja v odvisnosti od vidnih satelitov in kakovosti signala. Ker hočemo natančnost umetno povečati, lahko primerjamo izračunani položaj z zemljevidom in popravimo položaj tako, da ga virtualno prestavimo na najbližjo točko na cesti oziroma poti znanega zemljevida (če seveda je kakšna cesta v radiju natančnosti).



Slika prikazuje nekaj cest in dve meritvi, označeni z ×. Okoli vsake meritve je črtkana krožnica, katere radij ponazarja natančnost meritve.

Pri levi meritvi je naš pravi položaj najbrž najbližja točka na daljici levo od meritve (označena s krožcem).

Pri desni meritvi pa smo predaleč od vseh poti in zato ne bi bilo pametno umetno popravljati meritve, ker bi s tem najbrž samo povečali napako.

Mi imamo na voljo nekaj podobnega; imamo funkcijo

```
function GPS(var X, Y: integer): integer;
int GPS(int* X, int* Y);
```

ki nam vrne natančnost meritve v metrih (radij), v parametrih pa vrne izmerjeni položaj v metrih od nekega izhodišča. Imamo tudi podatke o poteh in cestah: njihov potek je opisan z daljicami, za vsako daljico pa poznamo koordinate njenih dveh krajišč.

```
type Tocka = record x, y: integer end;
var Zemljevid: array [1..N, 1..2] of Tocka;
typedef struct { int x, y; } Tocka;
Tocka Zemljevid[N][2];
```

Na voljo imamo tudi funkcijo, ki zna za dano daljico (od A do B) in dano točko (recimo T) ugotoviti, katera je njej najbližja točka na daljici; poleg tega tudi vrne razdaljo med T in to najbližjo točko.

```
function NajblizjaTocka(A, B, T: Tocka; var Najblizja: Tocka): real;
float NajblizjaTocka(Tocka A, Tocka B, Tocka T, Tocka* Najblizja);
```

Opiši postopek, ki bo z uporabo teh funkcij in podatkov izboljšal natančnost izmerjenega položaja in vrnil novo izračunani položaj. Postopek naj torej prebere eno točko (izmerjeni položaj) in izpiše odgovor (ali najbližjo točko na kateri od daljic ali pa naj ugotovi, da nobena daljica ne leži dovolj blizu izmerjenega položaja).

Ker se tabela daljic ne bo pogosto spreminjala, lahko pred prvo uporabo svojega postopka podatke o daljicah tudi kako preurediš ali jih drugače organiziraš, da bo potem tvoj postopek lahko čim hitreje ugotovil, katera točka na daljicah je najbližja trenutni meritvi (po možnosti tako, da mu ne bo treba vsakič preiskati vseh daljic!). Predpostaviti smeš, da se daljice med seboj ne križajo, se pa seveda lahko stikajo.

2004.2.4 Skrajšanke

R: 603

Imamo skupino besed, iz katerih bi radi na vse možne načine sestavili skrajšanke. Skrajšanke (akronimi) so besede, ki jih lahko sestavimo tako, da vzamemo prvih nekaj črk vsake besede iz dane skupine besed in jih staknemo skupaj. Pri tem lahko poljubno spreminjamo vrstni red besed ter število prvih nekaj črk, ki jih bomo uporabili iz vsake besede. Vsako besedo smemo uporabiti samo enkrat.

Primer skrajšanke: RADAR = RAdio Detection And Ranging.

Napiši podprogram, ki bo iz danih N besed izpisal vse možne skrajšanke. Pri tem mora program zagotoviti:

- da bodo besede predstavljene v vseh mogočih vrstnih redih (primer za tri besede: ena, dve, tri; ena, tri, dve; dve, ena, tri; dve, tri, ena; tri, ena, dve; tri, dve, ena);
- da bo pri vsakem vrstnem redu zastopana vsaka beseda z vsemi možnimi začetki, od 0 pa vse do prvih M črk besede (primer: če je $M = 5$, moramo besedo `REGISTRACIJA` v skrajšankah predstaviti z vsemi naslednjimi začetki: „“ (prazen niz), „`R`“, „`RE`“, „`REG`“, „`REGI`“ in „`REGIS`“).

Lahko se zgodi, da ista skrajšanka nastane na več načinov (glej primer spodaj) — v tem primeru naj jo tvoj podprogram tudi po večkrat izpiše. Ne izpiše pa naj skrajšank, ki so prazni nizi.

Pomagaj si z naslednjimi deklaracijami:

```
const N = ...;
type TabelaT = array [1..N] of string;

procedure IzpisiVseSkrajsanke(M: integer; Besede: TabelaT);
```

Primer na str. 587 kaže, kaj dobimo pri skupini $N = 3$ besed, `REGISTRACIJA`, `NADZOR` in `SISTEM`, če želimo vse skrajšanke, kjer bo vsaka beseda prikazana z 0 pa do največ tremi črkami ($M = 3$).

(iz *registracija nadzor sistem*)

s si sis
n ns nsi nsis
na nas nasi nasis
nad nads nadsi nadsis

r rs rsi rsis
rn rns rnsi rnsis
rna rnas rnasi rnasis
rnad rnads rnadsi rnadsis

re res resi resis
ren rens rensi rensis
rena renas renasi renasis
renad renads renadsi renadsis

reg regs regsi regsis
regn regns regnsi regnsis
regna regnas regnasi regnasis
regnad regnads regnadsi regnadsis

(iz *registracija sistem nadzor*)

n na nad
s sn sna snad
si sin sina sinad
sis sisn sisna sisnad

r rn rna rnad
rs rsn rsna rsnad
rsi rsin rsina rsinad
rsis rsisn rsisna rsisnad

re ren rena renad
res resn resna resnad
resi resin resina resinad
resis resisn resisna resisnad

reg regn regna regnad
regs regsn regsna regsnad
regsi regsin regsina regsinad
regsis regsisn regsisna regsisnad

(iz *nadzor registracija sistem*)

s si sis
r rs rsi rsis
re res resi resis
reg regs regsi regsis
. . . . . .

(iz *nadzor sistem registracija*)

r re reg
s sr sre sreg
si sir sire sireg
sis sisr sisre sisreg
. . . . . .

(iz *sistem registracija nadzor*)

n na nad
r rn rna rnad
re ren rena renad
reg regn regna regnad
. . . . . .

(iz *sistem nadzor registracija*)

r re reg
n nr nre nreg
na nar nare nareg
nad nadr nadre nadreg
. . . . . .

Primer skrajšank pri nalogi 2004.2.4.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

[Na začetku tekmovanja smo tekmovalcem najprej razdelili naslednja navodila. Nekaj minut kasneje so dobili tudi besedilo nalog, za reševanje pa so imeli približno tri ure časa. — *Op. ur.*]

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*.`in` in izpisujejo svoje rezultate v *imenaloge*.`out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih

in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil mapo s svojim uporabniškim imenom ter navodili, ki jih pravkar prebiraš. Ko boš sedel pred računalnik, boš dobil nadaljnja navodila za prijavo v sistem.

Na vsakem računalniku imaš na voljo enoto (disk) `U:`, na kateri lahko kreiraš svoje datoteke (datoteke, ki so tam že od prej, pusti pri miru). Programi naj bodo napisani v programskem jeziku Pascal, C ali C++, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal, GNU C/C++ in GCJ. Za delo lahko uporabiš `FP` oz. `ppc386` (FreePascal), `TURBO` (Turbo Pascal) `GCC/G++` (GNU C/C++ — command line compiler), `TC` (Turbo C) in Java 2 SDK.

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `RTK.EXE`, ki ga lahko uporabiš za preverjanje svojih rešitev. Tukaj si lahko tudi ogledaš anonimizirane rezultate ostalih tekmovalcev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka, brez šumnikov).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

`rtk` *imenaloge*`.pas`
`rtk` *imenaloge*`.c`
`rtk` *imenaloge*`.cpp`
`rtk` *ImeNaloge*`.java`

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Na spletni strani boš dobil za vsak testni primer obvestilo o tem, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z drugimi datotekami kot z vhodno in izhodno. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov, prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi od 0 do 10 točk (praviloma 10, če je izpisal popolnoma pravilen odgovor, sicer pa 0; izjema je 2. naloga, kjer dobijo boljše rešitve več točk kot slabše), nato pa se te točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi

$\max\{0, M - 3(N-1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

Skupno število točk tekmovalca je vsota po vseh nalogah. Tekmovalce razvrstimo po skupnem številu točk.

Vsak tekmovalec se mora sam zase odločiti o tem, katerim nalogam bo posvetil svoj čas, v kakšnem vrstnem redu jih bo reševal in podobno. Verjetno je priporočljivo najprej reševati lažje naloge.

Poskusna naloga (ne šteje k tekmovanju)

`poskus.in`, `poskus.out`

Napiši program, ki iz vhodne datoteke prebere eno celo število (le-to je v prvi vrstici, okoli njega ni nobenih dodatnih presledkov ipd.) in izpiše njegov desetkratnik v izhodno datoteko.

Primer vhodne datoteke:	Ustrezna izhodna datoteka:
`123`	`1230`

Primer rešitve:

```
program PoskusnaNaloga;
var T: text; i: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * i); Close(T);
end.
```

```
#include <stdio.h>
int main() {
  FILE *f = fopen("poskus.in", "rt");
  int i; fscanf(f, "%d", &i); fclose(f);
  f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * i);
  fclose(f); return 0;
}
```

```
#include <fstream.h>
int main() {
  ifstream ifs("poskus.in"); int i; ifs >> i;
  ofstream ofs("poskus.out"); ofs << 10 * i;
  return 0;
}
```

```
import java.io.*;
public class Poskus {
  public static void main (String[] args) {
    try {
      StreamTokenizer st = new StreamTokenizer(new FileReader("poskus.in"));
      st.nextToken(); int i = (int) st.nval;
      PrintWriter os = new PrintWriter(new FileOutputStream("poskus.out"));
      os.println(10 * i); os.close();
```

```
    } catch (Exception e) { }
  }
}
```

NALOGE ZA TRETJO SKUPINO

2004.3.1 Otoki

`otoki.in`, `otoki.out`

R: 605 Če bi gladina svetovnih morij in oceanov narasla, bi se lahko kakšen otok popolnoma potopil in bi se tako število otokov na svetu zmanjšalo za ena. Po drugi strani pa bi lahko pri kakšnem otoku voda zalila le nižje ležeče dele, hribi pa bi še vedno štrleli ven in bi tako iz enega nastalo več samostojnih otokov; tako bi se število otokov na svetu povečalo. Podobno je, če se gladina oceanov zniža — kakšen otok lahko na novo pogleda iz vode, lahko pa se več otokov združi v enega samega, ker se plitvine med njimi izsušijo. Tako se torej, če spreminjamo gladino oceanov, število otokov zdaj povečuje, zdaj zmanjšuje. Nas pa zanima, kakšno je največje število otokov, ki bi ga lahko na svetu imeli, če bi mogli primerno izbrati gladino oceanov. (Za potrebe te naloge štejemo tudi celine kot velike otoke.)

Da bo naloga lažja, predpostavimo, da je Zemlja ploščata, ne pa okrogla, in da je njeno površje kot karirasta mreža, v kateri je višina površja znotraj vsake celice konstantna. Naj bo $v(x, y)$ višina celice na preseku vrstice y in stolpca x. Če je gladina oceanov g, si mislimo, da so vse celice (x, y), za katere je $v(x, y) < g$, potopljene pod vodo, vse ostale pa so kopne. Pri tej nalogi torej ne bomo kompliciral z jezeri, depresijami, rečnimi in jezerskimi otoki ipd. *Otok* definirajmo kot vsako tako skupino kopnih celic, ki je povezana (torej se da priti od poljubne celice otoka do poljubne druge celice otoka, pri tem pa ves čas hoditi po samih kopnih celicah in iti vedno s trenutne celice na eno od njenih štirih sosed, ki imajo z njo skupnega enega od robov) in ki ji ne moremo dodati nobene druge trenutno kopne celice, ne da bi ta pogoj prenehal veljati. (Glej primer spodaj.)

Napiši program, ki pri danih podatkih o višinah ugotovi, katero je največje možno število otokov, ki jih je mogoče dobiti s primernim izborom gladine oceanov g.

Vhodna datoteka: v prvi vrstici sta najprej števili h in w, ki povesta višino oz. širino kariraste mreže. To sta celi števili, velja $1 \le h \le 100$, $1 \le w \le 100$. Sledi h vrstic, v vsaki je po w števil, ki povedo višine celic: $v(1, y), v(2, y), \ldots, v(w, y)$. Višine so cela števila, zanje velja $0 \le v(x, y) \le 1\,000\,000\,000$.

Izhodna datoteka: vanjo naj tvoj program izpiše največje možno število otokov, ki ga je mogoče pri dani vhodni datoteki dobiti, če primerno izberemo gladino oceanov g.

Primer vhodne datoteke:

```
5 10
6 6 8 6 3 4 2 3 6 7
3 7 9 7 3 3 3 2 6 6
3 6 7 7 2 2 6 5 5 6
5 5 5 2 2 5 5 8 5 5
5 6 5 1 0 3 4 4 4 3
```

Tu je mogoče dobiti pet otokov (pri $g = 6$):

```
[6 6 8 9] 3 4 2 3 [6 7]
 3 [7 9 7] 3 3 3 2 [6 6]
 3 [6 7 7] 2 2 [6] 5 5 [6]
 5 5 5 2 2 5 5 [8] 5 5
 5 [6] 5 1 0 3 4 4 4 3
```

Pripadajoča izhodna datoteka:

```
5
```

2004.3.2 SBN

`sbn.in`, `sbn.out`

Mislimo si, da imamo nek zelo preprost računalnik. Njegov procesor ima 256 registrov, ki jih označimo z `r1`, `r2`, ..., `r256`. Vsak register lahko hrani poljubno predznačeno 32-bitno celo število (kot tip integer oz. **int**). Poleg tega je v računalniku še bralni pomnilnik (ROM), v katerem je shranjeno zaporedje ukazov, ki ga bo procesor izvajal. Običajnega bralno-pisalnega pomnilnika (RAM) ali kakšnih zunanjih enot pa ta računalnik nima.

R: 608

Procesor podpira eno samo inštrukcijo: „odštej in skoči, če je rezultat negativen“ (SBN — *subtract and branch if negative*). Vsak ukaz je takšne oblike:

Oznaka: SBN a, b, c, CiljnaOznaka

Procesor pri takem ukazu izračuna razliko b − c in jo shrani v a; nato pa, če je ta razlika manjša od 0, skoči na ukaz z oznako CiljnaOznaka (drugače pa se izvajanje nadaljuje pri naslednjem ukazu v zaporedju):

Oznaka: a := b − c; **if** a < 0 **then goto** CiljnaOznaka;

Operand a mora biti ime nekega registra, b in c pa sta lahko imeni registrov ali pa celoštevilski konstanti. Oznaka mora biti enolična (dva ukaza ne smeta imeti iste oznake), drugače pa je to lahko poljuben niz, sestavljen iz črk, števk in znakov „_“, ki se ne začne na števko in je dolg največ 20 znakov. Pri imenih oznak ne razlikujemo med velikimi in malimi črkami. Oznaka pred ukazom (z dvopičjem vred) lahko tudi manjka, le da se potem nanj pač ne bo dalo skakati. Tudi operand CiljnaOznaka (skupaj z vejico pred njim) lahko manjka; v tem primeru se bo izvajanje nadaljevalo pri naslednjem ukazu v zaporedju, ne glede na to, ali je bil rezultat odštevanja negativen ali ne.

Procesor začne z izvajanjem pri prvem ukazu v zaporedju, ustavi pa se, če je ravnokar izvedel zadnji ukaz v zaporedju in mu po njem ni bilo treba izvesti skoka (ker rezultat odštevanja ni bil manjši od 0 ali pa ker manjka četrti operand s ciljno oznako).

Izkaže se, da lahko z inštrukcijo SBN emuliramo več ali manj vse aritmetične in logične operacije, kakršne podpirajo običajni procesorji.

Napiši program, ki iz *vhodne datoteke* prebere neko naravno število n (zanj velja $2 \le n \le 100$) in v *izhodno datoteko* izpiše zaporedje ukazov SBN, ki uredi vrednosti v registrih r1, r2, ..., rn. Ne glede na to, kakšne so začetne vrednosti teh registrov, morajo biti ob koncu izvajanja tvojega zaporedja ukazov v teh registrih enake vrednosti, le prerazporejene tako, da velja r1 $\le$ r2 $\le \ldots \le$ rn; končne vrednosti ostalih registrov so lahko poljubne. Zaporedje ukazov sme biti dolgo največ 100 000 ukazov in se ob izvajanju ne glede na začetne vrednosti registrov ne sme izvajati več kot 1 000 000 korakov. Začetne vrednosti registrov r1, ..., rn so med vključno $-1\,000\,000$ in vključno $+1\,000\,000$, začetna vrednost registrov r($n+1$), ..., r256 pa je 2004.

Namig: zaporedje n vrednosti lahko urediš na primer tako, da najprej poiščeš najmanjšo med njimi in jo postaviš na prvo mesto; nato poiščeš najmanjšo med preostalimi in jo postaviš na drugo mesto; in tako naprej.

Točkovanje: pri vsakem testnem primeru lahko dobiš od 0 do 10 točk. Če zaporedje inštrukcij, ki ga tvoj program izpiše, ne ustreza zgornjim zahtevam in omejitvam, dobiš 0 točk, drugače pa je število točk odvisno od števila inštrukcij v tvojem zaporedju (recimo mu L):

Če je...	...dobiš toliko točk:
$L \le 3n$	10
$3n < L \le 6n$	9
$6n < L \le n^2/2$	8
$n^2/2 < L \le 3n^2/2$	7
$3n^2/2 < L \le 3n^2$	6
$3n^2 < L$	5

Spodaj je **primer** zaporedja ukazov, ki rešuje malo drugačen problem: v registru r6 izračuna povprečje vrednosti registrov r1, ..., r5, zaokroženo na najbližje celo število (pri tem predpostavi, da njihove vrednosti niso negativne in tudi ne prevelike).

```
                SBN r7, 0, r1
                SBN r7, r7, r2
                SBN r7, r7, r3
                SBN r7, r7, r4
                SBN r7, r7, r5
                SBN r6, r6, r6
                SBN r7, 0, r7
ZacetekZanke:   SBN r7, r7, 5, KonecZanke
                SBN r6, r6, 1, ZacetekZanke
KonecZanke:     SBN r7, r7, -2, ZaokroziDol
                SBN r6, r6, 1
ZaokroziDol:    SBN r6, 0, r6
```

2004.3.3 Kako so Butalci kupovali pamet pamet.in, pamet.out

R: 613 Ko so se Butalci odpravili v Tuje mesto po pamet, so najprej tja poslali butalskega policaja, da bi jim poročal o cenah in bi tako lahko izbrali najugodnejšo ponudbo. Brez tvoje pomoči pa seveda ne bo šlo.

Jim znaš **napisati program**, ki ugotovi najmanjši znesek, za katerega lahko kupijo N kosov pameti, če za vsakega od M trgovcev v mestu veš, koliko kosov pameti ima na voljo in koliko vsak kos pameti stane?

Vhodna datoteka. V prvi vrstici sta dve števili: N ($1 \le N \le 25$ — število Butalcev, ki potrebujejo novo pamet) in M ($1 \le M \le 500\,000$ — število trgovcev), v vrsticah 2 do $M+1$ pa najprej število kosov pameti K_i, ki jih ima na voljo i-ti trgovec, čemur sledi K_i nenegativnih celih števil $c_{i,j}$ s ceno pameti za vsak naslednji kupljen kos (torej: če pri njem kupiš en kos, plačaš $c_{i,1}$; če kupiš dva kosa, plačaš $c_{i,1} + c_{i,2}$ in tako naprej). Skupna vsota cen v vhodni datoteki ne presega ene milijarde, pameti pa bo vedno za vse dovolj. Skupno število kosov pameti, ki so trenutno naprodaj (torej $N_1 + N_2 + \ldots + N_M$) je največ 1 000 000.

(*Primer:* vrstica „`5 7 3 4 0 1`“ pomeni, da je pri tem trgovcu naprodaj pet kosov pameti, pri čemer en kos velja 7 tolarjev, dva kosa 10 tolarjev, trije ali štirje kosi 14 tolarjev in pet kosov 15 tolarjev.)

V *izhodno datoteko* izpiši najnižjo možno ceno, za katero je mogoče dobiti N kosov pameti.

Primer vhodne datoteke:

```
4 5
2 10 1
4 1 0 10 10
2 5 4
2 6 2
6 11 11 11 11 0 0
```

Pripadajoča izhodna datoteka:

```
9
```

Tu bi dva kosa pameti kupili pri drugem in dva pri četrtem trgovcu.

2004.3.4 Izomorfizem

`izomorf.in`, `izomorf.out`

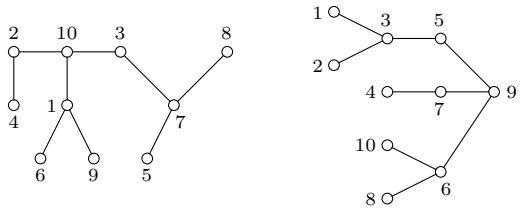
Graf je sestavljen iz množice *vozlišč* in množice *povezav*. Na sliki na str. 594 so vozlišča predstavljena s krožci, povezave pa s črtami med njimi. Vsaka povezava povezuje dve vozlišči. R: 614

Pri tej nalogi bomo imeli opravka z grafi, za katere veljajo naslednje lastnosti:

- iz vsakega vozlišča se da po povezavah priti do vsakega drugega;
- povezave ne tvorijo ciklov (na primer a–b, b–c in c–a);
- nobeno vozlišče ne nastopa v več kot treh povezavah.

Napiši program, ki prebere opisa dveh grafov s po n vozlišči in $n-1$ povezavami. Vozlišča so pri vsakem grafu oštevilčena od 1 do n. Pravimo, da sta grafa *izomorfna*, če lahko enega pretvorimo v drugega zgolj s preimenovanjem vozlišč. Tvoj program naj ugotovi, ali sta dana dva grafa izomorfna ali ne.

Vhodna datoteka: v prvi vrstici je celo število n ($1 \le n \le 1000$), ki pove, koliko vozlišč imata grafa, ki ju bo treba primerjati. Temu sledi $n-1$ vrstic,



Ilustracija k nalogi 2004.3.4.

Ta dva grafa sta izomorfna: če primerno preimenujemo vozlišča prvega grafa in jih malo premaknemo po papirju (seveda ne da bi prekinili kakšno povezavo ali dodali kakšno novo ali kaj podobnega), lahko dobimo drugi graf.

---

ki navajajo povezave prvega grafa, in še $n - 1$ vrstic, ki navajajo povezave drugega grafa. Vsaka od teh vrstic vsebuje dve števili, ločeni s presledkom — to sta številki vozlišč, ki ju povezuje ena od povezav.

Izhodna datoteka: če grafa iz vhodne datoteke nista izomorfna, naj tvoj program v izhodno datoteko ne izpiše ničesar (ustvariti pa to datoteko vendarle mora). Če pa grafa sta izomorfna, pomeni, da je mogoče vozlišča prvega grafa tako preštevilčiti, da dobimo drugi graf. V tem primeru naj tvoj program izpiše n vrstic, ki opisujejo eno takšno preštevilčenje. V vsaki vrstici naj bosta dve števili, recimo u in v (ločeni s presledkom), ki povesta, da moramo vozlišče u prvega grafa preimenovati v v.

Primer vhodne datoteke za grafa z gornje slike:

```
10
2 4
2 10
10 3
7 8
5 7
7 3
10 1
6 1
1 9
1 3
3 5
4 7
3 2
7 9
9 6
5 9
10 6
6 8
```

Ena od možnih pripadajočih izhodnih datotek:

```
8 1
5 2
7 3
3 5
10 9
4 4
2 7
1 6
6 10
9 8
```

2004.3.5 Čebelica Maja gre vasovat

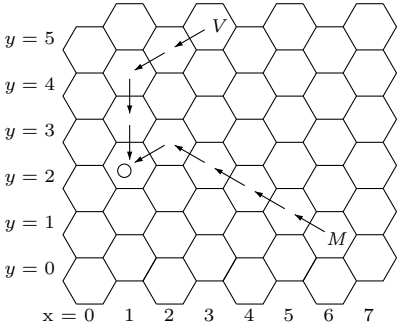
`maja.in`, `maja.out`

R: 619

Vili in čebelica Maja živita v čebeljem panju. To je satovje, ki ga sestavljajo sobice šesterokotne oblike. Ponoči čebele ne smejo leteti, zato se Maja na skrivaj odplazi vasovat k Viliju. Pri tem se plazi iz sobice v sobico, kar je zamudno, pa tudi tvegano opravilo, saj jo pri tem lahko zalotijo špeckahle. Maja pozna nekaj varnih mest v panju in jih predlaga Viliju, žal pa je ta čebelje pameti in se ne more odločiti, katerega od njih bi izbral, saj še tega ne zna prav izračunati, kako daleč je do posameznega mesta.

Podana imaš položaja Maje in Vilija ter treh predlaganih sobic. **Napiši program**, ki za vsako od njih ugotovi, kako daleč ima do te sobice Maja in kako daleč Vili. Dolžina poti se meri v številu sobic, ki jih pot vsebuje, pri tem pa se začetna sobica ne šteje. Na primer: najkrajša pot med dvema sosednjima sobicama (takima, ki imata skupno stranico) je dolga 1.

Podatki bodo podani v koordinatnem sistemu čebeljega panja, ki je predstavljen na spodnji skici. Pri 8 od 10 testnih primerov bodo vse koordinate med vključno 0 in vključno 500, pri ostalih dveh pa med 0 in 10000. Panj je dovolj velik in nima kakšnih lukenj ali manjkajočih sobic, tako da lahko predpostaviš, da vsak par celih števil (x, y) res predstavlja koordinati neke sobice.



Primer satovja. Če bi se hotela Maja in Vili sestati v sobici, označeni s krožcem, bi imela Maja do tja po najkrajši poti pet korakov, Vili pa štiri.

Vhodna datoteka. V prvi vrstici imaš koordinate Maje, v drugi pa Vilija. V naslednjih treh vrsticah so koordinate še treh drugih sobic v satovju.

V *izhodno datoteko* izpiši tri vrstice, za vsako od predlaganih treh sobic po eno. V vsaki vrstici naj bosta dve števili, ločeni s presledkom: dolžina najkrajše poti od Maje do trenutne sobice in še dolžina najkrajše poti od Vilija do trenutne sobice.

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:

`6 1`
`3 5`
`1 2`
`6 3`
`7 0`
	`5 4`
	`2 4`
	`1 7`

REŠITVE NALOG ZA PRVO SKUPINO

R2004.1.1 SMS

N: 579 V konstanti Tabela hranimo podatke o tem, katero tipko se uporablja pri posamezni črki. Vhodne podatke beremo znak po znak in pri vsakem pogledamo, če pripada isti tipki kot prejšnji znak (spremenljivka PrejTipka).

```
program SMS;
                { abcdefghijklmnopqrstuvwxyz }
const Tabela = '22233344455566677778889999';
var Znak, Tipka, PrejTipka: char;
    StCakanj: integer;
begin
  PrejTipka := 'x'; StCakanj := 0;
  while not Eoln do begin
    Read(Znak);
    if Znak in ['a'..'z'] then { Na kateri tipki je ta znak? }
      Tipka := Tabela[Ord(Znak) - Ord('a') + 1]
    else Tipka := '1'; { Najbrž je presledek. }
    if Tipka = PrejTipka then StCakanj := StCakanj + 1;
    PrejTipka := Tipka;
  end; {while}
  WriteLn(StCakanj);
end. {SMS}
```

Pomagamo si lahko s standardno funkcijo Ord(Znak), ki vrne številsko kodo znaka Znak. Črkam a, ..., z pripadajo zaporedne številske kode, zato nam razlika Ord(Znak) − Ord('a') + 1 pove položaj Znaka v abecedi in jo lahko uporabimo kot indeks v tabelo Tabela.

R2004.1.2 Ploščice

N: 579 Hranili bomo podatke o najbolj levi in najbolj desni x-koordinati, ki je še dosegljiva v trenutni vrstici (xOd, xDo), in o indeksu najbolj leve in najbolj desne dosegljive ploščice (iOd, iDo). Pri prvi vrstici je iOd = iDo = ZacetnaPloscica, x-koordinate pa dobimo s seštevanjem širin ploščic od levega roba do začetne ploščice.

Pri vsaki naslednji vrstici si zapomnimo, katere x-koordinate so dosegljive v prejšnji vrstici (xOdPrej, xDoPrej) in na podlagi tega izračunajmo vrednosti xOd, xDo, iOd, iDo. Vrednosti iOd in xOd moramo nastaviti pri prvi (najbolj levi) ploščici, ki je dosegljiva v tej vrstici (prepoznamo jo po tem, da je iOd takrat še 0, ker smo na začetku vrstice postavili na 0). Vrednosti iDo in xDo pa nastavimo pri vsaki dosegljivi ploščici, tako da bo na koncu obveljala najbolj desna ploščica, kar si pri teh dveh spremenljivkah tudi želimo.

```
program Ploscice;
var xOd, xDo, xOdPrej, xDoPrej, x, xx, y, i, iOd, iDo: integer;
begin
  x := 0;
  for i := 1 to ZacetnaPloscica − 1 do x := x + SirinaPloscice(1, i);
  xOd := x; xDo := xOd + SirinaPloscice(1, ZacetnaPloscica);
  iOd := ZacetnaPloscica; iDo := ZacetnaPloscica;
  for y := 2 to StVrstic do begin
    xOdPrej := xOd; xDoPrej := xDo; iOd := 0; x := 0;
    for i := 1 to StPloscic(y) do begin
      xx := x + SirinaPloscice(y, i);
      { Trenutna ploščica pokriva x-koordinate od x do xx. }
      if (x < xDoPrej) and (xx > xOdPrej) then begin
        iDo := i; xDo := xx;
        if iOd = 0 then begin iOd := i; xOd := x end;
      end; {if}
      x := xx;
    end; {for i}
  end; {for y}
  WriteLn('V zadnji vrstici so dosegljive ploščice ', iOd, '..', iDo, '.');
end. {Ploscice}
```

R2004.1.3 Ruleta

Pri vsaki številki se sprehodimo po vseh stavah in računajmo skupni znesek, ki bi ga morala igralnica izplačati, če bi bila izžrebana ta številka. V spremenljivki NajStevilka hranimo številko, ki zahteva najmanjši znesek (ta znesek pa hranimo v NajZnesek). Ko izračunamo znesek za naslednjo številko, ga primerjamo s spremenljivko NajZnesek in obdržimo manjši znesek.

N: 580

```
program Pohlep;
var NajStevilka, NajZnesek, i, Znesek, Stava: integer;
begin
  NajZnesek := 0;
  for i := 0 to 36 do begin
    Znesek := 0;
    for Stava := 1 to StStav do
      if AliZadene(Stava, i) then
        Znesek := Znesek + VisinaStave(Stava) * Kolicnik(Stava);
    if (i = 0) or (Znesek < NajZnesek) then
      begin NajStevilka := i; NajZnesek := Znesek end;
  end; {for i}
  WriteLn('Izžrebana naj bo številka ', NajStevilka, '.');
end. {Pohlep}
```

R2004.1.4 Tekoče stopnice

N: 581 Naš podprogram, ki se kliče vsako milisekundo, najprej pogleda, če se stopnice premikajo (TrenutnaSmer); če se, zmanjšuje vrednost števca KakoDolgoSe, ki pove, kako dolgo se morajo še premikati. Če v tem času kdo stopi na pravi senzor, pa vrednost KakoDolgoSe spet povečamo in tako zagotovimo, da bodo stopnice še dovolj časa tekle v tej smeri. Če pade števec KakoDolgoSe na 0, pa stopnice ustavimo.

Ko stopnice mirujejo, naš podprogram gleda, če kdo stoji na kakšnem od senzorjev; če pride do tega, požene stopnice v pravi smeri. Če stojijo ljudje na obeh senzorjih, se je malo težje odločiti, v katero smer bi pognali stopnice; mogoče bi bilo nepošteno, če bi se odločili vedno za isto smer, zato si raje zapomnimo zadnjo smer, v katero so se stopnice premikale, in jih poženimo v nasprotno smer. V praksi je tako ali tako malo verjetno, da bi stopila dva človeka skoraj hkrati (znotraj iste milisekunde) na senzorja vsak na svoji strani stopnišča.

```
var TrenutnaSmer: SmerT value Ustavi;
    ZadnjaSmer: SmerT value Ustavi;
    KakoDolgoSe: integer value 0;

procedure EnkratNaMilisekundo;
begin
  if TrenutnaSmer <> Ustavi then begin
    KakoDolgoSe := KakoDolgoSe − 1;
    if ((TrenutnaSmer = Gor) and PotnikNaSenzorju(Spodaj))
    or ((TrenutnaSmer = Dol) and PotnikNaSenzorju(Zgoraj))
    then KakoDolgoSe := CasVoznje * 1000;
    if KakoDolgoSe <= 0 then
      begin PozeniStopnice(Ustavi); TrenutnaSmer := Ustavi end;
  end
  else begin
    if PotnikNaSenzorju(Spodaj) then
      if PotnikNaSenzorju(Zgoraj) then
        if ZadnjaSmer = Gor then TrenutnaSmer := Dol
        else TrenutnaSmer := Gor
      else TrenutnaSmer := Gor
    else if PotnikNaSenzorju(Zgoraj) then
      TrenutnaSmer := Dol;
    if TrenutnaSmer <> Ustavi then begin
      KakoDolgoSe := CasVoznje * 1000;
      ZadnjaSmer := TrenutnaSmer;
      PozeniStopnice(TrenutnaSmer);
    end; {if}
```

```
  end; {if}
end; {EnkratNaMilisekundo}
```

REŠITVE NALOG ZA DRUGO SKUPINO

R2004.2.1 Naraščajoča števila

Pri tej nalogi je koristno vzdrževati množico še neizpisanih števil, recimo ji M. Ko preberemo naslednje število vhodnega zaporedja (recimo mu b), moramo izpisati najmanjše še neizpisano število, ki je večje od b, če pa takega ni, pa najmanjše neizpisano število sploh. Ko to število izpišemo, ga moramo seveda odstraniti iz M.

N: 583

Vhod: $a[1..n]$

$M := \{1, \ldots, n\}$;
for $i := 1$ to n do begin
  $x :=$ najmanjše tako število iz M, ki je večje ali enako $a[i]$;
    če so vsa manjša od $a[i]$, vzemi najmanjše število iz M;
  izpiši x;
  $M := M - \{x\}$;
end;

V praksi lahko množico M izvedemo na različne načine. Zelo preprost, čeprav ne najbolj učinkovit način bi bila tabela booleanov:

```
var A: array [1..N] of integer;              { Vhodno zaporedje. }
    Neizpisano: array [1..N] of boolean;     { Še neizpisana števila. }
    i, x: integer;
begin
  for x := 1 to N do Neizpisano[x] := true;
  for i := 1 to N do begin
    x := A[i];
    while not Neizpisano[x] do
      begin x := x + 1; if x > N then x := 1 end;
    WriteLn(x);
    Neizpisano[x] := false;
  end; {for}
end.
```

Slabost te rešitve je, da lahko porabimo veliko časa za pregledovanje tabele v zanki while. Časovna zahtevnost takega postopka je v najslabšem primeru $O(n^2)$ (npr. če so vsi elementi vhodnega zaporedja enaki).

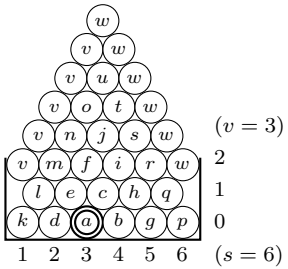
Učinkovitejšo rešitev dobimo, če hranimo neizpisane elemente v kakšni od drevesastih podatkovnih struktur (na primer AVL-drevo, rdeče-črno drevo

ipd.), ki podpirajo dodajanje, brisanje in iskanje najmanjšega elementa z določenega intervala ter za vsako od teh operacij porabijo le $O(\log n)$ časa. Časovna zahtevnost našega postopka bi bila tako le $O(n \log n)$.

Namesto drevesa bi lahko tudi ostali pri tabeli Neizpisano iz gornjega programa, le da bi jo dopolnili še z več manjšimi tabelami, v katerih bi vsak element predstavljal po dve, štiri, osem, šestnajst itd. zaporednih števil in bi imel vrednost true le, če so neizpisana vsa tista števila. Potem se nam ne bi bilo treba ves čas sprehajati po tabeli Neizpisano in povečevati x za 1, ampak bi lahko uporabili še te dodatne tabele in delali z x-om večje skoke (po dva, štiri, osem itd. elementov naenkrat). Tudi tako bi prišli do časovne zahtevnosti $O(n \log n)$.

R2004.2.2 Topovske krogle

N: 583 Oglejmo si primer na spodnji sliki:



Primer za $s = 6$, $v = 3$, $p = 3$. Krogle so označene s črkami od a naprej in to v takem vrstnem redu, v kakršnem padajo v zaboj. Lahko jih združimo v „leve“ in „desne“ skupine:

$L_0 = a,\ L_1 = def,\ L_2 = klmno,\ L_3 = vvvvv,$
$D_1 = bc,\ D_2 = ghij,\ D_3 = pqrstu,\ D_4 = wwwwww.$

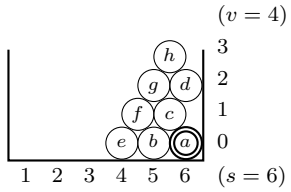
Vrstni red padanja je
$L_0, D_1, L_1, D_2, L_2, D_3, L_3, D_4$.

Vidimo, da gredo L-skupine lahko do največ L_k za $k = p - 1 + (v-1)/2$, D-skupine pa do D_k za $k = s - p + (v-1)/2$. (Če je v sod, vzamemo $v/2 - 1$ namesto $(v-1)/2$.) Če bi na primer v nekem konkretnem primeru ti dve formuli dovolili L-skupine do L_5 in D-skupine do D_3, bi to pomenilo, da dejansko nastopijo skupine do L_3 (za slednjo bi morala priti D_4, ki pa je nemogoča, zato tudi do L_4 in L_5 ne bomo prišli.)

Skupine $L_0, ..., L_{p-1}$ so polne in prispevajo $1, 3, ..., 2p-1$ krogel. Vse nadaljnje L-skupine so okrnjene (zaradi levega roba zaboja) in prispevajo po $2p - 1$ krogel.

Podobno so $D_1, ..., D_{s-p}$ polne in prispevajo $2, 4, ..., 2(s-p)$ krogel. Ostale D-skupine prispevajo po $2(s-p)$ krogel.

Prej opisani razmislek bi rekel, da nastopijo tu D-skupine do D_k za $k = s - p + v/2 - 1$, v našem primeru je to do D_1, v resnici pa moramo tu vzeti še eno D-skupino več, sicer bi potem neupravičeno zavrgli skupino L_2. Moramo pa pri tej zadnji L-skupini upoštevati, da ni čisto polna — za kroglami *efgh* bi morala v tej skupini nastopiti še ena krogla, vendar ta že pade čez rob.



Poseben primer je, če je v sod in $p = s$:

$$L_0 = a,\ D_1 = \text{(prazna)},\ L_1 = bcd,$$
$$D_2 = \text{(prazna)},\ L_2 = efgh.$$

Na tej sliki je primer za $v = 4$, $p = s = 6$.

Ostal je še en poseben primer — $s = 1$, ko se krogle nalagajo neposredno druga nad drugo in sosednje plasti niso zamaknjene za pol širine krogle kot pri večjih s. Pri $s = 1$ gre v zaboj preprosto v krogel.

Zdaj imamo vse potrebno, da lahko število krogel kar izračunamo:

```
function StKrogel(p, s, v: integer): integer;
var MaxKL, MaxKD, k, N: integer;
begin
  if s = 1 then begin StKrogel := v + 1; exit end;

  MaxKL := p - 1 + (v - 1) div 2;
  MaxKD := (s - p) + (v - 1) div 2;
  if (v mod 2 = 0) and (p = s) then MaxKD := MaxKD + 1;

  if MaxKL > MaxKD then MaxKL := MaxKD
  else if MaxKL < MaxKD then MaxKD := MaxKL + 1;

  N := 0;
  for k := 0 to Min(p - 1, MaxKL) do N := N + 2 * k + 1;
  if MaxKL >= p then N := N + (MaxKL - p + 1) * (2 * p - 1);
  if (v mod 2 = 0) and (p = s) then
    N := N - 1; { L_MaxKL je nepopolna — zadnja krogla se zvali čez rob. }

  for k := 1 to Min(s - p, MaxKD) do N := N + 2 * k;
  if MaxKD > s - p then N := N + (MaxKD - (s - p)) * 2 * (s - p);
  StKrogel := N + 1; { + 1 zato, ker naloga sprašuje, katera prva pade čez rob }
end; {StKrogel}
```

Malo preprostejša, vendar tudi manj učinkovita rešitev pa je simulacija — v neki tabeli vodimo za vsako x-koordinato (in še za polovične koordinate vmes) podatek o tem, kako visoko so nagrmadene krogle pri tem x. Potem lahko simuliramo, kako bi padale posamezne krogle, in vidimo, kdaj bi prva padla čez rob.

```
function StKrogel2(p, s, v: integer): integer;
const MaxS = ...;
var Visina: array [0..2 * MaxS] of integer;
    x, A, B, BB, C, N: integer;
begin
  for x := 0 to 2 * s do if Odd(x) then Visina[x] := -1 else Visina[x] := 0;
  N := 0; { N je števec krogel. }
```

```
  while true do begin
    x := 2 * p − 1; { Začetni položaj krogle pri padanju. }
    while (x > 0) and (x < 2 * s) do begin
      { B naj bo višina krogel na položaju x, A in C pa na sosednjih dveh položajih.
        BB naj bo višina krogel na položaju x, če bi trenutna krogla le še padla dol
        pri x in se ne bi več premikala levo ali desno. }
      A := Visina[x − 1]; B := Visina[x]; C := Visina[x + 1];
      BB := B; if A > BB then BB := A; if C > BB then BB := C;
      BB := BB + 1;
      { Premik na desno je mogoč, če je B > C in se krogla ne bi odbila od desne
        stene. Če tak premik ni mogoč, na podoben način razmislimo o razmiku v
        levo; če tudi to ne gre, se krogla ustavi. }
      if (B > C) and not ((x = 2 * s − 1) and (BB <= v)) then x := x + 1
      else if (B > A) and not ((x = 1) and (BB <= v)) then x := x − 1
      else break;
    end; {while}
    if (x <= 0) or (x >= 2 * s) then break; { Krogla je padla čez rob. }
    Visina[x] := BB; N := N + 1;
  end; {while}
  StKrogel2 := N;
end; {StKrogel2}
```

Spodnja tabela kaže, koliko krogel gre v nekatere majhne zaboje, preden pade prva čez rob.

	$s = 5$						$s = 6$					
v	$p = 1$	2	3	4	5	v	$p = 1$	2	3	4	5	6
6	15	28	33	22	15	6	15	30	43	37	22	15
5	15	28	33	22	9	5	15	30	43	37	22	9
4	8	19	24	13	8	4	8	19	32	26	13	8
3	8	19	24	13	4	3	8	19	32	26	13	4
2	3	10	15	6	3	2	3	10	21	15	6	3
1	3	10	15	6	1	1	3	10	21	15	6	1

R2004.2.3 GPS

N: 585 Ko dobimo od GPSa novo meritev in njeno natančnost, nas bo zanimalo, katera točka na daljicah je najbližja naši meritvi, pri tem pa še leži znotraj kroga, ki ima središče v izmerjenem položaju in čigar radij je ravno natančnost meritve.

Razdelimo ravnino na neke manjše celice, na primer s karirasto mrežo. Za vsako daljico lahko pogledamo, v katerih celicah vsaj deloma leži; enako storimo tudi za naš poizvedovalni krog. Za naš problem so zdaj zanimive le tiste daljice, ki ležijo vsaj deloma v kakšni od celic, v kateri vsaj deloma leži tudi naš krog. (Ta kriterij še ne zagotavlja, da ne bomo kakšne daljice pregledali po nepotrebnem, zagotavlja pa, da ne bomo spregledali nobene take,

ki vsaj deloma leži tudi v poizvedovalnem krogu.) Če si torej na začetku za vsako celico pripravimo seznam daljic, ki ležijo vsaj deloma v njej, moramo potem pri odgovarjanju na poizvedbo pregledati le sezname za tiste celice, ki se vsaj deloma prekrivajo z našim poizvedovalnim krogom.

Velikost celic moramo primerno izbrati; če bo premajhna, bo ležala vsaka daljica v veliko celicah in bodo zato seznami daljic za posamezno celico zasedli preveč prostora; poleg tega bo tudi poizvedovalni krog v tem primeru pripadal veliko celicam in bo treba pri vsaki poizvedbi pregledati veliko takih seznamov (to je nerodno predvsem, če jih nimamo vseh v pomnilniku, ampak bi morali za vsak seznam narediti en dostop do diska, da bi ga sploh prebrali). Če pa bodo celice prevelike, bo v vsaki od njih veliko daljic in bomo zato morali pri posamezni poizvedbi pregledati preveč daljic, kar nam bo požrlo preveč časa.

Še en koristen prijem, s katerim lahko poskusimo zmanjšati število daljic, ki jih bo treba pregledati, je ta, da prej pregledamo tiste celice, ki so bližje središču kroga. Če poznamo kakšno daljico, ki je od središča oddaljena d enot, potem nima smisla obiskovati celic, pri katerih je najbližja točka celice oddaljena od središča kroga za d ali več enot, saj v takih celicah gotovo ne bomo našli nobene boljše daljice od tiste, ki jo že poznamo.

Daljice lahko poindeksiramo tudi kako drugače, na primer tako, da ravnino z eno vodoravno in eno navpično premico razsekamo na štiri dele in to rekurzivno ponavljamo, ustavimo pa se, ko v posameznem delu ostane dovolj malo daljic ali pa ko posamezni del postane že premajhen. Na ta način dobimo drevesasto strukturo, ki se imenuje štiriško drevo (*quad-tree*). Obstaja še veliko drugih drevesastih podatkovnih struktur, namenjenih indeksiranju prostorskih podatkov, na primer R-drevesa in njihove številne različice.[107]

R2004.2.4 Skrajšanke

Uporabimo rekurzijo — na vsakem koraku poskusimo na vse možne načine izbrati naslednjo besedo (izmed tistih, ki jih doslej še nismo uporabili) in vse možne začetke (od 0 do M znakov). N: 586

```
const MaxN = 3; MaxM = 3;
type TabelaT = array [1..MaxN] of string;
```

[107] Literatura: A. Guttman: *R-trees: a dynamic index structure for spatial searching*, Proc. ACM SIGMOD, 1984, pp. 47–57; N. Roussopoulos, S. Kelley, F. Vincent, *Nearest neighbor queries*, Proc. ACM SIGMOD, 1995, pp. 71–79; G. R. Hjaltason, H. Samet: *Ranking in spatial databases*, Proc. SSD 1995, LNCS vol. 951, pp. 83–95; G. R. Hjaltason, H. Samet: *Distance browsing in spatial databases*, ACM TODS 24(2):256-318, June 1999; S. Bechtold, C. Böhm, D. A. Keim, H.-P. Kriegel: *A cost model for nearest-neighbor search in high-dimensional data space*, Proc. PODS 1997, pp. 78–86. Obstajajo tudi razni postopki za hitro gradnjo takega drevesa, če imamo že pri roki seznam vseh daljic, ki bi jih radi poindeksirali z njim, npr. I. Kamel, C. Faloutsos: *On packing R-trees*, Proc. CIKM 1993, pp. 490–99; S. T. Leutenegger, M. A. Lopez, J. Edgington: *STR: a simple and efficient algorithm for R-tree packing*, Proc. ICDE 1997, pp. 497–506.

```
procedure IzpisiVseKrajsanke(M, N: integer; Besede: TabelaT);
type MnozicaT = set of 1..MaxN;
var Skrajsanka: array [1..MaxN * MaxM] of char;
    Dolzina: integer;

  procedure Izpisi;
  var i: integer;
  begin
    for i := 1 to Dolzina do Write(Skrajsanka[i]);
    WriteLn;
  end; {Izpisi}

  procedure Rekurzija(Proste: MnozicaT);
  var StaraDolzina, i, j: integer;
  begin
    { Ostale so nam še besede iz množice Proste. Z ostalimi besedami smo
      doslej pripravili prvih Dolzina črk nastajajoče skrajšanke. }
    StaraDolzina := Dolzina;
    { Na vse možne načine izberimo naslednjo besedo. }
    for i := 1 to N do if i in Proste then begin
      Proste := Proste - [i];
      { Na vse možne načine izberimo število črk, ki jih bomo od te besede }
      for j := 0 to M do begin                          { obdržali. }
        if j > Length(Besede[i]) then break;
        { Pritaknimo začetek trenutne besede na konec nastajajoče skrajšanke. }
        if j > 0 then Skrajsanka[StaraDolzina + j] := Besede[i][j];
        Dolzina := StaraDolzina + j;
        { Če smo porabili vse besede, skrajšanko izpišimo... }
        if Proste = [] then begin if Dolzina > 0 then Izpisi end
        { ... sicer pa z rekurzivnim klicem nadaljujmo sestavljanje skrajšanke. }
        else Rekurzija(Proste);
      end; {for j}
      Proste := Proste + [i];
    end; {for i}
    Dolzina := StaraDolzina;
  end; {Rekurzija}

var i: integer; Proste: MnozicaT;
begin {IzpisiVseKrajsanke}
  Dolzina := 0;
  Proste := [];
  for i := 1 to N do Proste := Proste + [i];
  Rekurzija(Proste);
end; {IzpisiVseKrajsanke}
```

Vseh sestavljank, ki jih na ta način dobimo, je $N!(M+1)^N$ (lahko je seveda več enakih, med drugim tudi $N!$ pojavitev praznega niza), razen če ni kakšna od vhodnih besed krajša od M znakov.

Zanimivo, vendar malo težjo različico te naloge dobimo, če predpostavimo, da imamo tudi nek slovar besed in nas zanimajo le tiste skrajšanke, ki so navedene tudi v tem slovarju. Po možnosti bi do njih seveda radi prišli, ne da bi se morali ukvarjati tudi z vsemi ostalimi skrajšankami (ki jih je ogromno).

REŠITVE NALOG ZA TRETJO SKUPINO

R2004.3.1 Otoki

Najprej potopimo ves svet; število otokov je takrat 0, gladina pa zelo visoka. N: 590
Gladino potem počasi spuščamo; ko pogleda nova celica iz vode, postane nov otok; takrat pregledamo njene sosede in če so tudi že kopne, se otoki zlijejo. Ko smo pregledali vse celice na določeni višini, imamo pravo število otokov za to višino in to je zdaj eden od kandidatov za največje možno število otokov sploh.

$g := \infty$;
StOtokov := 0; MaxStOtokov := 1; StKopnih := 0;
for $x := 1$ to w do for $y := 1$ to h do Otok[y, x] := 0;

while StKopnih $< w \cdot h$ do begin
  (X, Y) := najvišja med vsemi celicami, ki imajo Otok$[y, x] = 0$; (†)
  $g' := v[Y, X]$;
  if $g' < g$ and StOtokov > MaxStOtokov then MaxStOtokov := StOtokov;
  $g := g$;

  Otok$[Y, X]$:= (neka nova enolična številka otoka);
  StOtokov := StOtokov + 1;

  za vsako od štirih sosed (X', Y') celice (X, Y) ponovi:
    if (Otok$[Y', X'] \neq 0$) and (Otok$[Y', X'] \neq$ Otok$[Y, X]$) then begin
      združi otoka Otok$[Y, X]$ in Otok$[Y', X']$; (‡)
      StOtokov := StOtokov − 1;
    end if;
  end for;
end while;

Časovna zahtevnost je odvisna od tega, kako implementiramo vrstici (†) in (‡). Preprosta izvedba (†) je, da se sprehodimo po celi tabeli višin in pogledamo, katera med potopljenimi celicami je najvišja. Preprosta izvedba (‡) je, da se sprehodimo po celi tabeli *Otok* in pri vsaki celici pogledamo, če pripada prvemu od obeh otokov; če mu, tisti element tabele *Otok* popravimo, da piše, da pripada ta celica drugemu otoku. Tako nam vsako izvajanje vrstic (†) in (‡) vzame $O(n)$ časa (če je n število vseh celic v mreži), celoten postopek pa zato $O(n^2)$.

Pri mrežah velikosti 100×100, s kakršnimi imamo opravka pri tej nalogi, je že ta preprosta izvedba dovolj hitra, zato na njej temelji tudi spodnji program. Za večje mreže, na primer 1000×1000, pa bi morali uporabiti kaj boljšega. Boljša implementacija (†) je, da na začetku posortiramo vse celice po padajoči $v[y, x]$, kar lahko naredimo v času $O(n \log n)$ (npr. postopek quicksort, ki ga imajo mnogi jeziki, npr. C in C++, že kar v standardni knjižnici). Boljša implementacija (‡) je, da z iskanjem v širino ali globino obiščemo le vse celice enega od obeh otokov in jih popravimo, da kažejo na drugi otok; pametno je obiskati manjšega od obeh otokov (velikosti otokov hranimo v neki dodatni tabeli): če naredimo tako, bo vsaka celica, ki ji spremenimo pripadnost otoku, pripadala po novem otoku, ki je vsaj dvakrat tolikšen kot tisti, ki mu je pripadala prej, in ker noben otok ne more imeti več kot n celic, tudi nobena celica ne bo več kot $\lg n$-krat spremenila pripadnosti otoku; to nam zagotavlja, da bo vrstica (‡) porabila vsega skupaj le $O(n \log n)$ časa. Zato tudi cel postopek porabi $O(n \log n)$ časa.[108]

```
program Otoki;
const
  MaxW = 100; MaxH = 100;
  DX: array [1..4] of integer= (−1, 1, 0, 0);
  DY: array [1..4] of integer= (0, 0, −1, 1);
var
  W, H: integer; { velikost mreže }
  V, Otok: array [1..MaxH, 1..MaxW] of integer;
  i, x, y, G, NovaG, NX, NY, SX, SY, NO: integer;
  StOtokov, MaxStOtokov, StKopnih, PrviProsti: integer;
  T: text;
begin
  { Preberimo vhodno datoteko. }
  Assign(T, 'otoki.in'); Reset(T); ReadLn(T, H, W);
  for y := 1 to H do begin
    for x := 1 to W do Read(T, V[y, x]);
    ReadLn(T);
  end; {for y};
  Close(T);

  { Na začetku so vse celice pod vodo. Postavimo G na nekaj velikega. }
  G := V[1, 1];
  for y := 1 to H do for x := 1 to W do begin
    if G < V[y, x] then G := V[y, x];
```

[108] Celoten postopek bi se dalo izboljšati celo do zahtevnosti $O(n)$, če bi poznali porazdelitev višin dovolj dobro, da bi lahko uporabili za urejanje celic po višini katerega od postopkov, ki porabijo le $O(n)$ časa, na primer bucket sort. Da bi tudi zlivanje otokov porabilo skupaj le $O(n)$ časa, bi morali uporabiti znano gozdnato strukturo za disjunktne množice (*disjoint-set forests*, gl. npr. Cormen *et al.*, *Introduction to Algorithms*, 22. pogl. v prvi izdaji, 21. v drugi).

```
    Otok[y, x] := 0;
  end; {for y}
  G := G + 1;
  StOtokov := 0; MaxStOtokov := 1;
  PrviProsti := 1; StKopnih := W * H;

  while StKopnih > 0 do begin

    { Znižajmo gladino toliko, da kakšna celica na novo pogleda iz vode. }
    NovaG := −1;
    for y := 1 to H do for x := 1 to W do
      if (Otok[y, x] = 0) and (V[y, x] > NovaG) then
        begin NovaG := V[y, x]; NX := x; NY := y end;

    { Če se je gladina res znižala, imamo v StOtokov pravo število otokov
      pri dosedanji gladini — mogoče je to največje število otokov doslej. }
    if (NovaG < G) and (StOtokov > MaxStOtokov) then
      MaxStOtokov := StOtokov;

    G := NovaG;
    { Celica (NX, NY) je pogledala iz vode. }
    Otok[NY, NX] := NY * W + NX;
    StOtokov := StOtokov + 1; StKopnih := StKopnih − 1;
    { Če ima kaj kopnih sosed, se otoki združujejo. }
    for i := 1 to 4 do begin
      SX := NX + DX[i]; SY := NY + DY[i];
      if (SX >= 1) and (SY >= 1) and (SX <= W) and (SY <= H) then
      if (Otok[SY, SX] <> 0) and (Otok[NY, NX] <> Otok[SY, SX]) then begin
        { Soseda (SX, SY) je kopna in ne pripada istemu otoku
          kot (NX, NY), zato ta dva otoka združimo. }
        NO := Otok[NY, NX];
        for y := 1 to H do for x := 1 to W do
          if Otok[y, x] = NO then Otok[y, x] := Otok[SY, SX];
        StOtokov := StOtokov − 1;
      end; {if}
    end; {for i}
  end; {while}

  { Izpišimo rezultat. }
  Assign(T, 'otoki.out'); Rewrite(T); WriteLn(T, MaxStOtokov); Close(T);
end. {Otoki}
```

Na `http://www.ngdc.noaa.gov/mgg/global/relief/ETOPO2/` so podatki o površju Zemlje (z ločljivostjo $2' \approx 3{,}7\,\mathrm{km}$ — torej mreža 10800×5400). Pri $g = 0$ (torej pri sedanji gladini oceanov) je na tej mreži 6024 otokov, največ otokov (47397) pa je pri $g = 306\,\mathrm{m}$ nad sedanjo gladino. (Pri tem smo definicijo sosednosti med celicami popravili tako, da smo upoštevali, da je Zemlja okrogla.) Kakšne posebne zveze z resničnostjo te številke najbrž nimajo, saj je v resnici ogromno otokov (in vzpetin ipd.) manjših od 3,7 km in se jih na

tej mreži najbrž sploh ne opazi.

Zanimivo in malo težjo različico naloge dobimo, če poskusmo podpreti tudi veliko večje mreže, tako da imamo lahko mrežo le na disku, ne gre pa cela v glavni pomnilnik našega računalnika.

R2004.3.2 SBN

N: 591

Ukaza SBN ni težko uporabiti za primerjanje števil po velikosti: a je manjši od b natanko tedaj, ko je razlika $a - b$ negativna.

```
        SBN r256, r12, r34, Oznaka
        { Tukaj vemo, da je r12 ≥ r34. }
        ...
Oznaka: { Tukaj vemo, da je r12 < r34. }
        ...
```

Ker imamo precej več kot n registrov (registrov je 256, naloga pa pravi, da je n največ 100), lahko preostale registre uporabimo za odlaganje pomožnih vrednosti, kot je v gornjem primeru razlika r12 − r34, ki smo jo vpisali v r256.

Prenašanje vrednosti iz enega registra v drugega je še enostavnejše. Prireditev r12 := r34 izvedemo takole:

```
SBN r12, r34, 0
```

Če bi radi zamenjali vrednosti v dveh registrih, potrebujemo tri prireditve:

```
SBN r256, r34, 0
SBN r34, r12, 0
SBN r12, r256, 0
```

Opisane operacije so pravzaprav že vse, kar potrebujemo za urejanje števil. Spomnimo se namiga iz besedila naloge in ga zapišimo v obliki algoritma (temu algoritmu se, mimogrede, reče „urejanje z izbiranjem“, *selection sort*):

```
var i, j: integer;
begin
  for i := 1 to n − 1 do
    { V registrih r[1], ..., r[i − 1] se že nahaja i − 1 najmanjših
      vrednosti tabele r. V tej iteraciji zunanje zanke bomo poskrbeli,
      da bo v register r[i] prišla najmanjša izmed preostalih vrednosti, torej
      izmed teh, ki so trenutno v registrih r[i], ..., r[n]. }
    for j := i + 1 to n do
      if not (r[i] < r[j]) then
        zamenjaj r[i] in r[j];
end.
```

Naš namišljeni procesor žal ne podpira posrednega naslavljanja; ne moremo mu na primer reči, naj nekaj naredi s tistim registrom, čigar številka je ta hip shranjena v registru `r123`. Možen izhod iz te zagate je, da vse zanke „razvijemo“ (*loop unrolling*) in kar ponovimo vse inštrukcije iz telesa zanke po enkrat za vsako vrednost števca. Tako pridemo do naslednje rešitve:

```
program Sbn;
var T: text; i, j, n: integer;
begin
  Assign(T, 'sbn.in'); Reset(T); ReadLn(T, n); Close(T);
  Assign(T, 'sbn.out'); Rewrite(T);
  for i := 1 to n - 1 do
    { Cilj notranje zanke je spraviti v r[i] najmanjšo od vrednosti r[i..n]. }
    for j := i + 1 to n do begin
      { Če je r[i] < r[j], skoči na end_i_j. }
      WriteLn(T, 'sbn r256, r', i, ', r', j, ', end_', i, '_', j);
      { Sicer zamenjaj registra r[i] in r[j]. }
      WriteLn(T, 'sbn r', j, ', r', i, ', 0'); { r[j] := r[i]. }
      { r[i] := r[i] - (r[i] - stara vrednost r[j]). }
      WriteLn(T, 'sbn r', i, ', r', i, ' r256');
      Write(T, 'end_', i, '_', j, ': ');
    end; {for j}
  end; {for i}
  { Izpišimo še en NOP — le toliko, da imamo kam pripeti zadnjo labelo. }
  WriteLn(T, 'sbn r256, r256, r256');
  Close(T);
end. {Sbn}
```

Pri zamenjavi smo porabili samo dva ukaza namesto treh, ker smo si pomagali z razliko med vrednostma registrov r[i] in r[j], ki smo jo naračunali malo prej pri primerjavi in jo shranili v r256. Tako porabimo po tri ukaze za vsak par registrov, kar nam da program dolžine $3n(n-1)/2+1$ (pri shemi točkovanja iz opisa naloge bi za to praviloma dobili sedem točk od desetih); če bi za zamenjavo porabili tri ukaze, bi bili s primerjavo vred že štirje in program bi bil dolg $4n(n-1)/2+1$ ukazov (za kar bi dobili šest točk od desetih).

Tako dolgo zaporedje ukazov smo dobili, ker smo morali v celoti razviti obe zanki, po i in po j. Do krajšega zaporedja pridemo, če najmanjše doslej odkrite vrednosti ne hranimo v r[i], pač pa v nekem posebnem registru (recimo mu m), neodvisnem od i.

```
var i, j, m: integer;
begin
  for i := 1 to n - 1 do begin
```

```
    { V registrih r[1], ..., r[i − 1] se že nahaja i − 1 najmanjših
      vrednosti tabele r. V tej iteraciji zunanje zanke bomo poskrbeli,
      da bo v register r[i] prišla najmanjša izmed preostalih vrednosti, torej
      izmed teh, ki so trenutno v registrih r[i], ..., r[n].
      Za začetek bomo z naslednjo zanko (po j) poskrbeli, da se bo ta
      najmanjša vrednost znašla v spremenljivki m, ostale vrednosti pa
      (mogoče malo premešane) v registrih r[i], ..., r[n − 1]. }
    m := r[n]; { Začasno si mislimo, da je register r[n] zdaj „prazen“. }
    for j := n − 1 downto 1 do begin
      if j < i then break;
      if m < r[j] then continue;
      zamenjaj r[j] in m;
    end; {for j}
    { Zdaj je m najmanjša izmed vrednosti, ki so bile prej v r[i], ..., r[n].
      Torej jo moramo vpisati v register r[i], dosedanjo vrednost r[i] pa
      pred tem premaknimo v prazni register r[n]. }
    r[n] := r[i]; r[i] := m; {⋆}
  end; {for i}
end.
```

Ker hočemo, da nam kode glavne zanke ne bi bilo treba ponavljati po enkrat za vsako vrednost i, se v njej ne smemo neposredno sklicevati na r[i]. Zato moramo vrstico {⋆} predelati v nekaj takšnega:

```
    for j := 1 to n − 1 do begin
      if j < i then continue;
      r[n] := r[j]; r[j] := m; break;
    end; {for j}
```

Zunanja zanka zdaj registra r[i] ne uporablja več neposredno, tako da njena koda ni več odvisna od trenutne vrednosti števca i. Zato nam v zaporedju ukazov SBN ne bo treba ponavljati telesa te zanke po enkrat za vsak i. Tako dobimo naslednjo rešitev z zaporedjem $8n-4$ ukazov SBN (pri našem točkovanju bi dosegla osem točk od desetih):

```
program Sbn;
const m = 254; i = 255; temp = 256;
var T: text; j, n: integer;
begin
  Assign(T, 'sbn.in'); Reset(T); ReadLn(T, n); Close(T);
  Assign(T, 'sbn.out'); Rewrite(T);
  WriteLn(T, 'sbn r', i, ', ', 1, ', 0');          { i := 1 }
  Write(T, 'zanka: ');
  WriteLn(T, 'sbn r', m, ', r', n, ', 0');         { m := r[n] }
  for j := n − 1 downto 1 do begin
    { if j < i then break }
```

```
      WriteLn(T, 'sbn r', temp, ', ', j, ', r', i, ', a_1');
      { if m < r[j] then continue }
      WriteLn(T, 'sbn r', temp, ', r', m, ', r', j, ', a_', j);
      { Zamenjaj r[j] in m. }
      WriteLn(T, 'sbn r', j, ', r', m, ', 0');          { r[j] := m }
      WriteLn(T, 'sbn r', m, ', r', j, ', r', temp);  { m := m − (m − stara r[j]) }
      Write(T, 'a_', j, ': ');
    end; {for j}
    { Zdaj je m = min(r[i..n]). Izvesti moramo r[n] := r[i] in r[i] := m. }
    for j := 1 to n − 1 do begin
      { if j < i then continue }
      WriteLn(T, 'sbn r', temp, ', ', j, ', r', i, ', b_', j);
      { Sicer je j = i. }
      WriteLn(T, 'sbn r', n, ', r', j, ', 0');                    { r[n] := r[i] }
      WriteLn(T, 'sbn r', j, ', r', m, ', 0');                    { r[j] := m }
      WriteLn(T, 'sbn r', temp, ', 0, 1, b_', n − 1);             { break }
      Write(T, 'b_', j, ': ');
    end; {for j}
    WriteLn(T, 'sbn r', i, ', r', i, ', −1');                     { i := i + 1 }
    { if i < n then goto zanka }
    WriteLn(T, 'sbn r', temp, ', r', i, ', ', n, ', zanka');
    Close(T);
end. {SBN}
```

Še krajšo in elegantnejšo rešitev dobimo, če uporabimo malo drugačen algoritem za urejanje.

```
var i, j: integer;
begin
  for i := 1 to n − 1 do
    { V zadnjih i − 1 registrih se že nahaja i − 1 največjih vrednosti
      cele tabele. V tej iteraciji zunanje zanke bomo poskrbeli, da bo v
      register r[n − i + 1] prišla (n − i + 1)-va največja vrednost. }
    for j := 1 to n − 1 do
      if not (r[j] < r[j + 1]) then
        zamenjaj r[j] in r[j + 1];
end.
```

Ko pride notranja zanka do največje vrednosti v še neurejenem delu tabele, bo ugotovila, da ta register ni manjši od naslednjega, in ju bo zato zamenjala; tako se tista največja vrednost premakne po tabeli za eno mesto naprej; v naslednji iteraciji bomo delali z naslednjim registrom, torej ravno s tistim, v katerega smo pravkar premaknili največjo vrednost; zato bo spet treba izvesti zamenjavo in tako naprej. Tako potone že ob prvem prehodu največja vrednost do konca tabele, ob drugem prehodu druga največja vrednost na predzadnje

mesto tabele in tako naprej. Ta postopek se imenuje „urejanje z mehurčki“ (*bubble sort*).

Za naše potrebe je ta postopek zelo primeren, ker sta gnezdeni zanki „neodvisni“ druga od druge — notranja zanka (po j) ne uporablja vrednosti i. Zato njenih ukazov v programu ni treba imeti v več kopijah (po eno za vsak i). Tako dobimo program, dolg samo $3n - 1$ ukazov.

```
program Sbn;
var T: text; i, n: integer;
begin
  Assign(T, 'sbn.in'); Reset(T); ReadLn(T, n); Close(T);
  Assign(T, 'sbn.out'); Rewrite(T);
  { Register r255 šteje prehode čez tabelo. }
  WriteLn(T, 'sbn r255, 0, ', n - 1'); { r255 := -(n - 1); }
  Write(T, 'zanka: ');
  for i := 1 to n - 1 do begin
    { Če je r[i] < r[i + 1], preskoči zamenjavo. }
    WriteLn(T, 'sbn r256, r', i, ', r', i + 1, ', end_', i);
    { Zamenjaj registra r[i] in r[i + 1]. }
    WriteLn(T, 'sbn r', i + 1, ', r', i, ', 0');
    WriteLn(T, 'sbn r', i, ', r', i, ' r256');
    Write(T, 'end_', i, ': ');
  end; {for i}
  { Povečajmo r255 za 1; ko ni več negativen, pomeni, da smo izvedli
    vseh n - 1 prehodov in lahko nehamo. }
  WriteLn(T, 'sbn r255, r255, -1, zanka');
  Close(T);
end. {Sbn}
```

Če nam je bolj kot dolžina programa pomemben čas izvajanja (število ukazov SBN, ki jih je treba izvesti, da uredimo n registrov), je odlična izbira urejanje z zlivanjem (mergesort). To nam zagotavlja časovno zahtevnost $O(n \log n)$, kar je načeloma precej bolje od zahtevnosti $O(n^2)$, ki jo dosežeta urejanje z izbiranjem in z mehurčki. Slabost urejanja z zlivanjem je, da potrebuje $n + 1$ pomožnih registrov namesto enega samega kot ostali tu opisani algoritmi. Če nas to hudo moti, lahko uporabimo urejanje s kopico (heapsort), ki potrebuje en sam pomožni register. Tako urejanje z zlivanjem kot s kopico lahko implementiramo tako, da dobimo (za urejanje n registrov) zaporedje približno $2n^2$ ukazov; je pa bilo urejanje s kopico vsaj v naši implementaciji malo počasnejše od urejanja z zlivanjem, saj je izvedlo slednje približno $2{,}5\, n \lg n$ ukazov, urejanje s kopico pa približno $4\, n \lg n$. Z algoritmom quicksort nam je uspelo dobiti sicer še hitrejše programe (do $2n \lg n$), vendar so tudi precej daljši (približno $n^4/6$ ukazov), razlike v hitrosti v primerjavi z zlivanjem pa postanejo opaznejše šele pri velikih n, ko je rešitev s quicksortom nesprejemljivo dolga.

R2004.3.3 Pamet

Nalogo lahko rešimo z rekurzijo. Rekurzivni podprogram preizkusi razne možnosti glede tega, koliko kosov pameti kupiti pri trenutnem trgovcu, pri vsaki možnosti pa izvede rekurzivni klic, da pregleda še možnosti nakupa ostalih kosov pameti pri ostalih trgovcih. Ko najdemo kakšno rešitev (torej ko uspemo nakupiti pravo število kosov pameti), jo primerjamo z najcenejšo doslej znano (spremenljivka MinCena) in če je cenejša, si jo zapomnimo. Med rekurzijo lahko vrednost MinCena uporabimo tudi za izogibanje preiskovanju neobetavnih nakupov („razveji in omeji“, *branch and bound*) — če so že doslej kupljeni kosi stali več kot MinCena denarja, nima smisla riniti še globlje v rekurzijo, saj bo končna cena tako dobljenih nakupov samo še višja in torej pri tem gotovo ne bomo dobili nobene boljše rešitve od najboljše doslej znane.

N: 592

```
program Butalci;
const
  MaxNB = 25;                  { največje število Butalcev }
  MaxNT = 500000;              { največje število trgovcev }
  MaxVsotaCen = 1000000000;    { vsota vseh cen v datoteki }
var
  NB, NT: integer;
  NK, SkupajNK: array [1..MaxNT] of integer;
  Cene: array [0..MaxNT, 0..MaxNB] of integer;
  MinCena: integer; { najmanjša doslej najdena cena za NB kosov pameti }

  { Ta podprogram predpostavi, da smo doslej že kupili NB − OstaloKosov kosov
    pameti in za to plačali CenaDoslej denarja, od trgovcev TrgovciOd..NT pa bi
    radi čim ceneje kupili še ostalih OstaloKosov pameti. }
  procedure Rekurzija(CenaDoslej, TrgovciOd, OstaloKosov: integer);
  var i, Cena: integer;
  begin
    i := 0;
    while (i <= OstaloKosov) and (i <= NK[TrgovciOd]) do begin
      Cena := CenaDoslej + Cene[TrgovciOd, i];
      if Cena < MinCena then begin
        if i = OstaloKosov then MinCena := Cena
        else if TrgovciOd < NT then
          { Nadaljujmo pri ostalih trgovcih, če imajo seveda vsi skupaj
            sploh dovolj pameti za naše potrebe. }
          if SkupajNK[TrgovciOd + 1] >= OstaloKosov − 1 then
            Rekurzija(Cena, TrgovciOd + 1, OstaloKosov − i);
      end; {if}
      i := i + 1;
    end; {while}
  end; {Rekurzija}
```

```
var T: text; it, ip, Cena: integer;
begin {Butalci}
  Assign(T, 'pamet.in'); Reset(T);
  ReadLn(T, NB, NT);
  for it := 1 to NT do begin
    Cene[it, 0] := 0;
    Read(T, NK[it]);
    for ip := 1 to NK[it] do begin
      Read(T, Cena);
      if ip <= NB then Cene[it, ip] := Cene[it, ip - 1] + Cena;
    end; {for}
    ReadLn(T);
  end; {for}

  { SkupajNK[i] = koliko kosov imajo trgovci od i-tega naprej. }
  SkupajNK[NT] := NK[NT];
  for it := NT - 1 downto 1 do
    SkupajNK[it] := SkupajNK[it + 1] + NK[it];

  MinCena := MaxVsotaCen + 1;
  Rekurzija(0, 1, NB);

  { Izpišimo rezultat. }
  Assign(T, 'pamet.out'); Rewrite(T); WriteLn(T, MinCena); Close(T);
end. {Butalci}
```

Če je Butalcev in/ali trgovcev veliko oz. če je testni primer dovolj neugodno sestavljen, bo ta program porabil preveč časa. Na testnih podatkih z našega tekmovanja bi zgornji program v sprejemljivo kratkem času rešil sedem od desetih testnih primerov. Do učinkovitejše rešitve bi prišli, če bi upoštevali, da podprogram Rekurzija vedno poišče najcenejši način, kako kupiti OstaloKosov kosov pameti od trgovcev TrgovciOd..NT. Vrednost, ki jo vrne, je enaka vrednosti tega nakupa, povečani za CenaDoslej. Torej bi lahko to vrednost (brez CenaDoslej), ko jo prvič izračunamo, shranili v neki tabeli in je kasneje ne bi bilo treba računati ponovno, ampak bi jo samo pobrali od tam. Takšni tehniki shranjevanja delnih rezultatov pravimo pomnjenje ali *memoizacija*. Lahko pa bi te rezultate računali tudi čisto sistematično, z gnezdenima zankama po TrgovciOd in po OstaloKosov (*dinamično programiranje*).

R2004.3.4 Izomorfizem

N: 593

Izberimo v vsakem od obeh grafov koren in ga tako predelajmo v hierarhično urejeno drevo. V takšnem drevesu lahko povezave usmerimo, da kažejo vedno od staršev na otroke (torej od vozlišča, ki je bližje korenu, na tisto, ki je dlje od njega); definicija izomorfizma naj ostane takšna kot prej, le da je zdaj pri ugotavljanju, ali lahko en graf dobimo iz drugega samo s preimenovanjem vozlišč, treba upoštevati tudi smer povezav.

Če v prvotnem grafu koren ni imel stopnje 3, nastane drevo, v katerem ima vsako vozlišče največ dve poddrevesi. Poddrevo, ki se začne v prvem drevesu pri u, in poddrevo, ki se začne v drugem drevesu pri v, sta izomorfni ali pa nista; naj nam to pove vrednost $I(u, v)$. Potem je: $I(u, v) = \mathsf{false}$, če nimata enako število poddreves; $I(u, v) = \mathsf{true}$, če sta oba brez poddreves; $I(u, v) = I(u', v')$, če imata vsak po eno poddrevo (u' in v'); in $I(u, v) = (I(u', v') \wedge I(u'', v'')) \vee (I(u', v'') \wedge I(u'', v'))$, če imata vsak po dve poddrevesi (u', u'' ter v', v''; eno od poddreves u-ja mora biti izomorfno enemu od poddreves v-ja, drugo pa drugemu, pri čemer vrstni red poddreves ni pomemben).[109] Vrednosti $I(u, v)$ lahko računamo z rekurzijo, lahko pa si že izračunane vrednosti tudi shranjujemo v kakšno tabelo, da jih kasneje ne bi bilo treba računati ponovno, če bi jih slučajno spet potrebovali. Slednje nam zagotavlja časovno zahtevnost $O(n^2)$. Spotoma lahko pripravljamo tudi tabelo, ki pove, katero vozlišče prvega drevesa ustreza kateremu vozlišču drugega drevesa (če sta drevesi res izomorfni).

Recimo, da sta prvotna grafa izomorfna, torej obstaja med njunima množicama vozlišč bijekcija f, ki spoštuje povezave: za vsak par vozlišč u in v velja $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$ (pri tem je E_1 množica povezav prvega, E_2 pa drugega grafa). Izberimo prvemu grafu poljubno vozlišče (s stopnjo 1 ali 2) kot koren; recimo mu r. Če zdaj poskusimo pri drugem grafu za koren vzeti po vrsti vsa njegova vozlišča, bomo prej ali slej preizkusili kot koren tudi $f(r)$ in takrat s primerjanjem dreves lahko ugotovimo, da izomorfizem res obstaja.

Po drugi strani, če enkrat ugotovimo, da izomorfizem med drevesoma z izbranima korenoma res obstaja, to seveda pomeni, da obstaja tudi med prvotnima grafoma.

Torej lahko izomorfizem res odkrivamo tako, da izberemo koren prvega drevesa in potem preizkusimo vse možne korene drugega; skupaj ima ta postopek časovno zahtevnost $O(n^3)$ in je za našo nalogo dovolj dober.

```
program Izomorfizem;
const MaxN = 1000;
type
  GrafT = record
    Stopnja: array [1..MaxN] of integer;
    Sosedje: array [1..MaxN, 1..3] of integer;
    { Ostala polja uporabljamo, ko grafu izberemo koren in ga predelamo v drevo. }
    Koren: integer;
    Otrok: array [1..MaxN, 1..2] of integer;
    StOtrok: array [1..MaxN] of integer;
  end; {GrafT}
```

[109] Če bi imeli opravka z drevesi, v katerih imajo lahko vozlišča tudi po več kot dve poddrevesi, bi ob primerjavi dveh vozlišč s po k poddrevesi morali preizkusiti vseh $k!$ načinov, kako razporediti poddrevesa enega vozlišča v pare s poddrevesi drugega vozlišča.

```
procedure PreberiGraf(var T: text; N: integer; var G: GrafT);
var i, u, v: integer;
begin
  for i := 1 to N do G.Stopnja[i] := 0;
  for i := 1 to N − 1 do begin
    ReadLn(T, u, v);
    G.Stopnja[u] := G.Stopnja[u] + 1; G.Sosedje[u, G.Stopnja[u]] := v;
    G.Stopnja[v] := G.Stopnja[v] + 1; G.Sosedje[v, G.Stopnja[v]] := u;
  end; {for i}
end; {PreberiGraf}

{ Ko v grafu izberemo koren, lahko iz njega naredimo drevo in za vsako
  vozlišče pogledamo, kdo so njegovi otroci oz. poddrevesa. }
procedure PostaviKoren(var G: GrafT; R: integer);

  procedure Rekurzija(u, Oce: integer);
  var i, v: integer;
  begin
    G.StOtrok[u] := 0;
    for i := 1 to G.Stopnja[u] do begin
      v := G.Sosedje[u, i]; if v = Oce then continue;
      G.StOtrok[u] := G.StOtrok[u] + 1;
      G.Otrok[u, G.StOtrok[u]] := v;
      Rekurzija(v, u);
    end; {for i}
  end; {Rekurzija}

begin {PostaviKoren}
  G.Koren := R;
  Rekurzija(R, −1);
end; {PostaviKoren}

var Preslikava: array [1..MaxN] of integer;

function StaDrevesiIzomorfni(N: integer; var G1, G2: GrafT): boolean;

  { Naslednji podprogram preveri izomorfnost dveh poddreves. }
  function Preveri(u, v: integer): boolean;
  var Sta: boolean;
  begin
    if G1.StOtrok[u] <> G2.StOtrok[v] then Sta := false
    else begin
      if G1.StOtrok[u] = 0 then Sta := true
      else if G1.StOtrok[u] = 1 then
        Sta := Preveri(G1.Otrok[u, 1], G2.Otrok[v, 1])
      else Sta :=    (Preveri(G1.Otrok[u, 1], G2.Otrok[v, 1])
                                    and Preveri(G1.Otrok[u, 2], G2.Otrok[v, 2]))
                 or (Preveri(G1.Otrok[u, 1], G2.Otrok[v, 2])
                                    and Preveri(G1.Otrok[u, 2], G2.Otrok[v, 1]));
```

```
      if Sta then Preslikava[u] := v;
    end; {if}
    Preveri := Sta;
  end; {Preveri}

begin {StaDrevesiIzomorfni}
  StaDrevesiIzomorfni := Preveri(G1.Koren, G2.Koren);
end; {StaDrevesiIzomorfni}

var T: text; u, N: integer; G1, G2: GrafT; StaIzo: boolean;
begin
  { Preberimo oba grafa. }
  Assign(T, 'izomorf.in'); Reset(T); ReadLn(T, N);
  PreberiGraf(T, N, G1); PreberiGraf(T, N, G2);
  Close(T);

  { Izberimo koren prvega grafa. }
  u := 1;
  while u <= N do if G1.Stopnja[u] < 3 then break else u := u + 1;
  PostaviKoren(G1, u);

  { Preizkusimo možne korene drugega grafa. }
  u := 1; StaIzo := false;
  while (u <= N) and not StaIzo do begin
    if G2.Stopnja[u] = G1.Stopnja[G1.Koren] then begin
      PostaviKoren(G2, u);
      StaIzo := StaDrevesiIzomorfni(N, G1, G2);
    end; {if}
    u := u + 1;
  end; {while}

  { Izpišimo rezultat. }
  Assign(T, 'izomorf.out'); Rewrite(T);
  if StaIzo then for u := 1 to N do WriteLn(T, u, ' ', Preslikava[u]);
  Close(T);

end. {Izomorfizem}
```

Ta algoritem je mogoče še precej izboljšati. Ko si enkrat izberemo oba korena, lahko izomorfnost dreves preverjamo učinkoviteje kot zgoraj. Poddrevesa pregledujmo po naraščajoči globini (globina poddrevesa = dolžina najdaljše veje v njem); označevali jih bomo s številkami ali „barvami“ in to tako, da izomorfna poddrevesa dobijo isto barvo, neizomorfna pa različno. Za začetek dajmo vsem listom (poddrevesa globine 0) številko 0, saj so si vsa izomorfna. Poddrevesi globine 1 sta si izomorfni natanko tedaj, ko imata obe enako število listov, zato lahko takim poddrevesom dodelimo kot barvo kar število listov. Za globlja poddrevesa pa lahko razmišljamo takole: dve taki poddrevesi sta si izomorfni, če sta obe enako globoki in če za vsako barvo velja, da imata obe enako število otrok take barve. Če torej za vsako poddrevo pripravimo

seznam barv njegovih otrok in ga uredimo, morata dve izomorfni poddrevesi dobiti zdaj popolnoma enak seznam. V našem primeru nima nobeno poddrevo več kot dveh otrok, zato so ti seznami dolgi največ dva elementa in je urejanje trivialno. Nato sestavimo zaporedje, katerega elementi so vsi seznami, dobljeni pri poddrevesih določene globine; to zaporedje uredimo, tako da pridejo enaki seznami (ki predstavljajo izomorfna poddrevesa) skupaj; zdaj torej ni težko pripisati vsaki skupini izomorfnih poddreves neko številko, ki je prej še nismo uporabili. Na koncu moramo le pogledati, če sta celotni drevesi dobili isto številko ali ne. Če obstaja n_i poddreves globine i, porabimo za urejanje tistega zaporedja $O(n_i \log n_i)$ časa, za vse globine skupaj pa zato $O(\sum_i n_i \log n_i) = O(\sum_i n_i \log n) = O(n \log n)$, kar je vsekakor precej bolje kot $O(n^2)$, kar je dosegel zgornji preprostejši algoritem.

Še ena izboljšava pa je, da pri drugem grafu ni treba preizkušati vseh možnih korenov, če pri prvem grafu koren pazljivo izberemo. Recimo, da bi porezali iz grafa vsa vozlišča s stopnjo 1; dobimo nek manjši graf, v katerem imajo nekatera vozlišča najbrž spet stopnjo 1; porežimo še ta; tako nadaljujemo in graf lupimo po plasteh od zunaj navznoter kot čebulo. Na koncu ostane neka plast z enim ali dvema vozliščema in ko pobrišemo še to, je graf prazen. Vozlišče (ali vozlišči) v zadnji plasti se imenuje *center* (središče) grafa. Če je bilo vsega skupaj k plasti in mi zdaj izberemo središče kot koren ter graf predelamo v drevo, bodo vsi listi na globini vsaj $k - 1$.

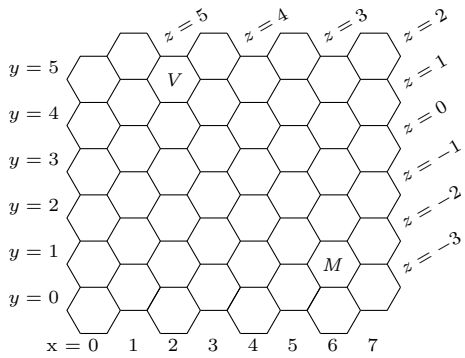
Poiščimo zdaj še center drugega grafa. Če se dobljeno število plasti kaj razlikuje od tistega pri prvem grafu, grafa že ne moreta biti izomorfna (kajti če sta izomorfna, lahko enega dobimo iz drugega samo s preimenovanjem vozlišč, to pa na odkrivanje centrov ne more vplivati). Če pa ima drugi graf tudi k plasti in bi ga zdaj radi predelali v drevo, da ga bomo lahko primerjali z drevesom, dobljenim iz prvega grafa, je jasno, da mora imeti drevo drugega grafa tudi vse liste na globini vsaj $k - 1$, tako kot jih ima prvo, saj drugače drevesi ne bosta mogli biti izomorfni. Torej za koren v drugem grafu nima smisla izbirati česa drugega kot centra drugega grafa! Ker ima lahko graf dva centra, moramo zdaj preizkusiti le dve možni drevesi, ki ju lahko dobimo iz drugega grafa, ne pa več $O(n)$ možnih dreves kot prej.

Ker lahko center poiščemo v času $O(n)$, bo preverjanje izomorfizma trajalo vsega skupaj $O(n^2)$, če uporabimo preprosti algoritem za izomorfizem dreves z začetka te rešitve, in $O(n \log n)$, če uporabimo izboljšani algoritem.

Majhna slabost naše naloge je, da v primeru, ko drevesi nista izomorfni, od programa zahteva samo prazno izhodno datoteko. Torej bi lahko kak bleferski program vedno pustil prazno izhodno datoteko in s tem pravilno rešil vse testne primere, pri katerih vhodna grafa res nista izomorfna (od desetih testnih primerov na našem tekmovanju je bil tak sicer en sam).

R2004.3.5 Čebelica Maja

Pri računanju razdalj na šesterokotni mreži je koristno vpeljati novo koordinatno os $z = y - \lfloor x/2 \rfloor$, kot to prikazuje spodnja slika. N: 595



Opisi premikov po mreži so precej preprostejši, če celice predstavimo s koordinatami (x, z) namesto (x, y).

Smer premika	Sprememba koordinat $\triangle x$	$\triangle z$
↑	0	1
↗	1	0
↘	1	−1
↓	0	−1
↙	−1	0
↖	−1	1

Recimo, da se hočemo premakniti iz točke (x_1, z_1) v (x_2, z_2); označimo $\triangle x = x_2 - x_1$ in $\triangle z = z_2 - z_1$.

Nekatere smeri premikanja nam spremenijo eno koordinato, drugo pa pustijo pri miru, dve smeri pa spremenita obe koordinati, vendar v nasprotni smeri (eno povečata in drugo zmanjšata). Če torej uporabljamo samo premike, ki spreminjajo po eno koordinato, lahko do $(\triangle x, \triangle z)$ pridemo v $|\triangle x| + |\triangle z|$ korakih.

Če sta $\triangle x$ in $\triangle z$ oba enako predznačena, je to tudi najkrajša možna pot. Smeri, ki spreminjata obe koordinati naenkrat, nam pri takih premikih ne koristita, ker eno koordinato povečujeta in drugo zmanjšujeta, mi pa bi radi ali obe zmanjšali ali pa obe povečali.

Če pa sta $\triangle x$ in $\triangle z$ različno predznačena, lahko uporabimo najprej eno od smeri, ki spreminja obe koordinati, da dosežemo pravi premik pri tisti koordinati, ki ima manjšo absolutno vrednost. Pri tem pa smo opravili že tudi del premika pri drugi koordinati (in to v pravo smer) in moramo zdaj opraviti v eni od smeri, ki spreminja le to koordinato, samo še toliko korakov, da doseže tudi ta koordinata pravo vrednost. Če je na primer $|\triangle x| \geq |\triangle z|$, naredimo najprej $|\triangle z|$ korakov, ki spreminjajo tako x kot z, nato pa še $|\triangle x| - |\triangle z|$ korakov, ki spreminjajo samo x. Če velja $|\triangle x| \leq |\triangle z|$, je razmislek podoben. Skupno število korakov je tako $\max\{|\triangle x|, |\triangle z|\}$.

```
program HotInsectAction;

  function Razdalja(X1, Y1, X2, Y2: integer): integer;
  var Z1, Z2, DX, DZ: integer;
  begin
    Z1 := Y1 − X1 div 2; Z2 := Y2 − X2 div 2;
```

```
    DX := X2 − X1; DZ := Z2 − Z1;
    if ((DX > 0) and (DZ > 0)) or ((DX < 0) and (DZ < 0))
    then Razdalja := Abs(DX) + Abs(DZ)
    else if Abs(DX) > Abs(DZ) then Razdalja := Abs(DX)
                              else Razdalja := Abs(DZ);
  end; {Razdalja}

var MX, MY, VX, VY, i, X, Y: integer; T, U: text;
begin {HotInsectAction}
  Assign(T, 'maja.in'); Reset(T);
  Assign(U, 'maja.out'); Rewrite(U);
  ReadLn(T, MX, MY); ReadLn(T, VX, VY);
  for i := 1 to 3 do begin
    ReadLn(T, X, Y);
    WriteLn(U, Razdalja(MX, MY, X, Y), ' ', Razdalja(VX, VY, X, Y));
  end; {for i}
  Close(T); Close(U);
end. {HotInsectAction}
```

Viri nalog za leto 2004: topovske krogle, GPS — Boris Gašperin; skrajšanke — Mitja Lasič; naraščajoča števila — Ivo List; tekoče stopnice, Butalci — Mojca Miklavec; čebelica Maja — Mojca Miklavec in Miha Vuk; SMS, izomorfizem — Boštjan Slivnik; otoki — Peter Keše, Jure Leskovec, Janez Brank; ploščice, ruleta, SBN — Janez Brank. Hvala Blažu Novaku za implementacijo rešitev nalog SBN in otoki.

Dodatne naloge

Včasih se ob pripravljanju nalog za tekmovanje nabere več nalog, kot jih tisto leto potrebujemo. Nekaj nalog torej ostane neuporabljenih, kar pa še ne pomeni, da so slabe ali nezanimive. Nekaj takšnih nalog je predstavljenih v tem razdelku. Opisi nalog, še posebej pa rešitve, praviloma niso tako dodelani kot pri nalogah, ki so bile uporabljene na tekmovanjih.

2002.X.1 Mobilni milijonar

Konec lanskega leta je veliko navdušenje povzročila Mobitelova igra Mobilni milijonar. Po zgledu Jonasovega Milijonarja na POP TV je bilo treba prek sporočil SMS pravilno in čim hitreje odgovoriti na zaporedje desetih vprašanj. Ker je pisanje sporočil SMS na mobilni telefon zamudno, so spretni programerji napisali program, ki je krmilil njihov mobilni telefon in odgovarjal na vprašanja. Pri tem jim je šlo na roko dejstvo, da je bilo vprašanj končno mnogo, ter da je sistem sporočal pravilni odgovor, če je igralec poslal napačnega (seveda je bilo v tem primeru igre konec in je bilo potrebno začeti znova). Program se je tako lahko, če je bil dovolj vztrajen, sam naučil pravilne odgovore na vprašanja in v mnogih poizkusih celo uspel pravilno odgovoriti na zaporedje desetih vprašanj. Če je pri tem imel še malo sreče, da je bil najhitrejši v odgovarjanju na zaporedje vprašanj (Mobitelovo omreže je bilo v tistem času prezasedeno s sporočili SMS za Mobilnega milijonarja, zato igralec ni mogel vplivati na hitrost vračanja odgovorov), je bila nagrada (milijon) njegova.

R: 633

Opiši postopek, ki se bo igral igro Mobilni milijonar, ne da bi vedel odgovor na eno samo vprašanje. Na voljo imaš naslednje funkcije:

```
{ Začne novo igro in prebere prvo vprašanje. }
procedure ZacniIgro; external;

{ Naslednje funkcije vračajo podatke o trenutnem vprašanju. }
function TrenutnoVprasanje: string; external; { Vrne prazen niz, če je igre konec. }
function StMoznihOdgovorov: integer; external;
function MozniOdgovor(StOdgovora: integer): string; external;

{ Pošlje odgovor; če je pravilen, prebere naslednje vprašanje in vrne
  prazen niz, sicer je igre konec, funkcija pa vrne pravilni odgovor. }
function PosljiOdgovor(Odgovor: string): string; external;
```

2002.X.2 Pretvarjanje znakov Unicode

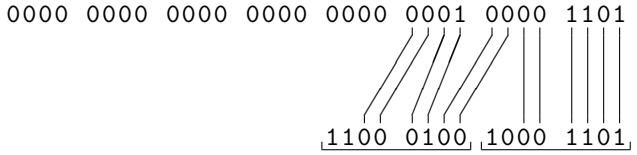
R: 635 Računalniki že desetletja uporabljajo osembitne znake za prikaz črk, kar omogoča prikaz 256 znakov. To pa je seveda premalo, da bi lahko ponazorili vse znake vseh obstoječih svetovnih jezikov. Problem rešuje standard Unicode, ki določa nabor 32-bitnih znakov. Za lažje prikazovanje znakov Unicode na računalnikih pa se vedno pogosteje uporablja pretvorba iz 32-bitnega nabora Unicode v niz enega ali več 8-bitnih znakov, imenovana UTF-8. Pretvorba UTF-8 deluje različno na različnih območjih 32-bitnega nabora znakov Unicode:

Območje Unicode (šestnajstiško)	Preslikava bitov (dvojiško)
00000000–0000007F	0xxxxxxx
00000080–000007FF	110xxxxx 10xxxxxx
00000800–0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000–001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000–03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000–7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
80000000–FFFFFFFF	pretvorba ni definirana

Oznaka „x“ pomeni en bit iz znaka Unicode; vedno se uporabi le potrebne spodnje bite.

Če je vrednost znaka med 0 in 127, se izpiše le en byte, ki vsebuje vseh spodnjih sedem bitov originalnega 32-bitnega znaka. V naslednjem območju (128 do 2047) se najprej izpiše byte, ki ima vedno nastavljene zgornje tri bite na 110, njegovih spodnjih pet bitov pa so biti 10 do 6 prvotnega znaka; v naslednjem bytu pa sta zgornja dva bita enaka 10, spodaj pa so še preostali biti prvotnega znaka (od 5 do 0).

Primer: črka „č“ ima po Unicode kodo 269 (šestnajstiško 0000010D) in se torej nahaja v drugem območju po zgornji tabeli. Preslika se v dva 8-bitna znaka:



Znak se torej preslika v zaporedje dveh bytov 11000100 10001101 (dvojiško) oziroma znaka s kodama C4 in 8D (šestnajstiško; kar je 196 in 141 v desetiškem zapisu).

Napiši podprogram UTF8, ki dobi za parameter 32-bitno število (lahko je kar tipa integer oz. **int**), ki predstavlja poljuben znak iz nabora Unicode, ter

izpiše zaporedje potrebnih osembitnih znakov glede na zgornjo pretvorbeno tabelo (če dobi znak iz območja 80000000–FFFFFFFF, naj ne izpiše ničesar). Za izpis imaš na voljo podprogram PutChar, ki izpiše en sam osembitni znak.

```
procedure PutChar(C: char); external;
extern int putchar(int c);
```

2003.X.1 Ražnjič

R: 637

Aci in Buba sta šla na gasilsko veselico, na kateri je bilo obilo bučne zabave, prodajali pa so tudi slastne ražnjiče. Veselica ju je pošteno zlakotila in zato jima je mama kupila en ražnjič, ki si ga bosta razdelila. Ražnjič je lesena palčka, na kateri so nabodeni kosi mesa in zelenjave.

Aci, ki bi rad zrastel v močnega fanta, ne mara drugega kot meso, Buba pa je vegetarijanka in jé le zelenjavo.

Zastavlja se jima vprašanje, kje naj razlomita paličico ražnjiča, tako da bo vsak od njiju dobil svoj konec, obenem pa hoče na njem čim več hrane, ki jo ima rad, in čim manj hrane, ki je ne mara. Aci in Buba sta pridna, zato vedno vse pojesta, prav tako pa bi bilo neolikano, če bi si med seboj menjala kose hrane.

Tako bo Aci s svojega kosa ražnjiča pojedel vse meso in se pošteno potrudil, da bo zmogel tudi zelenjavo. Buba bo z veseljem pojedla vso zelenjavo, le mesa, ki ga ne mara, bi rada pojedla čim manj.

Pomagaj Aciju in Bubi ter napiši program, ki jima bo povedal, kje naj razdelita ražnjič, tako da bosta oba skupaj dobila čim večjo količino (težo) hrane, ki jo imata rada, obenem pa bosta morala pojesti čim manj hrane, ki je ne marata.

Napiši program, ki najde mesto razreza ražnjiča, pri katerem bo vrednost izraza

$$Meso(Aci) - Zelenjava(Aci) + Zelenjava(Buba) - Meso(Buba)$$

čim večja. $Meso(x)$ pomeni količino mesa, $Zelenjava(x)$ pa količino zelenjave na x-ovem koncu ražnjiča.

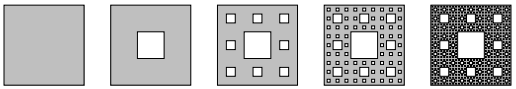
Na voljo imaš tabelo:

```
var Raznjic: array [0..N − 1] of integer;
int Raznjic[N];
```

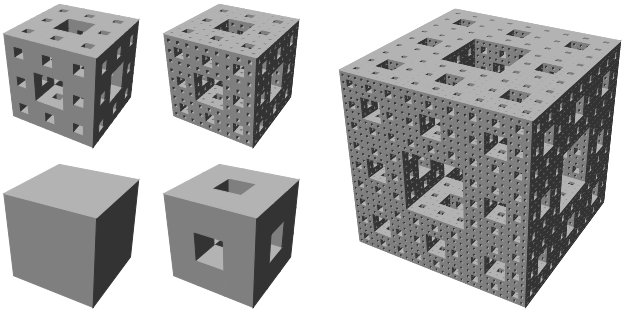
ki opisuje ražnjič, na katerem je nanizanih N kosov hrane. Če je Raznjic[i] $>$ 0, to pomeni, da je na i-tem mestu na ražnjiču kos mesa s težo Raznjic[i]. Če pa je Raznjic[i] $<$ 0, to pomeni, da je na i-tem mestu na ražnjiču kos zelenjave s težo $-$Raznjic[i].

2003.X.2 Kocka Sierpińskega

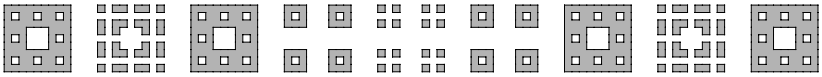
R: 638 Kvadrat Sierpińskega dobimo, če kvadrat v mislih razdelimo na 9 enakih delov in srednjega izrežemo. Vsakega izmed preostalih osmih kvadratov zopet razdelimo na 9 enakih delov in srednjega izrežemo, ... ter tako do neskončnosti. No, če smo iskreni, po neskončno korakih od začetnega kvadrata ne ostane kaj prida[110] in bi nam tudi najboljša lupa ne pomagala preveč, zato nas bodo zanimali le taki kvadrati, ki jih lahko dobimo po končno mnogo korakih. Primeri na sliki kažejo kvadrate, ki jih dobimo po nič, enem, dveh, treh in štirih korakih.



Podobno kot s kvadratom lahko naredimo tudi s kocko. Razdelimo jo na 27 kockic in zavržemo tiste, ki se ne dotikajo nobene od stranic prvotne kocke. Ostane nam dvajset od prvotnih 27 kockic in zdaj lahko na enak način obdelamo vsako od njih. Telo, ki se mu s ponavljanjem te operacije postopoma približujemo, se imenuje „kocka Sierpińskega“ ali tudi „Mengerjeva spužva“. Naslednja slika prikazuje telesa, ki jih dobimo po prvih nekaj korakih. Označimo s K_n telo, ki ga dobimo po n korakih.



Če želimo kocko, dobljeno po n korakih, narisati v dveh dimenzijah, jo lahko razrežemo na 3^n rezin in narišemo vsako posebej. Spodaj je primer razreza za kocko, kjer je $n = 2$. Opaziš lahko, da se vzorec s posamezne rezine pojavi večkrat. Tako so si prva, tretja, sedma in deveta rezina enake, prav tako druga in osma ter četrta in šesta rezina. Peta rezina ni enaka nobeni drugi.



[110] Ostane pravzaprav kar veliko točk (neštevno mnogo), le zelo so razpršene, tako da ima ta „lik“ (če lahko takšni množici točk rečemo lik) ploščino 0.

Napiši program, ki iz datoteke `kocka.in` prebere števili n ($0 \le n \le 7$) in r ($1 \le r \le 3^n$) in v datoteko `kocka.out` izpiše, pri koliko rezinah kocke K_n se pojavi isti vzorec kot pri r-ti rezini, nato pa še nariše r-to rezino kocke K_n. Prazna polja nariši s piko („`.`“), polna pa z zvezdico („`*`“).[111]

Primer vhodne datoteke:

```
2 4
```

Pripadajoča izhodna datoteka:

```
Vzorec 4. rezine se pojavi 2-krat.
***...***
*.*...*.*
***...***
.........
.........
.........
***...***
*.*...*.*
***...***
```

2003.X.3 Ugibanje nizov

Dva igralca se igrata naslednjo igro: prvi si je zamislil nek niz znakov (recimo mu s) in povedal drugemu le dolžino tega niza, drugi igralec pa bi zdaj rad ta niz uganil. Drugi igralec ugiba tako, da predlaga nek niz t, prvi igralec pa mu nato pove, na koliko mestih se istoležna znaka nizov s in t ujemata ter koliko izmed preostalih znakov t-ja se pojavlja na preostalih mestih s-ja. Na primer: če je $s =$ `abcabde` in $t =$ `ebaabda`, se niza ujemata na štirih mestih (`.b.abd.`), od ostalih znakov t-ja (to so znaki `e.a...a`) pa se na ostalih mestih s-ja pojavljata dva (namreč `e` in eden od `a`-jev). R: 642

Napiši podprogram, ki bo pomagal prvemu igralcu primerjati niza. Podprogram naj kot parametra dobi niza s in t in izpiše, na koliko mestih se ujemata in koliko izmed ostalih črk t-je se pojavlja na ostalih mestih t-ja. Predpostaviš lahko, da so nizi sestavljeni le iz malih črk angleške abecede (od `a` do `z`).

2003.X.4 Palindromi

Nek niz je *palindrom*, če se od konca proti začetku bere enako kot od začetka proti koncu (primeri: „radar“, „vodovodov“, „omejujemo“, pa v angleščini npr. “deified”, “reviver”, “rotator”). Znotraj nekega danega niza bi radi poiskali podnize, ki so palindromi. (Zanimali nas bodo le strnjeni podnizi, torej taki, pri katerih si črke sledijo neposredno druga za drugo.) Zanima nas taka skupina palindromnih podnizov, ki se ne prekrivajo, skupaj pa pokrijejo pa čim več črk prvotnega niza. Pri tem pa nočemo uporabljati palindromnih podnizov dolžine R: 643

[111] Zanimivo vprašanje je tudi, iz koliko ločenih koščkov je sestavljena r-ta rezina. Iz slike za kocko K_2 na str. 624 vidimo, da so za rezine kocke K_2 odgovori po vrsti takšni: 1, 16, 1, 4, 16, 4, 1, 16, 1.

1, saj bi z njimi lahko trivialno pokrili celoten niz. **Opiši postopek**, ki za dani niz poišče najboljšo takšno skupino palindromnih podnizov.

Primer: pri nizu *abcbaedeabc* bi lahko pokrili osem črk z dvema palindromoma, npr. *abcba* in *ede* ali pa *bcb* in *aedea*, še boljšo rešitev pa dobimo, če uporabimo zgolj palindrom *cbaedeabc*, ki pokrije kar devet črk.

2003.X.5 Seštevanje ulomkov

R: 645 **Napiši podprogram**, ki za dana cela števila a, b, c, d izračuna vsoto

$$\frac{a}{b} + \frac{c}{d}$$

in jo zapiše kot okrajšan ulomek.

2003.X.6 Urejanje logičnih vrednosti

R: 646 Imamo tabelo logičnih vrednosti:

```
const N = ...;
var Tabela: array [1..N] of boolean;

#define n ...
bool tabela[n];
```

Napiši podprogram, ki to tabelo čim hitreje uredi. Vrstni red je pri tipu boolean tak, da je vrednost false manjša od vrednosti true.

2003.X.7 Stikala

R: 648 V uporabniškem vmesniku nekega programa nastopa skupina n stikal (*checkboxes*), pri čemer jih sme biti naenkrat odkljukanih največ m. **Napiši podprogram**, ki ga bo sistem klical ob vsakem kliku na kakšno stikalo. Tvoj podprogram naj poskrbi, da ne bo nikoli odkljukanih več kot m stikal. Na voljo so podprogrami:

```
function StStikal: integer; external;
function MaxHkratiOdkljukanih: integer; external;
function JeOdkljukano(StStikala: integer): boolean; external;
procedure PostaviStikalo(StStikala: integer; Odkljukano: boolean); external;
```

Tvoj podprogram naj bo oblike:

```
procedure KlicanObSpremembi(StStikala: integer);
```

Parameter StStikala pove, na katero stikalo je uporabnik pravkar kliknil in mu s tem spremenil stanje.

2003.X.8 Nabori znakov

Recimo, da imamo nek niz znakov, v katerem je vsak znak predstavljen z nekim 32-bitnim celim številom (iz nekega velikega nabora znakov, na primer Unicode). Recimo tudi, da naš računalnik premore za izpisovanje nizov le pisave, ki ne podpirajo vseh možnih znakov, pač pa vsaka le neko podmnožico vseh možnih znakov. Zato se lahko zgodi, da našega niza ne moremo v celoti prikazati z eno pisavo, pač pa ga bo treba prikazati po koščkih in vmes preklapljati med pisavami. **Opiši postopek**, ki za dani niz in podatke o pisavah, ki so na voljo (za vsako pisavo poznamo številke vseh znakov, ki jih ta pisava vsebuje), ugotovi, kako ga je treba razdeliti na strnjene podnize, tako da bo teh podnizov čim manj in da se bo dalo vsak podniz v celoti izpisati z eno samo pisavo.[112] Opiši tudi podatkovne strukture, ki bi jih tvoj postopek uporabljal pri svojem delu. Predpostaviš lahko, da so številke znakov med 0 in 10^6 in da za vsak znak obstaja vsaj ena pisava, s katero ga lahko izpišemo.

R: 649

2003.X.9 Hiperkocka in mreža

Graf je struktura, sestavljena iz množice točk in množice povezav. Vsaka povezava povezuje dve točki. Pri tej nalogi so povezave vedno neusmerjene.

R: 650

Hiperkocka H_n je graf, ki ima 2^n točk, označenih z n-bitnimi celimi števili od 0 do $2^n - 1$. Povezava med točkama u in v obstaja natanko v primeru, ko dvojiški zapis števil u in v razlikuje v natanko enem bitu.

Mreža $M_{w,h}$ je graf, ki ima $w \cdot h$ točk, označenih s pari celih števil (x, y), $0 \le x < w$, $0 \le y < h$. Povezava med (x, y) in (x', y') obstaja natanko v primeru, če se točki v eni od koordinat ujemata, v drugi pa se razlikujeta za natanko 1.

Če v hiperkocki H_n spremenimo oznake točk in pobrišemo nekaj povezav, lahko iz nje dobimo mrežo $M_{2^m,2^{n-m}}$. **Opiši postopek**, ki za dani celi števili n in m, $0 \le m \le n$, pove, katere povezave hiperkocke H_n je treba pobrisati in kako je treba preimenovati ostale točke, da nam ostane ravno mreža $M_{2^m,2^{n-m}}$.

2004.X.1 Graf signala

Program za digitalno obdelavo signalov obdeluje tabelo vrednosti signala skozi čas:

R: 653

```
const StVrednosti = ...;
var Vrednosti: array [0..StVrednosti − 1] of integer;
```

[112] Malo drugačen problem pa dobimo, če se vprašamo, kako izpisati niz, tako da bomo pri tem porabili čim manj različnih pisav (četudi bo treba zato niz mogoče razsekati v večje število podnizov).

Tipično število vrednosti je nekaj sto tisoč. Program mora izrisovati tudi graf signala, česar ni smiselno početi bolj natančno, kot je ločljivost zaslona — približek signala sestavimo iz daljic, katerih začetna in končna točka se po x-osi razlikujeta za najmanj 1. (Recimo, da nas ne moti, če zaradi izpuščanja vrednosti na grafu manjkajo kakšni zanimivi pojavi, „špice“ ali kaj podobnega.) **Napiši del programa**, ki izriše približek signala. Poleg konstante StVrednosti in tabele Vrednosti uporabi še naslednje deklaracije:

```
const ZaslonX = ...;
  { Vodoravna ločljivost zaslona; lahko je manj ali več od StVrednosti. }

procedure NarisiDaljico(x1, y1, x2, y2: integer); external;
  { Nariše daljico od točke (x1, y1) do (x2, y2). }
```

Predpostaviš lahko, da se vrednosti gibljejo med 0 in navpično ločljivostjo zaslona, zato po y-osi signala ni treba premikati ali raztegovati.

Še deklaracije v C/C++:

```
#define StVrednosti ...
#define ZaslonX ...
int Vrednosti[StVrednosti];
extern void NarisiDaljico(int x1, int y1, int x2, int y2);
```

2004.X.2 Ribe

R: 654 Imamo zaporedje n rib. Za vsako ribo poznamo njeno velikost; število a_i je velikost i-te ribe ($i = 1, \ldots, n$). Izbrati si smemo eno od rib in nato zaporedoma početi naslednje: če sta obe sosedi trenutne ribe manjši od nje, se ustavimo, sicer pa ena od večjih sosed požre trenutno ribo (če sta obe sosedi večji, lahko sami izberemo, katera) in sama postane trenutna. Na koncu seštejemo velikosti vseh požrtih rib in vsoti prištejemo še velikost ribe, pri kateri se je postopek ustavil. **Opiši postopek**, ki ugotovi, kakšna je največja vsota, ki jo lahko na ta način dobimo. Če riba požre neko drugo, se ji velikost pri tem nič ne spremeni.

2004.X.3 Cezarjev kod

R: 658 Težava varne izmenjave podatkov ni le problem moderne civilizacije, temveč se z njo človeštvo sooča že od nekdaj. Pred dvema tisočletjema je Julij Cezar svojim vojskovodjem pošiljal sporočila, kodirana s preprosto zamenjavo, ki ji danes pravimo Cezarjev kod. Deluje tako, da vsako črko zamenjamo s črko, ki je v abecedi nekaj mest za njo. „Ključ“ tega kodiranja je torej število mest. Pri ključu 3 ($k = 3$) bi tako denimo črko `d`, ki se v angleški abecedi s 26 znaki

($n = 26$) nahaja na četrtem ($a = 4$) mestu, zamenjali z `g`, ki je sedma črka abecede: $a + k = 3 + 4 = 7$. Če kodiramo eno izmed zadnjih k črk abecede, je seveda ne moremo zamenjati s črko, ki je v abecedi k mest za njo, saj take črke sploh ni; zato v tem primeru jemljemo črke spet z začetka abecede. Če je torej $a + k > n$, bomo a zakodirali v črko $a + k - n$.

Če vemo, da je neko besedilo zakodirano s Cezarjevo šifro, ga lahko zelo enostavno dekodiramo, saj obstaja le $n - 1 = 25$ možnih vrednosti ključa k. **Napiši podprogram**, ki na osnovi znanega (podanega) dela besedila razbere in vrne ključ, po katerem je kodirano celotno sporočilo. Sporočilo sestavljajo le velike črke angleške abecede brez presledkov (`ABCDEFGHIJKLMNOPQRSTUVWXYZ`).

Tvoj podprogram naj bo takšne oblike:

```
function Dekodiraj(Kodirano, Znano: string): integer;
```

ali, v C/C++:

```
int Dekodiraj(char* Kodirano, char* Znano);
```

Kot vhod dobi dva niza: kodirano sporočilo (Kodirano), in nek strnjen podniz nekodiranega sporočila (Znano). Vrne naj vrednost ključa k. Lahko se zgodi, da je podniz nekodiranega sporočila izbran tako nesrečno, da je možnih več ključev k; v tem primeru lahko vrneš poljubnega med njimi.

Če hočeš, lahko predpostaviš, da imaš na voljo naslednji dve funkciji za pretvarjanje črk v zaporedne številke in nazaj:

```
const n = ...;

function StevilkaCrke(Crka: char): integer;
  { Črka mora biti ena od 'A', ..., 'Z'.
    Funkcija vrne njen položaj v abecedi (število med vključno 1 in n). }

function CrkaIzStevilke(Stevilka: integer): char;
  { Številka mora biti med vključno 1 in n. Vrne ustrezno izmed črk 'A', ..., 'Z'. }
```

Primer: če je nekodirano sporočilo

```
DOBRODOSLINATEKMOVANJUIZZNANJARACUNALNISTVA
```

in ga kodiramo po ključu $k = 3$, se črke preslikajo v skladu z naslednjo tabelo:

```
iz  ABCDEFGHIJKLMNOPQRSTUVWXYZ
v   DEFGHIJKLMNOPQRSTUVWXYZABC
```

Kodirano sporočilo bi v tem primeru bilo:

```
GREURGRVOLQDWHNPRYDQMXLCCQDQMDUDFXQDOQLVWYD
```

Če bi podprogram Dekodiraj dobil to sporočilo kot prvi parameter, kot drugega pa na primer niz `VANJUIZZ` (torej nek primeren podniz nekodiranega sporočila), bi moral vrniti vrednost 3.

2004.X.4 Števila zveri

R: 660 V spletni enciklopediji celoštevilskih zaporedij najdemo pod geslom A051003 naslednje zaporedje števil, ki vsebujejo (v desetiškem zapisu) 666 kot podniz:

$$666, 1666, 2666, 3666, 4666, 5666, 6660, 6661, 6662, 6663,$$
$$6664, 6665, 6666, 6667, 6668, 6669, 7666, 8666, 9666, 10666, \ldots$$

Recimo, da bi nas namesto števila 666 zanimalo kot podniz neko drugo naravno število, recimo b. Da nam pomagaš pri računanju zaporedja števil, ki vsebujejo b, nam **opiši postopek**, ki pri danih naravnih številih a in b izračuna, katero je najmanjše tako naravno število, ki je večje ali enako a in vsebuje b kot podniz (če ju zapišemo v desetiškem zapisu). Da bo lažje, lahko predpostaviš, da je a večji ali enak b.

Primer: pri $a = 6500$ in $b = 666$ je prvo primerno število 6666, pri $a = 5436934$ in $b = 705$ pa 5437050.

Tvoj postopek naj bo **čim bolj učinkovit**. Če se le da, naj deluje dovolj hitro, da bo uporaben tudi v primerih, ko sta števili a in b zelo veliki (recimo, da ima b v desetiškem zapisu več sto števk, a pa več tisoč števk). Opiši tudi, kako bi takšna števila predstavil v računalniku, ni pa ti treba pisati celotne implementacije v kakšnem konkretnem programskem jeziku.

2004.X.5 Zamenjave

R: 669 Ko napišemo predlogo besedila, je potrebno posamezne dele prilagoditi ciljnemu uporabniku. V predlogi označimo mesta, kamor naj se vpišejo ime, priimek, naslov itd. Včasih pa to ni dovolj in potrebne so bolj zakomplicirane zamenjave.

Napiši program, ki bo zamenjal gesla v besedilu z novimi vrednostmi in pri tem upošteval dano tabelo zamenjav.

Na začetku *vhodne datoteke* program dobi v vsaki vrstici po en par ⟨*geslo, nova vrednost*⟩ (vmes je presledek; geslo torej nikoli ne vsebuje presledka). Nato sledi prazna vrstica in pa besedilo. Gesla se v tabeli ne ponavljajo.

V besedilu se lahko pojavljajo nizi oblike `$(geslo)`, katere moraš zamenjati z novimi vrednostmi. Ti nizi so lahko tudi gnezdeni, na primer `$(geslo$(drugo_geslo))`; nikoli pa se ne raztezajo čez več vrstic. Če se oblika `$(nekaj)` pojavi v ravnokar obdelanem besedilu (po neki zamenjavi), je ne smeš zamenjati, saj bi take zamenjave lahko vodile v neskončno zanko. Če določenega gesla ni v tabeli, ne naredi pri njem nobene spremembe. Če sumiš, da je v besedilu napaka (manjka zaklepaj), ga prav tako pusti nespremenjenega.

Vsakič, ko program naleti na znaka `$$` v *originalnem* besedilu, naj izpiše namesto njiju le en `$`. Znak `$` ne sme voditi v novo zamenjavo (recimo v

primeru `$$(geslo)`, kjer je rezultat `$(geslo)`). Lahko pa je del gnezdene zamenjave (npr. `$(ge$$lo)`, kjer je rezultat nova vrednost pod geslom `ge$lo`). Podobno naj, če v originalnem besedilu naleti na znaka `$)`, izpiše namesto njiju le zaklepaj `)`.

V naslednjih primerih pa v originalnem besedilu ne naredi sprememb: če določenega gesla ni v tabeli (pustiš nedotaknjena tudi `$(` in `)`); če je v vrstici napaka (pri nizu `$(` manjka ustrezni zaklepaj); če znaku `$` sledi nek znak, ki ni niti `$` niti `(` niti `)`.

V *izhodno datoteko* izpiši besedilo, ki nastane po opisanih zamenjavah.

Omejitve. Dolžina gesel in novih vrednosti je manjša od 256 znakov. Število vseh zamenjav v tabeli je manjše od 1000. Vsaka vrstica besedila je krajša od 256 zankov. Globina gnezdenja gesel je največ 50.

Primer vhodne in izhodne datoteke je na str. 632.

2004.X.6 Vžigalice

Na mizi je nekaj vžigalic, razporejenih v v vrstic. V prvi vrstici je a_1 vžigalic, v drugi a_2 in tako naprej. Dva igralca izmenično vlečeta poteze. Vsaka poteza je sestavljena iz tega, da trenutni igralec izbere eno od vrstic in vzame iz nje eno ali več vžigalic. Igro izgubi tisti, ki pobere z mize zadnjo vžigalico. Pri vsakem začetnem razporedu vžigalic se izkaže, da obstaja natanko za enega od igralcev zmagovalna strategija — torej lahko igra tako, da bo zanesljivo zmagal, pa karkoli bo že počel drugi igralec. **Opiši postopek**, ki za dani začetni razpored vžigalic na mizo ugotovi, kateri igralec ima takšno zmagovalno strategijo.

R: 671

2004.X.7 Trikotniki

Raztreseni profesor Galerkin preučuje trdnost letalskih kril z metodo končnih elementov. Ta metoda lik, ki predstavlja obliko krila, razdeli na trikotnike, nato pa izvaja trdnostne izračune. Iznašel je novo, boljšo obliko krila, žal pa je izgubil opis njene razdelitve na trikotnike, brez katerega so rezultati neuporabni. Na srečo mu je ostal seznam daljic, ki so trikotnike sestavljali. Pomagaj mu s **programom**, ki bo izpisal seznam prvotnih trikotnikov. (Trikotnikov, ki so znotraj razdeljeni na manjše trikotnike, ne izpiši. Daljice so razpostavljene tako, da ne tvorijo nobenega mnogokotnika z več kot tremi oglišči, ki ne bi bil navznoter z dodatnimi daljicami razdrobljen na trikotnike. Daljice se ne sekajo in ne prekrivajo, pa tudi dotikajo se ne, razen v tem smislu, da imata lahko dve daljici skupno krajišče.)

R: 678

Daljice so podane v *vhodni datoteki*, katere prva vrstica vsebuje eno celo število — število daljic (največ 10 000), v vsaki nadaljnji vrstici pa je četverka realnih stevil x_1, y_1, x_2, y_2, ki predstavljajo daljico od točke (x_1, y_1) do (x_2, y_2). Dve števili imaš lahko za enaki, če se razlikujeta za manj kot 10^{-6}.

Vhodna datoteka („␣“ označuje enega od presledkov, ki se ga drugače ne bi videlo):

```
geslo niz
drugo_geslo drug_niz
gnezdeno_geslo drugo_
gnezdeno_geslo2␣
$(negnezdeno)geslo trik
$(gesloZoklepaji) nizGeslaZOklepaji
geslo4 $(zamenjavaZoklepaji)
geslo41 $(zamenjava
geslo42 Zoklepaji)
zamenjavaZoklepaji ne_pride_v_postev
dolar $
zaklepaj )

$(geslo) $(drugo_geslo)
$($(gnezdeno_geslo)geslo)
$($(gnezdeno_geslo2)geslo)
$($(negnezdeno)geslo)
$($(gesloZoklepaji))
$(geslo4)
$(geslo41)$(geslo42)
$(nedefinirano_geslo)
Primer napake:
  $(geslo$(gnezdeno_geslo)
Odvečen zaklepaj: $(gnezdeno_geslo))
Ubežne sekvence: $$(geslo$)
Trik brez uporabe ubežnih sekvenc:
  $(dolar)(geslo$(zaklepaj)
In z gnezdenjem:
  $($(dolar)(geslo$(zaklepaj))
  ali $($$(geslo$))
$(dolar)$(geslo) in $$(geslo$)
  ali $$$$(geslo$)
  in $(dolar)$(dolar)(geslo)
$ $a$b$c$d
```

Pripadajoča izhodna datoteka:

```
niz drug_niz
drug_niz
niz
trik
nizGeslaZOklepaji
$(zamenjavaZoklepaji)
$(zamenjavaZoklepaji)
$(nedefinirano_geslo)
Primer napake:
  $(geslodrugo_
Odvečen zaklepaj: drugo_)
Ubežne sekvence: $(geslo)
Trik brez uporabe ubežnih sekvenc:
  $(geslo)
In z gnezdenjem:
  $($(geslo))
  ali $($(geslo))
$niz in $(geslo)
  ali $$(geslo)
  in $$(geslo)
$ $a$b$c$d
```

Primer vhodne in izhodne datoteke za nalogo 2004.X.5.

Izhodna datoteka naj vsebuje v prvi vrstici število trikotnikov. V nadaljnjih vrsticah naj bo seznam trikotnikov, vsak v petih vrsticah. Trikotnik z oglišči (x_1, y_1), (x_2, y_2) in (x_3, y_3) naj bo zapisan kot:

(*prazna vrstica*)
x_1 y_1
x_2 y_2
x_3 y_3
x_1 y_1

Opis trikotnika se lahko začne s poljubnim od njegovih treh oglišč, nadalje-

vati pa se mora v pozitivni smeri (obratni od urinega kazalca). Vrstni red trikotnikov ni pomemben.

Primer dveh vhodnih datotek:

```
3
0.0 0.0 1.0 0.0
1.0 0.0 0.0 1.0
0.0 0.0 0.0 1.0
```

```
6
0.0 0.0 2.0 0.0
1.0 0.8 0.0 0.0
1.0 0.8 1.0 2.0
1.0 2.0 0.0 0.0
2.0 0.0 1.0 0.8
2.0 0.0 1.0 2.0
```

Možni pripadajoči izhodni datoteki:

```
1

0.0 0.0
1.0 0.0
0.0 1.0
0.0 0.0
```

```
3

0.0 0.0
2.0 0.0
1.0 0.8
0.0 0.0

0.0 0.0
1.0 0.8
1.0 2.0
0.0 0.0

1.0 0.8
2.0 0.0
1.0 2.0
1.0 0.8
```

REŠITVE DODATNIH NALOG

R2002.X.1 Mobilni milijonar

N: 621

Ko odgovarjamo na vprašanja, se nam počasi nabira znanje o njih. Če uspemo na neko vprašanje pravilno odgovoriti, si zapomnimo, kateri odgovor je bil pravilen, in v bodoče na to vprašanje vedno odgovarjajmo s tem odgovorom. Dokler pravilnega odgovora še ne poznamo, pa si shranjujmo vsaj odgovore, ki smo jih pri tem vprašanju že preizkusili in ugotovili, da so napačni. Pri odgovarjanju potem vedno izberimo kakšnega izmed odgovorov, za katere še nismo ugotovili, če so napačni ali ne. Spodnji program shranjuje podatke o znanih odgovorih v zapise tipa OdgovoriT in predpostavlja, da ima na voljo podprogram Poisci, ki poišče zapis za dano vprašanje (oz. vrne false, če takega zapisa še ni), in Shrani, ki shrani zapis za dano vprašanje (in pri tem povozi morebitni že obstoječi zapis). V praksi bi lahko Poisci in Shrani uporabljala tabelo parov ⟨*vprašanje*, *zapis tipa* OdgovoriT⟩, še učinkoviteje pa bi bilo te zapise shranjevati v razpršeno tabelo. Tabelo znanih napačnih odgovorov

(OdgovoriT.Napacni) bi bilo mogoče koristno preurediti v seznam (npr. če ne poznamo neke primerne zgornje meje za število možnih napačnih odgovorov, kot je MaxStNapacnih v spodnjem programu) ali pa kar v razpršeno tabelo (to bi bilo še posebej koristno v primeru, da nam lahko sistem predlaga veliko različnih napačnih odgovorov; z razpršeno tabelo bi lahko hitreje preverjali, kateri izmed trenutno ponujenih odgovorov so že znani kot napačni).

```
program MobilniMilijonar;

  procedure ZacniIgro; external;
  function TrenutnoVprasanje: string; external;
  function StMoznihOdgovorov: integer; external;
  function MozniOdgovor(StOdgovora: integer): string; external;
  function PosljiOdgovor(Odgovor: string): string; external;

  const MaxStNapacnih = ...;
  type OdgovoriT = record
    Pravilni: string;
    Napacni: array [1..MaxStNapacnih] of string;
    nNapacnih: integer;
  end; {OdgovoriT}

  function Poisci(Vprasanje: string; var Odg: OdgovoriT): boolean; external;
  procedure Shrani(Vprasanje: string; Odg: OdgovoriT); external;

var Vpr, Odg: string; Odgovori: OdgovoriT; i, j: integer; Zmaga: boolean;
begin
  Zmaga := false;
  repeat
    ZacniIgro;
    while true do begin

      Vpr := TrenutnoVprasanje;
      if Vpr = '' then begin Zmaga := true; break end;
      if not Poisci(Vpr, Odgovori) then { To vprašanje vidimo prvič. }
        begin Odgovori.Pravilni := ''; Odgovori.nNapacnih := 0 end;

      if Odgovori.Pravilni <> 0 then begin
        { Pravilni odgovor že poznamo; pošljimo ga... }
        Odg := PosljiOdgovor(Odgovori.Pravilni); Assert(Odg = '');
        continue; { ... in pojdimo na naslednje vprašanje. }
      end; {if}

      { Poiščimo kak odgovor, za katerega še ne vemo, ali je napačen. }
      i := 1;
      while i <= StMoznihOdgovorov do begin
        Odg := MozniOdgovor(i); j := 1;
        while j <= Odgovori.nNapacnih do
          if Odgovori.Napacni[j] = Odg then break else j := j + 1;
        if j > Odgovori.nNapacnih then break else i := i + 1;
      end; {while}
```

```
      Assert(i <= StMoznihOdgovorov);
      { Pošljimo ta odgovor. }
      Odg := PosljiOdgovor(MozniOdgovor(i));
      if Odg = '' then { Odgovor je bil pravilen! }
        Odgovori.Pravilni := MozniOdgovor(i)
      else begin { Odgovor je bil napačen. }
        Assert(Odgovori.nNapacnih < MaxStNapacnih);
        Odgovori.nNapacnih := Odgovori.nNapacnih + 1;
        Odgovori.Napacni[Odgovori.nNapacnih] := MozniOdgovor(i);
      end; {if}
      Shrani(Vpr, Odgovori);
      if Odg <> '' then break; { Napačen odgovor, konec igre. }
    end; {while}
  until Zmaga;
end. {MobilniMilijonar}
```

R2002.X.2 Pretvarjanje znakov Unicode

Z nekaj stavki **if** lahko ugotovimo, v katero območje spada naše vhodno število c, in nato vsako območje obravnavamo posebej ter skonstruiramo ustrezno zaporedje bytov, ki ga bo treba izpisati. Da izpulimo iz c-ja primerne skupine bitov, uporabimo operatorja **shr** (zamikanje desno) in **and** (logični „in“ nad istoležnimi biti). Z operatorjem **or** (logični „ali“ nad istoležnimi biti) poskrbimo še za prižiganje enic na mestih, ki jih zahteva standard za pretvorbo. Zaradi berljivosti je v spodnjem podprogramu več celoštevilskih konstant zapisanih v šestnajstiškem sistemu; spoznamo jih po znaku $ na začetku. Takšnih šestnajstiških konstant standardni pascal sicer ne predvideva, podpirajo pa jih mnogi prevajalniki. (Nestandardna, vendar v praksi široko podprta sta tudi operator **shr** in raba **and** in **or** nad celimi števili.)

N: 622

```
procedure UTF8_1(c: integer);
begin
  if c < $80 then
    PutChar(Chr(c))
  end else if c < $800 then begin
    PutChar(Chr($C0 or (c shr 6)));
    PutChar(Chr($80 or (c and $3F)));
  end else if c < $10000 then begin
    PutChar(Chr($E0 or (c shr 12)));
    PutChar(Chr($80 or ((c shr 6) and $3F)));
    PutChar(Chr($80 or (c and $3F)));
  end else if c < $200000 then begin
    PutChar(Chr($F0 or (c shr 18)));
    PutChar(Chr($80 or ((c shr 12) and $3F)));
    PutChar(Chr($80 or ((c shr 6) and $3F)));
```

```
    PutChar(Chr($80 or (c and $3F)));
  end else if c < $4000000 then begin
    PutChar(Chr($F8 or (c shr 24)));
    PutChar(Chr($80 or ((c shr 18) and $3F)));
    PutChar(Chr($80 or ((c shr 12) and $3F)));
    PutChar(Chr($80 or ((c shr 6) and $3F)));
    PutChar(Chr($80 or (c and $3F)));
  end else begin
    PutChar(Chr($FC or (c shr 30)));
    PutChar(Chr($80 or ((c shr 24) and $3F)));
    PutChar(Chr($80 or ((c shr 18) and $3F)));
    PutChar(Chr($80 or ((c shr 12) and $3F)));
    PutChar(Chr($80 or ((c shr 6) and $3F)));
    PutChar(Chr($80 or (c and $3F)));
  end; {if}
end; {UTF8_1}
```

Če hoteli preveriti še, da c ni z intervala `80000000`–`FFFFFFFF` (kjer pretvorba ni definirana), bi bilo pametno to storiti že na začetku podprograma. Drugače bi namreč lahko (če pomeni našemu prevajalniku tip integer predznačena cela števila) takšna števila naš podprogram videl kot negativna, pogoj **if** c < $80 bi bil izpolnjen in izpisal bi se nek nesmiseln znak. Neveljavne c-je lahko polovimo s takšnim pogojem na začetku podprograma:

```
  if (c and not $7FFFFFFF) <> 0 then SporociNapako;
```

Eden od razlogov, zakaj leta 2002 te naloge nismo uvrstili na tekmovanje, je bilo prav dejstvo, da ima očitno, puščobno in prav nič domiselno rešitev, ki jo vidimo zgoraj. Lahko pa smo pri reševanju te naloge tudi malo bolj ustvarjalni. Števila, manjša od 128, obravnavajmo kot poseben primer, ostala števila pa čisto sistematično. Če ugotovimo, kateri je najvišji prižgani bit števila c, lahko izračunamo, koliko bytov bo treba izpisati (recimo k). V prvem bytu mora biti prižganih najvišjih k bitov, v ostalih pa le najvišji bit; prvi byte vsebuje najvišjih nekaj bitov števila c, vsak od ostalih bytov pa po šest bitov c-ja.

```
procedure UTF8_2(c: integer);
var n, cc, m, k: integer;
begin
  if c < 128 then begin PutChar(chr(c)); exit end;
  { n naj bo indeks najvišjega prižganega bita (0..31) v c-ju. }
  cc := c; n := 0;
  while cc <> 1 do begin n := n + 1; cc := cc shr 1 end;
  if n > 30 then begin WriteLn('Napaka!'); exit end;
  { Koliko bytov bo treba izpisati? }
  k := (n + 4) div 5;
  { V prvem izpisanem bytu mora biti prižganih vrhnjih k bitov. }
```

```
  PutChar(Chr((((1 shl k) − 1) shl (8 − k)) or (c shr (6 * (k − 1)))));
  { V vsakem nadaljnjem bytu izpišemo po 6 bitov c-ja. }
  while k > 1 do begin
    k := k − 1;
    PutChar(Chr(128 or ((c shr (6 * (k − 1))) and 63)));
  end; {while}
end; {UTF8_2}
```

Slabost te rešitve pa je, da je lahko precej počasnejša od prve; polna je zank in pogojnih stavkov, ki se izvajajo precej dlje kot preproste bitne operacije v prvi rešitvi. Koliko ta razlika pomeni v praksi, je sicer odvisno od tega, kako dolgo se izvaja podprogram PutChar, ki ga kličeta obe rešitvi enako mnogokrat. Pri naših poskusih, ko je PutChar le zapisoval znake v neko tabelo v pomnilniku, je bila druga rešitev pet- do šestkrat počasnejša od prve (razen pri znakih od 0 do 127, kjer sta obe praktično enako hitri).

R2003.X.1 Ražnjič

Naj bo m_A (oz. z_A) količina mesa (oz. zelenjave), ki jo pri določeni razdelitvi ražnjiča poje Aci, m_B (oz. z_B) pa enako za Bubo. Naloga pravi, da moramo maksimizirati $m_A - z_A + z_B - m_B$. N: 623

Naj bo m skupna teža vseh kosov mesa, z pa vseh kosov zelenjave. Potem je $m_B = m - m_A$ in $z_B = z - z_A$; če vstavimo to v izraz, ki ga maksimiziramo, dobimo $m_A - z_A + z - z_A - m + m_A$, kar je $z - m + 2(m_A - z_A)$. Ker sta z in m neodvisna od tega, kako prelomimo ražnjič, bo torej za optimalno rešitev zadostovalo že maksimiziranje razlike $m_A - z_A$. V naši tabeli so predstavljeni kosi mesa s pozitivnimi, kosi zelenjave pa z negativnimi števili, zato je $m_A - z_A$ kar vsota tistih elementov naše tabele, ki predstavljajo Acijeve kose hrane. Zaradi načina delitve ražnjiča dobi Aci bodisi prvih nekaj kosov ali pa zadnjih nekaj kosov. Če označimo s s_i vsoto prvih i števil v tabeli, s t_i pa vsoto zadnjih i števil, vidimo, da moramo poiskati vrednost $\max\{s_0, \ldots, s_n, t_0, \ldots, t_n\}$. Ker je vsota vseh števil v tabeli enaka $m - z$, je $t_i = m - z - s_{n-i}$. Naš maksimum je zato naprej enak

$$\begin{aligned} & \max\{\max\{s_0, \ldots, s_n\}, \max\{m - z - s_n, \ldots, m - z - s_0\}\} \\ = \; & \max\{\max\{s_0, \ldots, s_n\}, m - z + \max\{-s_n, \ldots, -s_0\}\} \\ = \; & \max\{\max\{s_0, \ldots, s_n\}, m - z - \min\{s_0, \ldots, s_n\}\}. \end{aligned}$$

Ko se sprehajamo po tabeli in računamo vsote $s_0, \ldots, s_n$, si moramo torej zapomniti največjo in najmanjšo med njimi in na koncu izračunati maksimum po dobljeni formuli.

```
procedure Razdeli;
var Vsota, MinVsota, MaxVsota, MinKje, MaxKje, i: integer;
```

```
begin
  MinVsota := 0; MinKje := 0; MaxVsota := 0; MaxKje := 0; Vsota := 0;
  for i := 1 to N do begin
    Vsota := Vsota + Raznjic[i − 1];
    if Vsota < MinVsota then begin MinVsota := Vsota; MinKje := i end;
    if Vsota > MaxVsota then begin MaxVsota := Vsota; MaxKje := i end;
  end; {for i}
  if MaxVsota > Vsota − MinVsota
  then WriteLn('Aci naj dobi levih ', MaxKje, ' kosov,',
               'Buba pa desnih ', N − MaxKje, '.')
  else WriteLn('Aci naj dobi desnih ', N − MinKje, ' kosov,',
               'Buba pa levih ', MinKje, '.');
end; {Razdeli}
```

R2003.X.2 Kocka Sierpińskega

N: 624

Ker delimo pri sestavljanju kocke Sierpińskega vsako kocko vedno na $3 \times 3 \times 3$ manjše kocke, je koristno pri opisovanju položaja znotraj kock uporabljati trojiški zapis števil. Kocko K_n si predstavljajmo v trodimenzionalni mreži, sestavljeni iz $3^n \times 3^n \times 3^n$ manjših kockic; nekatere od njih so v K_n res prisotne, nekatere pa ne. Položaj vsake kockice lahko opišemo s trojico koordinat (x, y, z), pri čemer so $x, y, z \in \{0, 1, \ldots, 3^n - 1\}$, tako da si lahko te koordinate predstavljamo tudi kot n-mestna števila v trojiškem zapisu.

Spomnimo se, da smo do kocke K_n prišli tako, da smo jo razdelili na 27 manjših kock, nato sedem od njih zavgrli, na ostalih dvajsetih pa nadaljevali z enakim postopkom; iz vsake od tistih dvajsetih tako nastane pravzaprav po en izvod kocke K_{n-1}. Za vsako od teh 27 kock velja, da se vse kockice v njej ujemajo v prvi števki svoje x-koordinate, prav tako pa tudi v prvi števki svoje y-koordinate ter v prvi števki svoje z-koordinate. Za posamezno kockico (x, y, z) torej ni težko ugotoviti, v kateri od 27 kock leži: pogledati moramo le prve števke trojiškega zapisa koordinat x, y in z (koordinate pri tem zapišemo vedno z n števkami, četudi mora biti zato prvih nekaj števk enakih 0).

Če bi radi za neko kockico (x, y, z) ugotovili, ali je v naši kocki K_n prisotna, lahko zdaj razmišljamo takole. Poglejmo, kateri od 27 kock, v katere je razdeljena K_n, pripada kockica (x, y, z); če je to ena od tistih sedmih kock, ki jih pri tvorbi kocke K_n v celoti zavržemo (prepoznamo jih po tem, da imata vsaj dve od treh koordinat v trojiškem zapisu prvo števko 1, ne pa 0 ali 2), vemo, da naša kockica pač ni prisotna v K_n; drugače pa preračunajmo njene koordinate relativno glede na tisto kopijo kocke K_{n-1}, v kateri smo se znašli (kar pomeni le, da jim moramo v trojiškem zapisu odbiti prvo števko), in nadaljujmo z enakim razmislekom kot doslej. Rekurzija se konča, ko pridemo do kocke K_0, ki je pravzaprav že sama po sebi ena sama osnovna kockica in nima lukenj.

```
{ V parametru nn naj bo vrednost 3^n. }
function JePrisotna(x, y, z, nn: integer): boolean;
var i, nn1: integer;
begin
  while nn > 1 do begin
    { Koliko koordinat ima prvo številko 1? }
    i := 0; nn1 := nn div 3;
    if x div nn1 = 1 then i := i + 1;
    if y div nn1 = 1 then i := i + 1;
    if z div nn1 = 1 then i := i + 1;
    if i >= 2 then break;
    { Porežimo prvo številko vseh koordinat. }
    x := x mod nn1; y := y mod nn1; z := z mod nn1; nn := nn1;
  end; {while}
  { Če se ustavimo prej kot pri nn = 1, pomeni, da se je
    zanka končala s stavkom break, ker smo naleteli na luknjo. }
  JePrisotna := (nn = 1);
end; {JePrisotna}
```

Z besedami lahko rečemo tudi: kockica (x, y, z) je prisotna, če se nikoli ne zgodi, da bi imeli vsaj dve izmed števil x, y in z v trojiškem zapisu na istem mestu številko 1.

Zdaj ni težko narisati posamezne rezine kocke K_n:

```
procedure NarisiRezino(var T: text; z, nn: integer);
var x, y: integer;
begin
  for y := 0 to nn - 1 do begin
    for x := 0 to nn - 1 do
      if JePrisotna(x, y, z, nn)
      then Write(T, '*') else Write(T, '.');
    WriteLn(T);
  end; {for y}
end; {NarisiRezino}
```

Paziti moramo le na to, da besedilo naloge šteje rezine od 1 do 3^n, naš podprogram pa šteje z-koordinate od 0 do $3^n - 1$.

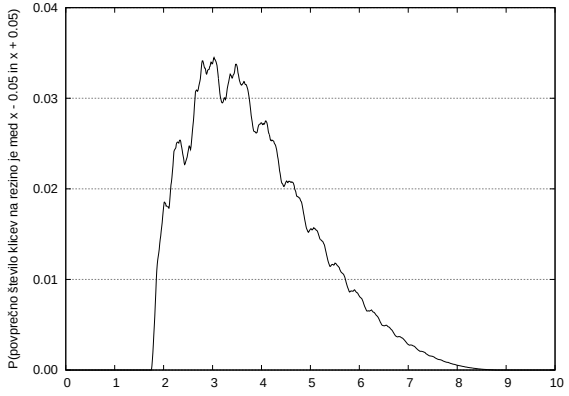
Kakšna je časovna zahtevnost tega postopka? Dovolj bo, če preštejemo število ponovitev zanke v podprogramu JePrisotna. Naj bo J_n povprečno število iteracij po vseh klicih JePrisotna za kocko K_n. Ko smo gradili kocko K_n, smo začeli s kocko s stranico 3^n in iz nje izrezali sedem kock s stranico 3^{n-1}; če pade naša trojica (x, y, z) v eno od njih, se ustavi tista zanka že po prvi iteraciji. V nasprotnem primeru pa naredimo po tisti prvi še nekaj iteracij, v povprečju ravno J_{n-1}; tako dobimo $J_n = 1 + (20/27)J_{n-1}$. Iz te zveze lahko izpeljemo eksplicitno formulo $J_n = 27/7 \cdot (1 - (20/27)^n)$, ki nam pove, da je zaporedje J_n strogo naraščajoče in konvergira k vrednosti $27/7$. Povprečno število izvajanj

naše zanke (če gledamo povprečje po vseh 27^n kockicah znotraj kocke K_n) je torej vedno $\le 27/7$, ne glede na to, kako velik n vzamemo. To je posledica dejstva, da so naše kocke K_n vse bolj redke, kolikor večje n-je gledamo. V kocki K_n je prisotnih le 20^n kockic od vseh 27^n možnih, torej le $(20/27)^n$ vseh kockic, ta vrednost pa pri naraščajočem n-ju pada proti 0. Zato se vse redkeje dogaja, da bi bilo treba izvesti veliko iteracij zanke **while**.

Podprogram NarisiRezino bo torej v povprečju porabil za vsak klic funkcije JePrisotna le konstantno mnogo časa, zato je njegova časovna zahtevnost v povprečju $O(9^n)$ (ker mora preveriti prisotnost za $3^n \cdot 3^n$ kockic). Res pa je, da se konstantni faktorji od rezine do rezine razlikujejo; če ima rezina veliko lukenj, bo potrebnih manj iteracij. Za rezino $z = 0$ (ki je pravzaprav kar navaden kvadrat Sierpińskega; na tej rezini je od 9^n možnih kockic prisotnih kar 8^n kockic) se na primer izkaže, da je povprečno število izvajanj notranje zanke pri kockicah s te rezine enako $9 \cdot (1-(8/9)^n)$. Po drugi strani je za rezino $z = (3^n - 1)/2$ (na sredi kocke, kjer je največ lukenj; prisotnih je le 4^n od 9^n možnih kockic) povprečno število izvajanj notranje zanke le $9/5 \cdot (1 - (4/9)^n)$ iteracij na kockico. Torej imamo pri rezini $z = 0$ skoraj petkrat toliko dela kot pri sredinski rezini $z = (3^n - 1)/2$ (razmerje je tem bližje 5, čim večji je n). Ostale rezine so nekje vmes; če označimo povprečno število iteracij po kockicah iz rezine z kocke K_n z $J_n(z)$, je

$$J_n(z) = \begin{cases} 1 + 4J_{n-1}(z \bmod 3^{n-1}) & : \text{če je } 3^{n-1} \le z < 2 \cdot 3^{n-1} \\ 1 + 8J_{n-1}(z \bmod 3^{n-1}) & : \text{sicer.} \end{cases}$$

Graf na spodnji sliki kaže, kako so porazdeljene vrednosti $J_n(z)$ pri $n = 30$, torej pri kocki K_{30}. Vidimo lahko, da so res vse med približno 1,8 in 9, vendar so visoke vrednosti $J_n(z)$ redkejše (le malo rezin zahteva veliko iteracij), zato je povprečje res blizu $22/7 \approx 3{,}86$.



Ta graf prikazuje (kot funkcijo števila x), pri kolikšnem deležu rezin kocke K_{30} leži povprečno število iteracij v zanki podprograma JePrisotna, če ga pokličemo za vse kockice s tiste rezine, na intervalu $[x - 0{,}05, x + 0{,}05]$. Povprečno število iteracij po rezini se giblje od približno 1,8 za zelo „prazne“ rezine (na sredini kocke) do skoraj 9 za zelo „polne“ (na površju kocke), povprečje po vseh rezinah pa je približno 22/7.

Oglejmo si zdaj še drugi del naloge: kako ugotoviti, koliko rezin je enakih

neki dani rezini. Naša kocka K_n je sestavljena iz $3 \times 3 \times 3$ delov, od katerih je sedem praznih, v dvajsetih pa tičijo kopije kocke K_{n-1}. V srednji tretjini kocke K_n (torej za $3^{n-1} \le z < 2 \cdot 3^{n-1}$) manjka pet delov (vsi razen vogalnih), v zgornji in spodnji tretjini pa le eden (osrednji). Zgornja in spodnja tretjina sta si torej povsem enaki; če je z neka rezina iz zgornje ali spodnje tretjine, je njeno število pojavitev v celi kocki dvakrat tolikšno kot samo v njeni tretjini. Za srednjo tretjino pa velja, da se nobena njena rezina ne pojavlja v kateri od ostalih dveh tretjin, saj se od rezin v teh dveh tretjinah razlikuje po razporedu praznih delov. Ostane le še vprašanje, kolikokrat se neka rezina pojavlja v svoji tretjini kocke K_n. Ker so si glede praznih delov vse rezine v neki tretjini enake, jih lahko razlikujemo le na podlagi nepraznih delov — tistih, kjer je del naše rezine pravzaprav rezina manjše kocke K_{n-1}; natančneje povedano, neprazni deli rezine z kocke K_n so enaki rezini ($z \bmod 3^{n-1}$) kocke K_{n-1}. Rezina z kocke K_n se torej v svoji tretjini te kocke pojavi prav tolikokrat, kolikorkrat se pojavi rezina ($z \bmod 3^{n-1}$) kocke K_{n-1} v celi kocki K_n.

```
function StPojavitevRezine(z, nn: integer): integer;
var s: integer;
begin
  s := 1;
  while nn > 1 do begin
    nn := nn div 3;
    { Če ni iz srednje tretjine, je število pojavitev dvakrat tolikšno. }
    if z div nn <> 1 then s := s * 2;
    z := z mod nn;
  end; {while}
  StPojavitevRezine := s;
end; {StPojavitevRezine}
```

Vidimo lahko, da ta funkcija ne dela drugega, kot da prešteje, kolikokrat se v trojiškem zapisu števila z pojavlja števka 1. Če se pojavlja k-krat, se rezina z v kocki K_n pojavlja 2^{n-k}-krat. Različnih rezin s takšnim številom pojavitev pa je $\binom{n}{k}$, kajti na toliko načinov si lahko izberemo, katere izmed n števk bodo imele vrednost 1. Ker sta zgornja in spodnja tretjina vsake kocke enaki, je oblika rezine odvisna le od tega, kje v trojiškem zapisu števila z so enice, ne pa tudi od tega, kje na ostalih mestih so ničle in kje dvojke.[113]

Zapišimo še glavni blok našega programa:

[113] Sebi podobnim množicam, kot je kocka Sierpińskega iz te naloge, pravimo „fraktali“. Za več o njih gl. npr. MathWorld *s. v.* “Fractal” ali poplavo literature, npr. B. B. Mandelbrot, *The Fractal Geometry of Nature*. Po istem kopitu kot tukaj kocko Sierpińskega bi lahko dobili še kup podobnih fraktalov, npr. če bi zgolj spremenili pravilo o tem, na koliko delov razdelimo kocko na vsakem koraku in katere od njih zavržemo. Namesto s kocko bi lahko delali tudi s tetraedrom, s štiristrano piramido, pa z liki, npr. s kvadratom („preproga Sierpińskega“), trikotnikom (iz njega bi lahko dobili npr. trikotnik Sierpińskega ali pa Kochovo snežinko), lahko tudi z navadno daljico (dobimo „Cantorjev prah“, ki ga vidimo tudi na robovih naše kocke Sierpińskega). Če uporabimo podobne prijeme nad krivuljami,

```
var i, z, n, nn: integer; T: text;
begin {MengerjevaSpuzva}
  Assign(T, 'kocka.in'); Reset(T); ReadLn(T, n, z); Close(T);
  z := z - 1;
  nn := 1; for i := 1 to n do nn := nn * 3;
  Assign(T, 'kocka.out'); Rewrite(T);
  WriteLn(T, 'Vzorec ', z + 1, '. rezine se pojavi ',
            StPojavitevRezine(z, nn), '-krat.');
  NarisiRezino(T, z, nn);
  Close(T);
end. {MengerjevaSpuzva}
```

Za konec si oglejmo še vprašanje, iz koliko ločenih kosov je sestavljena rezina z kocke K_n; označimo to z $A_n(z)$ (za $0 \le z < 3^n$). Koristno je vpeljati še pomožno količino: koliko izmed teh kosov se dotika vsakega roba rezine; recimo temu $B_n(z)$. Pokazati je mogoče, da velja naslednje (označimo $z' = z \bmod 3^{n-1}$, torej z brez prve števke v trojiškem zapisu):

$$A_0(z) = B_0(z) = 1$$

$$A_n(z) = \begin{cases} 4A_{n-1}(z'), & \text{če } 3^{n-1} \le z < 2 \cdot 3^{n-1} \\ 1, & \text{sicer, če } B_{n-1}(z') = 1 \\ 8(A_{n-1}(z') - B_{n-1}(z')), & \text{sicer} \end{cases}$$

$$B_n(z) = \begin{cases} 2B_{n-1}(z'), & \text{če } 3^{n-1} \le z < 2 \cdot 3^{n-1} \\ 3B_{n-1}(z') - 2, & \text{sicer} \end{cases}$$

Najmanjše možno število kosov je 1, kar dosežemo npr. pri $A_n(0)$ za poljuben n. Največje število kosov pri rezinah kocke K_n pa je $(44/35) \cdot 8^{n-1} + (8/5) \cdot 3^{n-1} + (8/7)$, kar dosežemo npr. pri $A_n(1)$.[114]

R2003.X.3 Ugibanje nizov

N: 625

Z zanko pojdimo po obeh nizih in primerjajmo istoležne znake; tako lahko preštejemo, na koliko mestih se niza `s` in `t` ujemata v istoležnih znakih. Za ostale znake niza `t` pa nas zanima, koliko jih najdemo med ostalimi znaki niza `s`. Torej, če se neka črka a pojavlja v nizu t recimo $\#_t(a)$-krat, v nizu s pa $\#_s(a)$-krat, in je $\#_t(a) \le \#_s(a)$, vemo, da bomo lahko v s-ju našli vse pojavitve črke a iz niza t; če pa je $\#_t(a) > \#_s(a)$, bomo lahko pokrili vse s-jeve pojavitve črke a, nekaj pojavitev a-ja v t-ju pa bo še ostalo. V vsakem

lahko dobimo npr. razne prostor zapolnjujoče krivulje, kot so Peanova, Hilbertova, zmajeva itd.

[114] Do te formule ni težko priti na podlagi prej navedenih rekurzivnih formul. Dokaz, da res nima nobena rezina več kot toliko kosov, pa je malo bolj zamuden in ga tu ne bomo navajali. Pomagamo si lahko tako, da najprej dokažemo, da doseže pri $z = 1$ svoj maksimum vrednost $B_n(z)$, pa tudi vrednost $A_n(z) - 3B_n(z)$; iz tega pa ni težko dokazati, da doseže pri $z = 1$ svoj maksimum tudi $A_n(z)$.

primeru se torej število ujemanj poveča za $\min\{\#_s(a), \#_t(a)\}$. Vrednosti $\#_s$ in $\#_t$ hrani spodnji podprogram v tabelah HistS in HistT (imeni sta dobili po tem, da vsebujeta ravno to, kar potrebujemo, če hočemo s histogramom predstaviti pogostost posameznih črk v nizih s in t).

```
procedure Primerjaj(s, t: string);
var i, StUjemanj: integer; c: char;
    HistS, HistT: array ['a'..'z'] of integer;
begin
  if Length(s) <> Length(t) then
    begin WriteLn('Niza sta različno dolga!'); exit end;
  for c := 'a' to 'z' do
    begin HistS[c] := 0; HistT[c] := 0 end;
  StUjemanj := 0;
  for i := 1 to Length(s) do
    if s[i] = t[i] then StUjemanj := StUjemanj + 1
    else begin
      HistS[s[i]] := HistS[s[i]] + 1;
      HistT[t[i]] := HistT[t[i]] + 1;
    end; {if}
  WriteLn('Niza se ujemata na ', StUjemanj, ' mestih.');
  StUjemanj := 0;
  for c := 'a' to 'z' do
    if HistS[c] < HistT[c]
    then StUjemanj := StUjemanj + HistS[c]
    else StUjemanj := StUjemanj + HistT[c];
  WriteLn('Od ostalih črk niza t se jih ', StUjemanj,
          ' pojavlja na ostalih mestih niza s.');
end; {Primerjaj}
```

R2003.X.4 Palindromi

Recimo, da je naš vhodni niz s dolg n znakov. Označimo s $s[i..j]$ podniz, ki obsega znake od i-tega do j-tega. Naj bo $D(i)$ množica dolžin palindromnih podnizov niza s, ki se končajo pri znaku $s[i]$, torej i-tem znaku niza s. Opazimo, da sta si prvi in zadnji znak palindroma vedno enaka, vse vmes pa je tudi samo zase palindrom (na primer: v palindromu *radar* tiči palindrom *ada*). Če se torej pri znaku $s[i]$ končuje nek palindrom dolžine d, pomeni, da sta si znaka $s[i]$ in $s[i-d+1]$ enaka, znaki vmes pa tvorijo nek palindrom, ki ima torej dolžino $d-2$ in se končuje pri znaku $s[i-1]$. Očitno pa je, da velja tudi obratno: če sta si $s[i]$ in $s[i-d+1]$ enaka in znaki med njima tvorijo palindrom, je tudi celoten podniz $s[i-d+1..i]$ palindrom. Tako torej vidimo:

N: 625

$$d \in D(i) \iff (d-2) \in D(i-1) \wedge s[i-d+1] = s[i].$$

To pravilo lahko uporabimo za $d > 1$, za $d = 1$ pa tako ali tako vedno vemo, da je črka $s[i]$ sama zase palindrom in je torej $1 \in D(i)$. V množico $D(i)$ je koristno vključiti tudi število 0, ker bomo potem lahko z zgornjim pravilom opazili tudi palindrome dolžine 2 (če sta v nizu s dve zaporedni enaki črki).

Ko imamo enkrat množice $D(i)$, ni težko poiskati najboljšega razporeda palindromov. Naj bo $f(i)$ največje število znakov, ki jih lahko pokrijemo s palindromnimi podnizi niza $s[1..i]$, ne da bi se ti podnizi med seboj prekrivali. Možni kandidati za najboljši razpored so naslednji: (1) lahko ne uporabimo nobenega palindroma, ki se konča pri $s[i]$, in v tem primeru bo $f(i)$ kar enak $f(i-1)$, torej najboljšemu razporedu za niz $s[1..i-1]$; (2) lahko pa uporabimo nek palindrom dolžine d s koncem pri $s[i]$, in ker se palindromi med seboj ne smejo prekrivati, predstavljajo ostali palindromi v našem razporedu najboljše pokritje niza $s[1..i-d]$, torej $f(i-d)$. Tako vidimo:

$$f(i) = \max\{f(i-1), \max\{f(i-d) + d : d \in D(i), d > 1\}\}.$$

Vrednost funkcije $f(i)$ lahko torej računamo kar sproti, medtem ko določamo tudi množico $D(i)$.

Časovna zahtevnost našega postopka je sorazmerna s skupnim številom elementov v vseh množicah $D(i)$, torej s skupnim številom vseh palindromnih podnizov niza s; v najslabšem primeru je to $O(n^2)$. Prostorska zahtevnost pa je le $O(n)$ za tabele, v katerih hranimo funkcijo f, dve množici $D(i)$ in še eno tabelo, ki nam pove, kako smo prišli do vrednosti $f(i)$, da bomo lahko rekonstruirali najboljšo rešitev.

```
program Palindromi;
const MaxDolz = ...;
var s: string;
    { D[id] vsebuje seznam elementov množice D(i), D[1 − id] pa množice D(i − 1). }
    D: array [0..1, 1..MaxDolz + 1] of integer;
    nD: array [0..1] of integer; { nD[.] = velikost množice D[.] }
    f, fKako: array [0..MaxDolz] of integer;
    i, j, id, dd: integer;
begin
    ReadLn(s);
    id := 0; nD[id] := 0; f[0] := 0;
    for i := 1 to Length(s) do begin
     id := 1 − id; nD[id] := 2; D[id, 1] := 0; D[id, 2] := 1;
     { V tabeli D[1 − id] je množica D(i − 1). V tabeli D[id] pa
       bomo pripravili množico D(i); zaenkrat smo dodali vanjo števili 0 in 1. }
     f[i] := f[i − 1]; fKako[i] := 1; { če ne uporabimo palindroma s koncem pri s[i] }
     for j := 1 to nD[1 − id] do begin
       dd := D[1 − id, j]; { trenutni element množice D(i − 1) }
       if dd >= i − 1 then continue;
       if s[i − dd − 1] <> s[i] then continue;
```

```
        { Dodajmo dd + 2 v množico D(i). }
        nD[id] := nD[id] + 1; D[id, nD[id]] := dd + 2;
        if f[i − dd − 2] + dd + 2 > f[i] then { nova najboljša rešitev }
          begin f[i] := f[i − dd − 2] + dd + 2; fKako[i] := dd + 2 end;
      end; {for j}
    end; {for i}
    { Rekonstruirajmo rešitev. V tabelo D[0] bomo pisali indekse, na katerih
      se končajo uporabljeni polinomi. }
    nD[0] := 0; i := Length(s);
    while i > 0 do begin
      nD[0] := nD[0] + 1; D[0, nD[0]] := i;
      i := i − fKako[i];
    end; {while}
    for j := nD[0] downto 1 do begin
      i := D[0, j]; dd := fKako[i]; if dd = 1 then continue;
      { V najboljši rešitvi je uporabljen palindrom dolžine dd s koncem pri i. }
      while dd > 0 do begin dd := dd − 1; Write(S[i − dd]) end;
      Write(' ');
    end; {for j}
    WriteLn;
end. {Palindromi}
```

R2003.X.5 Računanje z ulomki

Za začetek preverimo, če je katero od števil b in d enako 0; če je, vrednost $a/b + c/d$ pač ni definirana. Drugače pa vemo, da lahko ulomka a/b in c/d spravimo na skupni imenovalec bd in njuno vsoto zapišemo kot $(ad+bc)/(bd)$. Zdaj jo moramo le še okrajšati. Označimo števec z $u = ad + bc$, imenovalec z $v = bd$; poiskati moramo torej največji skupni delitelj števil u in v (recimo mu $\gcd(u, v)$) in ju oba deliti z njim. Eleganten način za iskanje največjega skupnega delitelja je Evklidov algoritem (glej str. 449). Ta temelji na opažanju, da je $\gcd(u, v) = \gcd(v, u \bmod v)$, pri čemer je $u \bmod v$ ostanek pri deljenju u z v. Če to formulo uporabljamo v zanki, postajata števili u in v vse manjši, dokler ne deljenje enkrat ne izide ($u \bmod v = 0$, kar je znak, da je največji skupni delitelj kar v).

N: 626

```
program Ulomki;

  function Gcd(u, v: integer): integer;
  var t: integer;
  begin
    while v > 0 do begin t := v; v := u mod v; u := t end;
    Gcd := u;
  end; {Gcd}

  procedure Krajsaj(var u, v: integer);
  var t: integer;
```

```
  begin
    t := Gcd(Abs(u), Abs(v)); u := u div t; v := v div t;
  end; {Krajsaj}

  procedure Izpisi(u, v: integer);
  begin
    if v < 0 then begin u := -u; v := -v end; { prenesimo predznak v števec }
    if v > 1 then Write(u, '/', v) else Write(u);
  end; {Izpisi}

var a, b, c, d, u, v: integer;
begin {Ulomki}
  ReadLn(a, b, c, d);
  if (b = 0) or (d = 0) then
    begin WriteLn('Deljenje z nič!'); exit end;
  Write('Vsota '); Izpisi(a, b); Write(' in '); Izpisi(c, d);
  u := a * d + b * c; v := b * d;                                    { (†) }
  Krajsaj(u, v);
  Write(' je '); Izpisi(u, v); WriteLn('.');
end. {Ulomki}
```

Slabost tega programa je, da za skupni imenovalec vedno uporabi produkt $b \cdot d$. To vsekakor je skupni imenovalec, ni pa nujno najmanjši skupni imenovalec; če sta b in d velika, bi bil lahko njun produkt prevelik za spremenljivko tipa integer, njun najmanjši skupni imenovalec pa mogoče ne. Če označimo najmanjši skupni večkratnik števil b in d z $\operatorname{lcm}(b,d)$ in se spomnimo, da je $\operatorname{lcm}(b,d) \cdot \gcd(b,d) = b \cdot d$, vidimo, da lahko skupni imenovalec $\operatorname{lcm}(b,d)$ računamo po formuli $(b/\gcd(b,d)) \cdot d$. Poleg tega lahko ulomka a/b in c/d pred računanjem še pokrajšamo, da bomo delali ves čas s čim manjšimi števili. Vrstico (†) gornjega programa bi torej lahko zamenjali z:

```
  Krajsaj(a, b); Krajsaj(c, d);
  v := (b div Gcd(b, d)) * d;
  u := a * (v div b) + c * (v div d);
```

R2003.X.6 Urejanje logičnih vrednosti

N: 626

Ena možnost je, da preprosto preštejemo, kolikokrat se pojavlja v tabeli false, in potem v prvih toliko celic vpišemo vrednost false, v ostale pa true.

```
procedure Urejanje;
var i, StFalse: integer;
begin
  StFalse := 0;
  for i := 1 to N do if not Tabela[i] then StFalse := StFalse + 1;
  for i := 1 to StFalse do Tabela[i] := false;
```

```
  for i := StFalse + 1 to n do Tabela[i] := true;
end; {Urejanje}
```

Slabost te rešitve je, da smo morali narediti dva prehoda skozi tabelo: v prvem smo šteli vrednosti false, v drugem pa smo popravljali vsebino tabele. Tabelo lahko uredimo tudi z enim samim prehodom: z enim števcem se premikamo z leve, z drugim z desne in če pride levi števec do vrednosti true, desni pa do false, ju zamenjamo. Levi števec pušča na svoji levi same vrednosti false, desni števec pa na svoji desni same vrednosti true. Ko se števca srečata, vemo, da je celotna tabela urejena. Ta postopek se zgleduje po postopku za razdeljevanje tabele pri urejanju z algoritmom quicksort.

```
procedure Urejanje2;
var i, j: integer;
begin
  i := 1; j := n;
  while i < j do begin
    { Invarianta: Tabela[1..i − 1] = false, Tabela[j + 1..n] = true. }
    while (i < j) and not Tabela[i] do i := i + 1;
    while (i < j) and Tabela[j] do j := j − 1;
    if i < j then begin
      Tabela[i] := false; Tabela[j] := true;
      i := i + 1; j := j − 1
    end; {if}
  end; {while}
end; {Urejanje2}
```

Še en način, kako urediti celotno tabelo v enem samem prehodu, pa je ta, da pred seboj „odrivamo“ vse vrednosti true, ki smo jih dotlej že našli, ko pa pridemo mimo kakšne vrednosti false, jo premaknemo nazaj za vse doslej najdene vrednosti true.[115]

```
procedure Urejanje3;
var i, j: integer;
begin
  i := 1; while i <= n do if Tabela[i] then break else i := i + 1;
  j := i; i := i + 1;
  while i <= n do begin
    { Invarianta: Tabela[1..j − 1] = false, Tabela[j..i − 1] = true. }
    if not Tabela[i] then
      begin Tabela[j] := false; Tabela[i] := true; j := j + 1 end;
    i := i + 1;
  end; {while}
end; {Urejanje3}
```

[115] Gl. npr. Wikipedia *s. v.* Several Unique Sort.

Za poskus smo izmerili čas, potreben za urejanje tabele 10^8 elementov. Naslednja tabela kaže povprečne čase po desetih urejanjih:

Začetno stanje tabele	Urejanje	Urejanje2	Urejanje3
tabela naključnih vrednosti	2,00 s	1,48 s	1,74 s
padajoče urejena tabela	1,52 s	1,31 s	1,26 s
naraščajoče urejena tabela	1,52 s	0,67 s	0,73 s
same vrednosti true	1,49 s	0,67 s	0,75 s
same vrednosti false	1,55 s	0,67 s	0,73 s

V praksi je mogoče še najzanimivejši primer tabela z naključnimi podatki, na kateri je, kot vidimo, najhitrejša rešitev Urejanje2, ki je približno 25 % hitrejša od Urejanje; Urejanje3 pa je približno na pol poti med njima.

R2003.X.7 Stikala

N: 626 Naš podprogram po vrsti pregleduje stikala in pri tem šteje, koliko je odkljukanih. Če to število preseže največje dovoljeno število hkrati odkljukanih stikal, bomo eno od odkljukanih stikal izključili. Naloga nič ne določa, katero naj izključimo, zato naj bo to kar zadnje odkljukano stikalo, na katero pri pregledovanju stikal naletimo. Pazimo le na to, da ne bomo izključili tistega stikala, ki ga je uporabnik ravnokar vključil, ampak raje kakšno drugo (razen če je MaxHkratiOdkljukanih enako 0).

```
procedure KlicanObSpremembi(StStikala: integer);
var i, StOdkljukanih, NekoOdkljukano: integer;
begin
  { Če je uporabnik stikalo izključil, zdaj gotovo ni vključenih preveč stikal. }
  if not JeOdkljukano(StStikala) then exit;
  { Preštejmo odkljukana stikala. }
  StOdkljukanih := 0; NekoOdkljukano := StStikala;
  for i := 1 to StStikal do if JeOdkljukano(i) then begin
    if i <> StStikala then NekoOdkljukano := i;
    StOdkljukanih := StOdkljukanih + 1;
    if StOdkljukanih > MaxHkratiOdkljukanih then break;
  end; {for i, if}
  { Izključimo kakšno stikalo, če je to potrebno. }
  if StOdkljukanih > MaxHkratiOdkljukanih then
    PostaviStikalo(NekoOdkljukano, false);
end; {KlicanObSpremembi}
```

R2003.X.8 Nabori znakov

Če je vsak znak predstavljen z nekim celim številom, je niz, ki bi ga radi izpisali, pravzaprav zaporedje takšnih celih števil, recimo $s = s_1 s_2 \ldots s_n$. Pisave pa lahko predstavimo z množicami števil, ki povedo, katere znake vsebuje posamezna pisava. Recimo, da je pisav m in označimo jih s $P_1, P_2, \ldots, P_m$. N: 627

Naloga sprašuje, kako razdeliti s na čim manj podnizov, pri čemer mora za vsak podniz obstajati kakšna pisava, ki vsebuje vse njegove znake. Pri vsakem znaku s-ja se lahko vprašamo, ali je pametno pri njem začeti nov podniz ali pa nadaljevati dosedanjega; načeloma je to odvisno od tega, s katero pisavo ta dosedanji podniz pišemo, saj mogoče trenutnega znaka sploh ni mogoče pisati z njo. Zato si zastavimo podprobleme oblike: kakšno je najmanjše število podnizov, na katere je treba razdeliti niz $s_1 s_2 \ldots s_i$, tako da se bo dalo vsak podniz v celoti izpisati z eno pisavo in da bo mogoče zadnji podniz izpisati s pisavo P_j? Recimo temu najmanjšemu številu podnizov kar $f(i,j)$. Takoj opazimo, da če $s_i \notin P_j$, je podproblem sploh nerešljiv in si lahko mislimo, da je $f(i,j) = \infty$. Tudi pri $i = 1$, torej izpisovanju prvega znaka, je rešitev na dlani: enega znaka sploh ne moremo razdeliti na več podnizov, zato je $f(1,j) = 1$ (če je seveda $s_1 \in P_j$). Pri kasnejših znakih, recimo pri s_i, pa moramo pregledati obe možnosti: (1) če začnemo tu nov podniz, bo skupno število podnizov enako $1 + f(i-1,k)$, če je k pisava, s katero smo končali dosedanji podniz (tisti, ki se končuje z znakom s_{i-1}). Ker smo začeli pri s_i nov podniz, nam je čisto vseeno, s katero pisavo smo izpisovali prejšnji niz, zato bomo vzeli tisti k, pri katerem je $f(i-1,k)$ najmanjša, saj hočemo čim manj podnizov. (2) Če pa tu ne začnemo novega podniza, ostane število podnizov enako kot pri prejšnjem znaku, torej $f(i-1,j)$. Od teh dveh možnosti moramo izbrati najmanjšo. Tako vidimo:

$$f(1,j) = \begin{cases} 1, & \text{če } s_1 \in P_j \\ \infty & \text{sicer.} \end{cases}$$

$$f(i,j) = \begin{cases} \min\{f(i-1,j), 1 + \min\{f(i-1,k) : 1 \le k \le m\}\}, & \text{če } s_i \in P_j \\ \infty & \text{sicer.} \end{cases}$$

Opazimo lahko lepo lastnost, da $\min f(i-1,k)$ po k ni nič odvisen od j, zato ga bomo lahko izračunali le enkrat in ga potem uporabljali pri računanju vrednosti $f(i,j)$ za vse možne j.

Po gornjih formulah bi lahko računali vrednosti $f(i,j)$ z rekurzivnim podprogramom, vendar bi bilo to neučinkovito, ker bi tako mnoge podprobleme reševali po večkrat. Najmanj, kar bi morali narediti, je to, da že izračunane vrednosti $f(i,j)$ odlagamo v neko tabelo in jih kasneje, če jih spet potrebujemo, preberemo od tam in jih ne računamo še enkrat (temu pristopu pravimo „pomnjenje“ ali „memoizacija“). Lahko pa tudi opazimo, da za izračun vrednosti $f(i,\cdot)$ potrebujemo le vrednosti $f(i-1,\cdot)$, zato lahko funkcijo f računamo

čisto sistematično z dvema gnezdenima zankama, zunanjo po naraščajočih i in notranjo po pisavah j (dinamično programiranje).

Če nas na koncu zanima tudi, kje točno so meje med podnizi, si moramo pri računanju vrednosti $f(i,j)$ zapomniti, kako smo prišli do vsakokratnega minimuma.

Razmislimo še o podatkovnih strukturah, ki bi jih uporabljal naš program. Niz s lahko predstavimo s tabelo; tudi vrednosti $f(i,j)$ lahko hranimo v tabeli. Za predstavitev pisav je več možnosti; predvsem si želimo, da bi se dalo čim hitreje pregledati vse pisave, s katerimi je mogoče napisati nek dani znak. Če imamo za vsako pisavo le seznam vseh znakov v njej, bo treba pregledati celoten seznam, kar bo šlo prepočasi; če bi imeli naraščajoče urejen seznam, shranjen v tabeli, bi ga lahko preiskali z bisekcijo, kar bi bilo že precej bolje. Uporabili bi lahko tudi razpršeno tabelo in tako še hitreje preverjali, ali je nek znak v pisavi prisoten ali ne. Lahko pa bi ustvarili tudi „obrnjen indeks“ (*inverted index*), v katerem bi imeli za vsak znak pripravljen seznam vseh pisav, ki ga vsebujejo. Slednje je za naše potrebe še najbolj učinkovito, saj se bomo tako lahko pri vsakem znaku (vsakem i) ukvarjali le s tistimi pisavami (tistimi j), ki ga res vsebujejo.

Naloga omenja tudi alternativni problem: poiskati najmanjše število različnih pisav, potrebnih za izpis vseh znakov niza s. To je pravzaprav le malo drugače zapisan problem pokrivanja množic (*set covering*), ki je znan NP-težak problem in zanj ne poznamo učinkovitih algoritmov, ki bi dajali optimalne rešitve (v našem primeru: poiskali najmanjše potrebno število pisav). Lahko bi napisali rešitev na podlagi rekurzije in načela „razveji in omeji“ (*branch and bound*), ki pa bi utegnila pri velikem številu pisav porabiti nesprejemljivo mnogo časa. Lahko pa z učinkovitejšimi algoritmi pridemo do rešitev, ki sicer ne bodo nujno optimalne, bo pa vsaj znano, za največ koliko so slabše od optimalnih (Johnsonov aproksimacijski algoritem (gl. op. na str. 70) za pokrivanje množic zagotavlja, da ne bi porabili več kot $(1+\lg n)$-krat toliko pisav kot optimalna rešitev). Z raznimi hevristikami, lokalno optimizacijo, simuliranim ohlajanjem ipd. lahko poskusimo potem takšne rešitve še izboljšati.

R2003.X.9 Hiperkocka in mreža

N: 627

Do elegantne rešitve lahko pridemo s pomočjo Grayevega koda. To je način, kako našteti vsa n-bitna števila v takšnem vrstnem redu, da se po dve sosednji števili razlikujeta v natanko enem bitu.[116] Za prvih nekaj vrednosti n je

[116] Imenuje se po Franku Grayu, ki je takšno kodiranje uporabil v nekem patentu leta 1953 (podobne stvari so bile sicer znane že tudi prej). Grayevih kodov je več (pravzaprav zelo veliko, gl. npr. zaporedji A091299 in A006069 v *The On-Line Encyclopedia of Integer Sequences*); tu opisana različica je tista, na katero v praksi najpogosteje naletimo. Posplošimo jih lahko tudi na druge številske sestave (naštevanje števil od 0 do $b^n - 1$ tako, da

Grayev kod takšen:

$$\begin{aligned} &n = 1 : 0, 1 \\ &n = 2 : 00, 01, 11, 10 \\ &n = 3 : 000, 001, 011, 010, 110, 111, 101, 100 \end{aligned}$$

Grayev kod za n-bitna števila dobimo tako, da vzamemo kod za $(n-1)$-bitna števila, nato pa še isti kod v obratnem vrstnem redu; številom v prvi polovici tako dobljenega zaporedja pripišemo na levi še eno ničlo, številom v drugi polovici pa enico. Ni se težko prepričati, da se zdaj res vsako n-bitno število pojavlja v tem zaporedju natanko enkrat in da se po dve sosednji števili razlikujeta v natanko enem bitu. Naj bo $g_n(i)$ položaj (med 0 in 2^n-1) števila i v Grayevem kodu za n-bitna števila.

Preimenujmo v naši hiperkocki H_n vsako točko u v $(u \operatorname{div} 2^{n-m}, u \bmod 2^{n-m})$. Za x-koordinato torej uporabimo zgornjih m bitov števila u, za y-koordinato pa ostalih $n-m$ bitov. Zdaj obstaja povezava med (x, y) in (x', y') natanko v primeru, če se točki v eni od koordinat ujemata, v drugi pa se razlikujeta natanko v enem bitu.

Preimenujmo točke še enkrat tako, da bivša (x, y) po novem postane $(g_m(x), g_{n-m}(y))$. Recimo, da se po tem preimenovanju neki točki ujemata v eni od koordinat, v drugi pa se razlikujeta za 1, npr. (x, y) in $(x \pm 1, y)$. Zaradi lastnosti Grayevega koda sta morali biti x-koordinati teh dveh točk pred preimenovanjem dve taki števili, ki se razlikujeta le v enem bitu, y-koordinati pa sta bili pri obeh točkah enaki, torej obstaja v grafu med tema dvema točkama povezava. Podobno bi razmišljali, če bi se točki ujemali v x-koordinati in se v y-koordinati razlikovali za 1. Tako torej vidimo, da ima graf po dosedanjih preimenovanjih že točke s prav takimi oznakami kot mreža $M_{2^m,2^{n-m}}$, obenem pa vsebuje že tudi vse njene povezave. Zdaj ni treba drugega, kot da pregledamo vse povezave našega grafa in pobrišemo tiste, ki jih na mreži ni (to bodo povezave, ki povezujejo po dve točki, ki se v eni koordinati sicer ujemata, v drugi pa se razlikujeta za več kot 1).

Slabost opisane naloge je, da je precej lahka, če reševalec že pozna Grayev kod, drugače pa je težje priti do elegantne rešitve.

Za konec si oglejmo še, kako lahko računamo $g_n(t)$. Če se n-bitno število t začne v dvojiškem zapisu na ničlo, je v prvi polovici zaporedja, ki ga določa Grayev kod, sicer pa v drugi polovici; vsaka od teh dveh polovic zase je pravzaprav le malo dopolnjen (in v primeru druge polovice še obrnjen) Grayev kod za $(n-1)$-bitna števila, torej lahko natančen položaj števila t ugotovimo s

---

se dve zaporedni števili razlikujeta v natanko eni števki, če ju zapišemo v b-iškem zapisu); gl. tudi str. 90.

pomočjo njegovega položaja v Grayevem kodu za $(n-1)$-bitna števila:

$$g_0(0) = 0$$
$$g_n(t) = \begin{cases} g_{n-1}(t), & \text{če } t < 2^{n-1} \\ 2^n - 1 - g_{n-1}(t \bmod 2^{n-1}), & \text{sicer.} \end{cases}$$

Pri n-bitnem številu t je $t \bmod 2^{n-1})$ preprosto t brez svojega zgornjega bita (bita $n-1$). Vrednost $2^n - 1 - g_{n-1}(t \bmod 2^{n-1})$ pa ni nič drugega kot n-bitno število, ki ima zgornji bit prižgan, na spodnjih $n-1$ bitih pa ima vrednost $g_{n-1}(t \bmod 2^{n-1})$, v kateri so vsi biti ravno obrnjeni. Ker se $g_{n-1}(t \bmod 2^{n-1})$ seveda lahko računa po enakem pravilu, lahko rezultat računamo tudi bit za bitom in si pri tem le zapomnimo, koliko obračanj bita so zahtevali višji nivoji rekurzije. Če bi imeli števili t in $g_n(t)$ eksplicitno predstavljeni kot tabeli bitov, bi lahko postopek zapisali takole:

```
StObracanj := 0;
for i := n − 1 downto 0 do begin
  if Odd(StObracanj) then g[i] := t[i] xor 1 else g[i] := t[i];
  StObracanj := StObracanj + t[i];
end; {for i}
```

Vidimo lahko, da je dovolj, če si zapomnimo le, ali je število obračanj sodo ali liho. Namesto da bi StObracanj povečevali za t[i], je dovolj, če jo z njim **xor**amo, tako da bo StObracanj hranila pravzaprav le spodnji bit pravega števila obračanj; zato lahko obračanje zdaj namesto s stavkom **if** zapišemo kar kot g[i] := t[i] **xor** StObracanj. Zdaj pa vidimo, da med g[i] in StObracanj ni nobene razlike več in naš postopek se poenostavi v:

```
g[n − 1] := t[n − 1];
for i := n − 2 downto 0 do
  g[i] := t[i] xor g[i + 1];
```

Iz te zanke lahko tudi takoj opazimo, da je g[i] pravzaprav preprosto **xor** vseh bitov t[i..n − 1]. To pa lahko računamo še bolj elegantno z zamikanjem:

```
g := t xor (t shr 1);
g := g xor (g shr 2);
g := g xor (g shr 4);
g := g xor (g shr 8);
...
```

Po prvi vrstici imamo v vsakem bitu g-ja **xor** dveh zaporednih bitov t-ja; po drugi vrstici imamo zato v bitu g[i] že **xor** štirih bitov t[i..i + 3], po tretji v vsakem že **xor** osmih bitov t-ja in tako naprej. Štiri vrstice zadostujejo za n-je do 16; v splošnem potrebujemo $\lceil \lg n \rceil$ vrstic, če dekodiramo n-bitni Grayev kod.

R2004.X.1 Graf signala

V zanki se bomo istočasno premikali po x-koordinatah zaslona (od 0 do ZaslonX − 1) in po vrednostih iz tabele Vrednosti (indeksi gredo od 0 do StVrednosti − 1. Če je vrednosti več kot x-koordinat, se (kot zahteva naša naloga) premaknemo v vsakem koraku vsaj za en piksel v desno: x-koordinato povečamo za 1 in izračunamo, katero vrednost bi bilo najprimerneje prikazati na novi x-koordinati. Če pa je vrednosti manj kot x-koordinat, se premaknemo v vsakem koraku za eno vrednost naprej po tabeli Vrednosti in izračunamo, na kateri x-koordinati bi bilo treba prikazati to vrednost. N: 627

Formula za preračun med x-koordinatami in indeksi v tabeli Vrednosti temelji na ideji, naj skrajna leva x-koordinata ($x = 0$) ustreza prvi vrednosti (indeks 0 v tabeli Vrednosti), skrajna desna x-koordinata (torej ZaslonX − 1) pa zadnji vrednosti (torej StVrednosti − 1); vmes pa naj x-koordinate in indeksi vrednosti naraščajo premosorazmerno (če smo prehodili že polovico zaslona, hočemo videti vrednost na polovici tabele; ipd.). Tako na primer za vrednost z indeksom i vidimo, da je na $i/(\text{StVrednosti} - 1)$ poti od začetka do konca tabele (npr. na pol poti, če je i ravno indeks srednjega elementa tabele), zato mu ustreza koordinata na $i/(\text{StVrednosti} - 1)$ poti od levega do desnega roba zaslona; to pa je $x = (\text{ZaslonX} - 1) \cdot i/(\text{StVrednosti} - 1)$. Na enak način lahko pridemo tudi do formule za preračun x-koordinat v indekse. Ker tidve formuli ne dasta nujno celoštevilskih rezultatov, jih moramo pred uporabo še zaokrožiti do najbližjega celega števila (saj zahteva podprogram NarisiDaljico celoštevilske koordinate, indeksi v tabelo Vrednosti pa morajo biti tudi celoštevilski).

```
procedure NarisiSignal;
var i, x, y, x2, y2: integer;
begin
  x := 0; i := 0; y := Vrednosti[0];
  while (i < StVrednosti − 1) and (x < ZaslonX − 1) do begin
    if StVrednosti > ZaslonX then begin
      x2 := x + 1;
      i := Round(x2 / (ZaslonX − 1) * (StVrednosti − 1));
    end else begin
      i := i + 1;
      x2 := Round(i / (StVrednosti − 1) * (ZaslonX − 1));  { (†) }
    end; {if}
    y2 := Vrednosti[i];
    NarisiDaljico(x, y, x2, y2);
    x := x2; y := y2;
  end; {while}
end; {NarisiSignal}
```

Ena od slabosti takšnega prikazovanja signala je (kot pravi že besedilo naloge), da se lahko marsikaj zanimivega izgubi, če je število vrednosti veliko večje od

vodoravne ločljivosti zaslona. Pri vsaki x-koordinati prikažemo le eno vrednost, med x in $x+1$ pa mogoče mnogo vrednosti izpustimo; če se tam vmes zgodi v signalu kaj zanimivega, uporabnik o tem ne bo izvedel ničesar. Možna izboljšava našega podprograma bi lahko pri vsaki x-koordinati pregledala vse vrednosti signala, ki se po formuli iz vrstice (†) preslikajo v to x-koordinato; na grafu bi potem lahko prikazali razne statistične lastnosti te skupine vrednosti, na primer njihovo povprečje, srednjo vrednost (mediano), minimum, maksimum, itd.

R2004.X.2 Ribe

N: 628

Ker ribe vedno lahko požirajo le svoje sosede, tvorijo požrte ribe v vsakem trenutku neko strnjeno podzaporedje prvotnega zaporedja, trenutna riba pa je bodisi tista neposredno levo ali pa tista neposredno desno od požrtega podzaporedja. Vsota, ki bi jo radi maksimizirali, je vsota velikosti trenutne ribe in vseh požrtih.

Do največje vsote lahko pridemo s preprostim požrešnim algoritmom. Če trenutna riba nima nobene sosede, ki bi jo lahko požrla, se naš postopek ustavi; če ima eno, jo ta soseda pač požre; če pa ima dve sosedi in bi jo lahko požrla katerakoli od njiju, naj jo požre manjša izmed teh dveh sosed (oz. poljubna, če sta obe enako veliki). Večja izmed teh dveh sosed lahko kasneje še vedno požre tisto manjšo, obratno pa ne bi šlo.

Spodnji podprogram izračuna največjo vsoto, ki jo lahko dobimo, če začnemo pri ribi z. Trenutna riba in vse že požrte tvorijo neko strnjeno podzaporedje rib od L do R, vsota njihovih velikosti pa je v spremenljivki Vsota. Velikost trenutne ribe hranimo v spremenljivki T.

```
function NajboljsaVsota(Z: integer): integer;
var L, R, T, Vsota, aL1, aR1: integer;
begin
  L := Z; R := Z; T := A[Z]; { Trenutna riba je Z, požrta ni še nobena. }
  Vsota := T;
  while (L > 0) or (R < N) do begin
    { Trenutna riba je bodisi L ali pa R, vse ostale iz podzaporedja L..R pa so
      že požrte. Trenutno ribo lahko torej požre bodisi riba L - 1 ali pa riba R + 1.
      Poglejmo, kako veliki sta tidve ribi; če smo že na robovih zaporedja,
      si mislimo, da sta tam neki majhni ribi, ki trenutne ne moreta požreti. }
    if L > 1 then aL1 := A[L - 1] else aL1 := T - 1;
    if R < N then aR1 := A[R + 1] else aR1 := T - 1;
    { Mogoče nas ne more požreti nobena od sosed. }
    if (aL1 < T) and (aR1 < T) then break;
    { Če nas lahko požre le ena, naj nas pač tista;
      če obe, pa naj nas požre manjša od njiju. }
    if (aR1 < T) or ((T <= aL1) and (aL1 <= aR1))
```

```
    then begin Vsota := Vsota + aL1; L := L − 1; T := aL1 end
    else begin Vsota := Vsota + aR1; R := R + 1; T := aR1 end;
  end; {while}
  NajboljsaVsota := Vsota;
end; {NajboljsaVsota}
```

Do najboljše vsote sploh bi prišli, če bi poklicali NajboljsaVsota(Z) za vse z od 1 do n in si zapomnili največjo izmed dobljenih vsot. Podprogram NajboljsaVsota mora v najslabšem primeru pregledati vse ribe, zato ima časovno zahtevnost $O(n)$. Ker ga moramo klicati po enkrat za vsak z od 1 do n, je skupna časovna zahtevnost našega postopka $O(n^2)$. Prepričajmo se, da je ta rešitev res pravilna (dokaz je rahlo puščoben in se ga lahko brez velike škode preskoči); nato pa si bomo ogledali še eno učinkovitejšo rešitev.

Dokaz pravilnosti. Recimo, da je bilo nekaj rib že požrtih. To, kaj se bo lahko dogajalo v nadaljevanju, je odvisno le od tega, katere ribe so bile doslej požrte in katera je trenutna, neodvisno pa je od tega, pri kateri ribi smo začeli in v kakšnem vrstnem redu so se ribe žrle med sabo. Označimo torej trenutno stanje s trojico (l, r, t), $1 \le l \le r \le n$, ki nam pove, da so požrte vse ribe od l do r razen ene od rib l in r, ki pa je trenutna in ima velikost t (torej je t ali enako a_l ali pa enako a_r). (Spomnimo se, da že požrte ribe vedno tvorijo neko strnjeno podzaporedje vseh rib — v našem primeru je to ali $l+1, \ldots, r$ ali pa $l, \ldots, r-1$, odvisno pač od tega, ali je trenutna riba l ali riba r.)

Iz stanja (l, r, t) se lahko premaknemo v $(l-1, r, a_{l-1})$, če je $l > 1$ in $a_{l-1} \ge t$, ali pa v $(l, r+1, a_{r+1})$, če je $r < n$ in $a_{r+1} \ge t$. Vsota, ki jo skušamo maksimizirati, se v prvem primeru poveča za a_{l-1}, v drugem pa za a_{r+1}. Naloga pravzaprav sprašuje, kako priti do stanja s čim večjo vsoto, če začnemo v stanju (z, z, a_z) za nek z, $1 \le z \le n$.

Če v nekem stanju ni mogoč noben premik naprej, se moramo pač ustaviti; če je mogoč en sam premik, gremo pač po njem (ker velikosti rib ne morejo biti negativne, se nam ne splača ustaviti predčasno). Kaj pa, če sta mogoča oba premika — z drugimi besedami, če lahko trenutno ribo z velikostjo a požreta obe njeni sosedi, tako a_{l-1} kot a_{r+1}? Zgoraj smo zatrdili, da je bolje, če trentuno ribo požre manjša od obeh sosed. O tem se lahko prepričamo takole.

Recimo, da je $a_{l-1} \le a_{r+1}$ (obe pa sta seveda $\ge t$). Ali bo kaj narobe, če v naslednjem koraku trenutno ribo požre riba $l-1$, ne pa riba $r+1$? Naj bo $(l^\star, r^\star, t^\star)$ neko najboljše stanje, ki ga je iz trenutnega (l, r, t) sploh še mogoče doseči. To, da v naslednjem koraku žre riba $l-1$, je narobe le v primeru, če iz stanja $(l-1, r, a_{l-1})$ (v katerega bi po tem žretju prišli) ne moremo več priti v stanje $(l^\star, r^\star, t^\star)$ (niti v nobeno drugo enako dobro stanje). Prepričajmo se, da se to ne more zgoditi.

Ker se požrto podzaporedje ob nadaljnjih premikih lahko le širi, mora biti $l^\star \le l$ in $r^\star \ge r$. Recimo, da je $l = l^\star$; pot od trenutnega stanja (l, r, a) do $(l^\star, r^\star, t^\star)$ so torej sestavljala sama žretja z desne: $r+1$ je požrla trenutno

ribo (z velikostjo t), nato je riba $r+2$ požrla ribo $r+1$ in tako naprej. Toda ker je $a_{l-1} \geq t$ in hkrati $a_{l-1} \leq a_{r+1}$, bi lahko to pot popravili tako, da bi najprej $l-1$ požrla trenutno ribo, nato bi $r+1$ požrla ribo $l-1$, nadaljevalo pa bi se tako kot prej. S tem bi prišli namesto do stanja $(l, r^\star, t^\star)$ do stanja $(l-1, r^\star, t^\star)$, ki je vsaj tako dobro (pravzaprav še boljše; njegova vsota je večja za a_{l-1}). Torej s tem, ko smo začeli z žretjem z leve, nismo bili na koncu nič na slabšem, kvečjemu na boljšem.

Druga možnost je, da je $l^\star < l$. Mislimo si pot od (l, r, t) do $(l^\star, r^\star, t^\star)$. Če se ta pot začne z žretjem z leve, se iz (l, r, t) premaknemo v $(l-1, r, a_{l-1})$ in je torej $(l^\star, r^\star, t^\star)$ iz tega stanja še vedno dosegljivo; prav to smo hoteli dokazati. Če pa se pot od (l, r, t) do $(l^\star, r^\star, t^\star)$ začne z nekaj (recimo k) žretji z desne (kar nas pripelje do stanja $(l, r+k, a_{r+k})$), mora v njej vendarle prej ali slej nastopiti prvo žretje z leve (vsaj eno žretje z leve namreč potrebujemo, da bomo prišli od l do $l^\star$, ki je $< l$): takrat bo riba $l-1$ požrla ribo $r+k$ in prišli bomo v stanje $(l-1, r+k, a_{l-1})$. V prejšnjih korakih je riba $r+k$ požrla ribo $r+k-1$, ta še prej ribo $r+k-2$, …, riba $r+2$ je požrla ribo $r+1$, ta pa prej trenutno ribo (z velikostjo t). Vse to pomeni, da je $a_{l-1} \geq a_{r+k} \geq \ldots \geq a_{r+1}$; po drugi strani pa smo na začetku rekli, da je $a_{l-1} \leq a_{r+1}$. Torej so vse te ribe v resnici enako velike: $a_{l-1} = a_{r+1} = a_{r+2} = \ldots = a_{r+k}$. Potem pa ni nič narobe, če našo pot preuredimo tako, da najprej izvedemo žretje z leve, nato pa k žretij z desne; ker so vse vpletene ribe enako velike, je to izvedljivo in nas popelje v stanje $(l-1, r+k, a_{r+k})$. Ker je $a_{r+k} = a_{l-1}$, je to stanje isto kot stanje $(l-1, r+k, a_{l-1})$, do katerega smo prišli prej; našo pot do optimalnega stanja se bo torej dalo nadaljevati na enak način kot prej. Tako torej vidimo, da je stanje $(l^\star, r^\star, t^\star)$ vsekakor še naprej dosegljivo, če v trenutnem stanju izvedemo najprej žretje z leve.

Doslej smo razmišljali o možnosti, da je $a_{l-1} \leq a_{r+1}$; če velja $\geq$ namesto $\leq$, bi lahko z analognim razmislekom ugotovili, da je pametno najprej izvesti žretje z desne. V vsakem primeru torej vidimo, da je v primeru, ko lahko trenutno ribo požreta obe sosedi, najbolje, če jo najprej požre manjša od njiju.

Učinkovitejša rešitev. Naša gornja rešitev je imela časovno zahtevnost $O(n^2)$; oglejmo si zdaj primer rešitve z zahtevnostjo $O(n)$. Recimo, da bi začeli pri neki ribi z in potem po vrsti izvedli nekaj žretij z ribami $i_1, i_2, \ldots, i_k$. Med temi ribami jih je mogoče nekaj, ki ležijo v prvotnem zaporedju levo od z; ker ribo vedno požre njena soseda, se mora v tem zaporedju rib, ki so žrle, najprej pojaviti $z-1$, nekje kasneje mogoče $z-2$, še kasneje $z-3$ in tako naprej. Ker manjša riba ne more požreti večje, vidimo, da se v našem zaporedju rib, ki so žrle, lahko pojavi največ toliko rib levo od z-ja, dokler njihove velikosti še ne začnejo padati. Podoben razmislek velja za ribe, ki ležijo v prvotnem zaporedju desno od z.

Recimo, da je riba i „levi maksimum“, če je strogo večja od svoje leve sosede (ali pa leve sosede sploh nima, torej $i = 1$), in „desni maksimum“, če

je strogo večja od svoje desne sosede (ali pa je sploh nima, torej $i = n$). Naj bo l_z najbolj desna riba, ki je levi maksimum in ni desno od z; naj bo r_z najbolj leva riba, ki je desni maksimum in ni levo od z. V prejšnjem odstavku smo videli, da če začnemo pri ribi z, bodo v vsoto, ki jo iščemo, vključene v najboljšem primeru ribe od l_z do r_z. Vse te ribe pa tudi res lahko vključimo v našo vsoto. Zaporedje velikosti $a_{z-1}, a_{z-2}, \ldots, a_{l_z}$ je nepadajoče, enako tudi zaporedje $a_{z+1}, a_{z+2}, \ldots, a_{r_z}$; zdaj lahko v mislih ti dve zaporedji zlivamo: če je trenutni člen prvega manjši od trenutnega člena drugega, se premaknemo naprej po prvem zaporedju in izvedemo žretje z leve, sicer pa se premaknemo naprej po drugem in izvedemo žretje z desne. Očitno je, da pri tem delamo same dovoljene korake: riba vedno žre svojo sosedo in to tako, ki ni večja od nje.

Če torej za neko z poznamo l_z in r_z, že tudi vemo, kaj bi nam vrnila funkcija NajboljsaVsota(z): vrnila bi vsoto $a_{l_z} + a_{l_z+1} + \ldots + a_{r_z-1} + a_{r_z}$. Lepo pri tem je, da ko enkrat poznamo l_z in r_z, ni težko priti do l_{z+1} in r_{z+1}. Definicija pravi, da je l_{z+1} najbolj desni levi maksimum, ki še ni desno od $z+1$. Mi pa vemo, da je en levi maksimum pri l_z in da nato vse do vključno z ni nobenega (sicer bi bil l_z šele tam); potem pa ostane le še $z+1$, ostale ribe so že desno od $z+1$. Torej je l_{z+1} bodisi enak l_z bodisi enak $z+1$, odvisno pač od tega, ali je $z+1$ levi maksimum ali ni. Podobno lahko razmišljamo pri r_{z+1}. En desni maksimum je pri r_z, med njim in z ni nobenega, tisti levo od z-ja pa so tudi levo od $z+1$ in za nas ne pridejo v poštev. Torej je $r_{z+1} = r_z$, če le ne leži levo od $z+1$; toda to je možno le v primeru, da je $r_z = z$: takrat moramo pač poiskati prvi naslednji desni maksimum.

Torej, ko se z povečuje, se tudi l_z in r_z povečujeta. Pri vsakem z-ju moramo izračunati vsoto velikosti rib $a_{l_z} + a_{l_z+1} + \ldots + a_{r_z-1} + a_{r_z}$; če se l_z poveča za 1, vsota na levi strani izgublja seštevance, če se r_z poveča za 1, pa jih na desni strani pridobiva; v vsakem primeru je torej ni težko popravljati. Zato imamo, ko pregledamo vse z-je od 1 do n, z računanjem novih vrednosti l_z in r_z ter popravljanjem vsot vsega skupaj le $O(n)$ dela.

```
function NajboljsaVsota2: integer;
var Z, LZ, RZ, Vsota, NajVsota, S: integer;
begin
  LZ := 1; RZ := 0; Vsota := 0; NajVsota := 0;
  for Z := 1 to N do begin
    { Trenutno je v LZ vrednost l_{z-1}, v RZ pa r_{z-1}. Izračunajmo novi LZ = l_z. }
    if Z = 1 then S := A[Z] - 1 else S := A[Z - 1];
    if S < A[Z] then { Z je levi maksimum; povečajmo LZ do Z. }
      while LZ < Z do begin Vsota := Vsota - A[LZ]; LZ := LZ + 1 end;
    { Izračunajmo novi RZ = r_z. }
    if RZ < Z then { Z - 1 je bil desni maksimum, }
      repeat        { zdaj pa moramo najti naslednjega. }
        RZ := RZ + 1; Vsota := Vsota + A[RZ];
```

```
        if RZ = N then S := A[RZ] − 1 else S := A[RZ + 1];
      until S < A[RZ];
    { Zdaj imamo v LZ pravo l_z, v RZ pa r_z. V spremenljivki Vsota je
      vsota velikosti vseh rib od LZ do RZ; mogoče je to najboljša vsota doslej. }
    if Vsota > NajVsota then NajVsota := Vsota;
  end; {for Z}
  NajboljsaVsota2 := NajVsota;
end; {NajboljsaVsota2}
```

R2004.X.3 Cezarjev kod

N: 628 Naj bo P naš znani podniz nekodiranega sporočila (parameter Znano), K pa naše kodirano sporočilo (parameter Kodirano). Najpreprostejša rešitev je najbrž ta, da poskusimo kodirati P z vsemi možnimi ključi (kličemo podprogram Kodiraj) in za vsakega od tako dobljenih kodiranih nizov preverimo, če se pojavlja kot podniz v kodiranem sporočilu K.

```
{ Če ne bi v nalogi pisalo, da sta podprograma StevilkaCrke in
  CrkaIzStevilke že dana, bi ju lahko napisali takole. }
function StevilkaCrke(Crka: char): integer;
  begin StevilkaCrke := Ord(Crka) − Ord('A') + 1 end;
function CrkaIzStevilke(Stevilka: integer): char;
  begin CrkaIzStevilke := Chr(Stevilka + Ord('A') − 1) end;

function Kodiraj(Niz: string; Kljuc: integer): string;
var Kodiran: string; i, Stev: integer;
begin
  Kodiran := '';
  for i := 1 to Length(Niz) do begin
    Stev := StevilkaCrke(Niz[i]) + Kljuc;
    if Stev > n then Stev := Stev − n;
    Kodiran := Kodiran + CrkaIzStevilke(Stev);
  end; {for i}
  Kodiraj := Kodiran;
end; {Kodiraj}

function Dekodiraj(Kodirano, Znano: string): integer;
var k: integer; KodZnano: string;
begin
  for k := 1 to n − 1 do begin
    KodZnano := Kodiraj(Znano, k);
    if JePodniz(Kodirano, KodZnano) > 0 then
      begin Dekodiraj := k; exit end;
  end; {for k}
  Dekodiraj := 0; { Primernega ključa sploh ni. }
end; {Dekodiraj}
```

Preverjanja, ali je KodZnano res podniz niza Kodirano, se lahko lotimo na različne načine. Najpreprosteje bo, če po vrsti pregledujemo vse možne položaje podniza v nizu in preverjamo, če se naš podniz res pojavlja na tistem mestu v nizu. Vsak znak podniza mora biti enak istoležnemu znaku niza.

```
function JePodniz(Niz, Podniz: string): boolean;
var i, j: integer;
begin
  for i := 1 to Length(Niz) − Length(Podniz) + 1 do begin
    { Poglejmo, če je Podniz enak Niz[i..i + Length(Podniz) − 1]. }
    j := 0;
    while j < Length(Podniz) do
      if Niz[i + j] = Podniz[j + 1] then j := j + 1 else break;
    { Podniz in Niz[i..i + Length(Podniz) − 1] se ujemata v prvih j znakih. }
    if j = Length(Podniz) then begin JePodniz := true; exit end;
  end; {while}
  JePodniz := false;
end; {JePodniz}
```

Kakšna je časovna zahtevnost te naše rešitve? Naj bo $|P|$ dolžina niza P, $|K|$ pa dolžina niza K. Za kodiranje P-ja z vsemi možnimi ključi smo porabili $O(n|P|)$ časa. Naj bo P_k niz, ki ga dobimo po kodiranju P-ja s ključem k. Ko podprogram JePodniz primerja P_k z nizom $K[i..i + |P| - 1]$, izvede recimo j_{ki} iteracij svoje notranje zanke **while**. Skupno število iteracij te zanke (po vseh klicih podprograma JePodniz) je torej $J := \sum_{k=1}^{n-1} \sum_{i=1}^{|K|-|P|+1} j_{ki}$. Opazimo lahko naslednje: recimo, da ob primerjanju niza P_k z nizom $K[i..i + |P| - 1]$ ugotovimo ujemanje prvih znakov, torej da je $P_k[1] = K[i]$. Če bi P kodirali s kakšnim drugim ključem, recimo s k' namesto s k, bi bil prvi znak dobljenega niza $P_{k'}$ vsekakor drugačen kot pri P_k, torej se prvi znak niza $P_{k'}$ gotovo ne bi ujemal s $K[i]$. Tako torej vidimo, da je od vrednosti $j_{1i}, j_{2i}, \ldots, j_{ki}$ lahko le ena večja od 1 (gotovo pa ni večja od $|P|$ — tako dolg je pač podniz, s katerim se ukvarjamo), ostale pa so enake 1 (razen če je $|P| = 0$; tedaj so vsi $j_{ki} = 0$). Torej je $\sum_{k=1}^{n-1} j_{ki} \leq (n-2) + |P|$. Tako dobimo $J = \sum_{i=1}^{|K|-|P|+1} \sum_{k=1}^{n-1} j_{ki} \leq |K|(n+|P|)$, kar pomeni, da ima naša rešitev časovno zahtevnost $O(|K|(n+|P|))$.

Obstajajo tudi učinkovitejše rešitve. Podprogram JePodniz bi lahko za iskanje podniza v nizu na primer uporabil Knuth-Morris-Prattov algoritem.[117] Ta ima najprej $O(|P|)$ dela za pripravo neke pomožne tabele, nato pa poišče podniz v nizu (ali pa ugotovi, da ga ni) v času $O(|K|)$. Skupna časovna zahtevnost vseh kodiranj in iskanj je tako le $O(n(|P| + |K|))$, kar je pravzaprav preprosto $O(n|K|)$, saj P gotovo ni daljši od K (oz. če je, lahko takoj rečemo, da primernega ključa ni).

[117] Gl. npr. Cormen *et al.*, *Introduction to Algorithms*, 34.4 v prvi izdaji, 32.4 v drugi.

Lahko pa bi si pri iskanju podnizov v nizu K pomagali z drevesom končnic (*suffix tree*).[118] To je podatkovna struktura, ki jo lahko za niz K zgradimo v $O(|K|)$ časa, nato pa v času $O(|P|)$ preverimo, če je nek P_k podniz K-ja. Tako bi porabili za celoten postopek le še $O(|K| + n|P|)$ časa. To utegne biti bolje od $O(n|K|)$ iz prejšnjega odstavka, če je P dovolj kratek v primerjavi s K.

R2004.X.4 Števila zveri

N: 630

Števila si predstavljajmo kot nize števk; v praksi bi jih shranjevali v tabelah. Oznaka xy naj pomeni stik nizov x in y, oznaka $x^{\underline{k}}$ pa stik k izvodov niza x. (Oznako x^k bomo uporabljali v običajnem pomenu, torej za k-to potenco števila x.) Števili a in b lahko razbijemo na posamezne števke in ju zapišemo kot $a = a_n a_{n-1} \ldots a_1 a_0$, $b = b_m b_{m-1} \ldots b_1 b_0$. Pri tem so a_0 enice, a_1 desetice in tako naprej. Ker je (kot pravi naloga) $a \geq b$, je $n \geq m$.

Naj bo c_i najmanjše naravno število, ki je $\geq a$ in vsebuje b kot podniz na mestih od i-tega do $(i+m)$-tega (mesta štejmo od desne proti levi; 0 so enice, 1 desetice, 2 stotice in tako naprej). (Na primer: za $a = 19345678$ in $b = 654$ imamo $c_0 = 19346\underline{654}$, $c_1 = 1934\underline{6540}$, $c_2 = 193\underline{6540}0$, $c_3 = 19\underline{6540}00$, $c_4 = 2\underline{6540}000$, $c_5 = \underline{654}00000$, $c_6 = \underline{654}000000$, itd.) Število, po katerem sprašuje naša naloga, je ravno najmanjši c_i po vseh možnih i.

Kako priti do c_i? Ta naj bi imel na mestih od i do $i+m$ vrednost b; oglejmo si, kakšne števke so na teh mestih v a-ju: recimo jim $d_i = a_{i+m} a_{i+m-1} \ldots a_i$. Če je $d_i = b$, se torej b pojavlja na teh mestih že v a-ju, zato je $c_i = a$. Če pa je $d_i \neq b$, lahko v mislih povečujemo a za 1 in gledamo, kaj se dogaja na mestih $i+m, \ldots, i$; ko tam dobimo vrednost b, se ustavimo in imamo c_i. No, ko povečujemo a za 1, se tudi d_i vsake toliko časa poveča za 1 (ko pride pri povečevanju a za 1 do prenosa z mesta $i-1$ na i; tik pred takšnim povečanjem je desnih i števk a-ja enakih $9^{\underline{i}}$, po njem pa $0^{\underline{i}}$), razen če je bil d_i pred trenutnim povečanjem enak $9^{\underline{m+1}} = 999\ldots9$ ($m+1$ devetic): v tem primeru pride do prenosa z mesta $i+m$ na $i+m+1$ in d_i pade na $0^{\underline{m+1}}$ ($m+1$ ničel).

Če je bila prvotna vrednost d_i manjša od b, bo pri povečevanju d_i dosegel b, še preden bo prišlo do tega padca na $0^{\underline{m+1}}$; c_i bo zato enak kar $a_n \ldots a_{i+m+1} b 0^{\underline{i}}$. Oglejmo si primer: $a = 12\underline{345}67$, $b = 987$, $i = 2$. Torej je d_i (podčrtani del a-ja) enak 345. Ko povečujemo a za 1, dobivamo $12\underline{345}68$, $12\underline{345}69$, …, $12\underline{345}99$, $12\underline{346}00$, $12\underline{346}01$, …, $12\underline{346}99$, $12\underline{347}00$, …, $12\underline{986}98$, $12\underline{986}99$, $12\underline{987}00$. Vidimo torej, da se d postopoma povečuje in to v vsakem takem trenutku, ko se števke desno od njega spremenijo iz samih devetic v same ničle. Ker je bil

[118] Ta drevesa je prvi predlagal Peter Weiner leta 1973 (*Linear pattern matching algorithms*, Proc. 14th Symp. on Switching and Automata Theory, pp. 1–11). Dandanes jih običajno gradimo z Ukkonenovim algoritmom (E. Ukkonen: *On-line construction of suffix trees*, Algorithmica, 14(3):249–260, September 1995). Gl. tudi R. Giegerich, S. Kurtz: *From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction*, Algorithmica, 19(3):331–353, November 1997.

d_i sprva manjši od b, se do trenutka, ko doseže d_i vrednost b, levo od njega v številu a ne zgodi še nič; desno pa so v tem prvem trenutku (takoj po zadnjem povečanju d_i) pač same ničle (v našem primeru dve ničli, ker je $i = 2$).

Če pa je bil d_i na začetku večji od b, se bo moral najprej povečati do $9^{\underline{m+1}}$, nato pasti na $0^{\underline{m+1}}$ in se nato povečati do b. Pri padcu d_i z $9^{\underline{m+1}}$ na $0^{\underline{m+1}}$ pa pride do prenosa z mesta a_{i+m} na a_{i+m+1}, torej se tisti del a-ja poveča za 1. Zato je v tem primeru $c_i = (a_n \ldots a_{i+m+1} + 1)b0^{\underline{i}}$. Oglejmo si isti primer kot prej: $a = 1234568$, $i = 2$, le za b vzemimo kakšno manjšo vrednost, recimo 210. Ko povečujemo a za 1, dobivamo 12$\underline{345}$68, 12$\underline{345}$69, …, 12$\underline{345}$99, 12$\underline{346}$00, 12$\underline{346}$01, …, 129$\underline{999}$8, 129$\underline{999}$9, 13$\underline{000}$00, …, 13$\underline{000}$99, 13$\underline{001}$00, …, 13$\underline{209}$98, 13$\underline{209}$99, 13$\underline{210}$00. Vidimo torej, da se mora d_i najprej povečati do samih devetic, nato pasti na same ničle (ob tem se del a-ja, ki leži levo od d, poveča za 1; v našem primeru se je povečal z 12 na 13), nato pa zrasti do b.

Zdaj znamo torej pri vsakem i določiti najmanjši tak c_i, ki je $\geq a$ in vsebuje b kot podniz od i-tega mesta naprej. Med c_i-ji, ki jih dobimo za različne vrednosti i, moramo vrniti najmanjšega (recimo mu $c^\star$).

```
function Naslednik(a_n a_{n-1} ... a_0, b_m b_{m-1} ... b_0);
begin
  c* := ∞;
  for i := 0 to n − m + 1 do begin
    d_i := a_{i+m} a_{i+m-1} ... a_i;
    if d_i = b then begin c* := a; break end;
    if d_i > b then c_i := (a_n a_{n-1} ... a_{i+m+1} + 1)b0^i
                else c_i := (a_n a_{n-1} ... a_{i+m+1})b0^i;
    if c_i < c* then c* := c_i;
  end; {for}
  Naslednik := c*;
end; {Naslednik}
```

Vrednost i je šla do $n - m + 1$ namesto do $n - m$, ker se lahko zgodi, da je iskani rezultat daljši od a; pri $a = 456$ in $b = 123$ je iskani rezultat na primer 1230. Pri $i = n - m + 1$ potrebujemo števko a_{n+1}; mislimo si, da je enaka 0. Večjih i nam ni treba pregledovati; ko se i povečuje od $n - m$ naprej, se c_i vsakič podaljša za eno ničlo na koncu, torej je desetkrat večji kot prej. Takih nima smisla pregledovati, saj nas zanima le najmanjši c_i.

Zgornji postopek predpostavlja, da znamo primerjati velika števila in povečevati število za 1. Oboje je enostavno; moramo se le zgledovati po tem, kako bi takšne reči počeli ročno. Ob primerjanju dveh števil bi gledali istoležne števke od bolj pomembnih proti manj pomembnim (torej od leve proti desni), dokler ne naletimo na mesto, kjer se števili v trenutni števki razlikujeta:

```
procedure Primerjaj(x_u x_{u-1} ... x_0, y_v y_{v-1} ... y_0);
begin
```

```
  for p := max{u, v} downto 0 do
    if x_p > y_p then begin WriteLn('x > y'); exit end
    else if x_p < y_p then begin WriteLn('x < y'); exit end;
  WriteLn('x = y');
end; {Primerjaj}
```

Če sta števili različno dolgi, ju v mislih podaljšajmo z ničlami: $x_p = 0$ za $p > u$ in $y_p = 0$ za $p > v$. Na primer: ko primerjamo $x = 123456$ in $y = 129876$, bi najprej primerjali 1 in 1 (pri $p = 5$), nato 2 in 2 (pri $p = 4$), nato pa 3 in 9 (pri $p = 3$) in iz dejstva, da je $3 < 9$, zaključili, da mora biti $x < y$.

Povečevanja števila za 1 pa se lotimo od desne proti levi (od manj pomembnih števk k bolj pomembnim). Spremenljivka c hrani prenos s prejšnjega mesta; temu prištejemo trenutno števko našega števila x; če je rezultat večji ali enak 10, se prenos nadaljuje še na naslednje mesto.

```
function PovecajZa1(x_u x_{u-1} ... x_0);
begin
  c := 1; i := 0;
  while i ≤ u or c > 0 do begin
    if i ≤ u then c := c + x_u;
    y_i := c mod 10; c := c div 10; i := i + 1;
  end; {while}
  PovecajZa1 := y_{i-1} ... y_0;
end; {PovecajZa1};
```

Kakšna je časovna zahtevnost naše rešitve? Pri vsakem i imamo $O(m)$ dela, da pripravimo število d_i in ga primerjamo z b-jem, nato pa še $O(n)$ dela, da pripravimo c_i in ga primerjamo s $c^\star$. Ker gre i do $n - m$, je skupna časovna zahtevnost v najslabšem primeru $O(n(n + m))$.

To rešitev lahko še izboljšamo. Naš b je dolg $m + 1$ števk; torej, če povečujemo a po 1, se v 10^{m+1} povečanjih izmenjajo na njegovih spodnjih $m+1$ števkah že vse možne vrednosti teh $m+1$ števk, med drugim tudi tista, pri kateri imajo te števke vrednost b. Kandidat c_i, ki ga naša funkcija Naslednik izračuna pri $i = 0$, je torej $< a + 10^{m+1}$, zato tudi za končni rezultat $c^\star$, ki ga ta funkcija vrne, velja $c^\star < a + 10^{m+1}$.

Osredotočimo se zdaj pri večjih i na tiste primere, ko je $d_i \neq b$. (Primerov, ko je $d_i = b$, se lahko rešimo takole: naš postopek naj na začetku pogleda, če se b pojavlja kot podniz v številu a; če se, je rezultat kar a in lahko takoj nehamo. S primernim algoritmom, kot je na primer Knuth-Morris-Prattov, bi se dalo to narediti v $O(n + m)$ časa.) Takrat je c_i vsekakor večji od a; za koliko večji? Recimo, da je $b \neq (d_i + 1) \bmod 10^{m+1}$. Če bi a postopoma povečevali po 1, bi sčasoma prišlo do prenosa z mesta $i - 1$ na i in ob tem bi se vrednost na mestih $i, \ldots, i + m$ povečala za 1 (mod 10^{m+1}). Ker smo rekli, da $b \neq (d_i + 1) \bmod 10^{m+1}$, vrednost na mestih $i, \ldots, i + m$ zdaj še

ni enaka b in bi morali s povečevanjem a-ja po 1 nadaljevati tako dolgo, da bi se vrednost na mestih $i, \ldots, i+m$ spremenila vsaj še enkrat; toda takoj po neki spremembi i-te števke so na spodnjih i številkah ničle in do naslednje spremembe pri i-ti številki bo prišlo šele, ko se bo a povečal še za 10^i. Tako torej vidimo, da je v tem primeru $c_i > a + 10^i$. Ker smo v prejšnjem odstavku ugotovili, da je $c^\star < a + 10^{m+1}$, to pomeni, da pri $i > m+1$ vrednost c_i prav gotovo ne pride v poštev za rezultat $c^\star$. Primer: $a = 12345678$, $b = 987$; pri $i = 4$ imamo $d_i = 234$ in $c_i = 19870000$. Če bi a povečevali po 1, bi prišli do 12350000, malo kasneje do 12360000 in šele precej kasneje do c_i. Že tisto povečanje od 12350000 do 12360000 pa nam zagotavlja, da je c_i vsaj za 10^i (v našem primeru: $10^4 = 10000$) večji od a. Takšen c_i je neobetaven, saj se že po samo 1000 povečanjih a-ja za 1 izmenjajo na spodnjih treh mestih vse kombinacije treh števk, med drugim tudi naša iskana 987. (Tako je c_0 tukaj enak 123456987, kar je precej boljša rešitev od 19870000.)

Tako torej vidimo, da glavne zanke našega postopka ni treba speljati do $i = n - m + 1$, ampak le do $i = m + 1$. Višje vrednosti i so zanimive le, če se tam v a-ju pojavi vrednost $d_i = (b-1) \bmod 10^{m+1}$ (na primer, če je $b = 123$, mora biti $d_i = 122$; če pa je $b = 000$, mora biti $d_i = 999$). Če bi bil a recimo enak 1234999956 in $b = 35$, torej $m = 1$, bi pri $i = 6$ dobili $d_i = 34$ in $c_i = 12350000000$, kar je v tem primeru tudi najmanjši od vseh c_i-jev. Čeprav je i velik (večji od $m+1$), je razlika $c_i - a$ majhna (v tem primeru le 44). V splošnem vidimo, da bo v takem primeru $c_i - a$ enako $10^i - a_{i-1}a_{i-2}\ldots a_0$, ker se mora a povečati le toliko, da prvič pride do prenosa z mesta $i-1$ na i (takrat se d_i poveča za 1 in tako postane enak b). Oglejmo si zdaj niz $s_i := a_{i-1}a_{i-2}\ldots a_0$. Če so vse te števke enake 9, je $c_i - a = 10^i - s_i = 1$; torej se bo b pojavil kot podniz že v $a+1$: boljše rešitve torej sploh ne bi mogli dobiti, razen če se ne pojavlja b kot podniz že v samem a-ju (kar pa tako ali tako preverimo posebej že na začetku, kot smo rekli v prejšnjem odstavku). Drugače pa naj bo u_i indeks najbolj leve take števke v s_i, ki je različna od 9 (torej je $s_i = 9\overline{^{i-u_i-1}}a_{u_i}a_{u_i-1}\ldots a_0$). Pri povečevanju a-ja po 1 bo sčasoma prišlo do prenosa z mesta $u_i - 1$ na u_i, tako da se bo u_i-ta števka povečala; ker pa je bila prej manjša od 9, zdaj ne bo prišlo do prenosa naprej. Zato se na mestih od i naprej ne bo zgodilo še nič; tam se lahko kaj zgodi šele ob naslednji spremembi na mestu u_i. Med dvema spremembama na mestu u_i pa se a poveča za 10^{u_i}, torej je $c_i - a > 10^{u_i}$. Spomnimo se, da je $c^\star \le c_0 < a + 10^{m+1}$; če je torej $u_i > m+1$, že vemo, da je c_i prevelik in da to ne bo tista najboljša rešitev, ki jo iščemo. Torej se s tistimi indeksi i, za katere je $u_i > m+1$, sploh ni treba ukvarjati. Pri ostalih indeksih i pa vidimo, da je $c_i - a = 10^i - s_i = 10^i - 9\overline{^{i-u_i-1}}a_{u_i}a_{u_i-1}\ldots a_0 = 10^{u_i+1} - a_{u_i}a_{u_i-1}\ldots a_0$. (Na primer: $100000 - 99932$ je enako $100 - 32$.) Zato za izračun razlike $c_i - a$ porabimo tu le $O(u_i) = O(m)$ časa, ne glede na to, kako velik je i. Te razlike lahko primerjamo med seboj po vseh i in vemo, da je najmanjši c_i (to pa je

naša iskana rešitev) tisti, ki ima najmanjšo vrednost $c_i - a$.

Recimo, da smo pri nekem $i > m+1$ ugotovili, da je $d_i = (b-1) \bmod 10^{m+1}$, obenem pa je $u_i \le m+1$ in smo zato morali računati $c_i - a$ v skladu z razmislekom iz prejšnjega odstavka; in recimo, da se nam je enako zgodilo tudi pri nekem kasnejšem indeksu j ($j > i$). Iz definicije u_i sledi, da so števke $a_{u_i+1}, a_{u_i+2}, \ldots, a_{i-1}$ vse enake 9; to med drugim pomeni (ker je $u_i \le m+1 < i$), da so števke $a_{u_i+1}, \ldots, a_{m+1}$ same devetice. Podobno, ker je $u_j \le m+1 < j$, so števke $a_{u_j+1}, \ldots, a_{j-1}$ same devetice — med njimi so tudi vse števke $a_{m+2}, \ldots, a_{j-1}$. Torej so števke od a_{u_i+1} do a_{j-1} same devetice, a_{u_i} pa ne; zato je $u_j = u_i$. V prejšnjem odstavku smo videli, da je $c_i - a = 10^{u_i+1} - a_{u_i} a_{u_i-1} \ldots a_0$; torej, ker je $u_j = u_i$, je $c_j - a = c_i - a$. Torej je $c_j = c_i$, kar pomeni, da s pregledovanjem indeksov, večjih od i, ne bomo pridobili nobene boljše rešitve od tiste pri i. Zato se lahko ustavimo, čim obdelamo prvi i, večji od $m+1$.

Zapišimo zdaj celoten postopek v enem kosu. Za razliko od prejšnjega postopka, ki je računal vrednosti c_i in si zapomnil najmanjšo (recimo $c^\star$), bomo tukaj računali razlike $r_i := c_i - a$ in si zapomnili najmanjšo. Če je $r^\star$ najmanjša razlika, je $c^\star = r^\star + a$ rezultat, ki ga iščemo. Lepo pri tem je, da so vrednosti c_i dolge $O(n)$ števk, vrednosti r_i pa (za tiste indekse i, ki jih bo pregledal naš novi algoritem) le $O(m)$ števk, zato je delo z njimi hitrejše.

algoritem Naslednik2(a, b);
vhod: $a = a_n a_{n-1} \ldots a_0$, $b = b_m b_{m-1} \ldots b_0$;
vrne najmanjše tako naravno število, ki je $\ge a$ in vsebuje b kot podniz;

1 **if** $m > n$ **or** ($m = n$ **and** $a \le b$) **then** vrni b;
2 **if** se b pojavlja kot podniz v a **then** vrni a;
3 $r^\star := \infty$;
4 **for** $i := 0$ **to** $\min\{m+1, n-m+1\}$ **do begin**
5 $\quad d_i := a_{i+m} a_{i+m-1} \ldots a_i$;
6 $\quad$ **if** $d_i > b$ **then** $r_i := 1b0^{\underline{i}}$ **else** $r_i := b0^{\underline{i}}$;
7 $\quad r_i := r_i - d_i a_{i-1} a_{i-2} \ldots a_0$;
8 $\quad$ **if** $r_i < r^\star$ **then** $r^\star := r_i$;
9 **end**; {*for*}
10 **if** $b = 0^{\underline{m+1}}$ **then** $b' := 9^{\underline{m+1}}$ **else** $b' := b - 1$;
11 naj bo i najmanjši indeks, ki je $> m+1$ in velja $a_{i+m} a_{i+m-1} \ldots a_i = b'$; (če takega i ni, skoči na korak 17);
12 naj bo u_i najmanjši tak indeks, ki je $< i$ in so $a_{u_i+1}, \ldots, a_{i-1}$ same devetke; če so $a_0, \ldots, a_{i-1}$ same devetke, si mislimo $u_i = -1$;
13 **if** $u_i \le m+1$ **then begin**
14 $\quad r_i := 10^{u_i+1} - a_{u_i} a_{u_i-1} \ldots a_0$; (pri $u_i = -1$ dobimo $r_i = 1$)
15 $\quad$ **if** $r_i < r^\star$ **then** $r^\star := r_i$;
16 **end**; {*if*}
17 vrni $a + r^\star$;

Kakšna je časovna zahtevnost tega novega postopka? $O(n+m)$ za vrstico 1; prav toliko za vrstico 2, če za iskanje podniza b v nizu a uporabimo npr. Knuth-Morris-Prattov postopek; d_i v vrstici 5 je dolg $m+1$ znakov, i pa je $\le m+1$, zato je r_i v vrstici 6 dolg $O(m)$; vrstice 5–8 porabijo zato v vsaki iteraciji zanke po $O(m)$ časa, teh iteracij pa je največ $m+1$ (vrstica 4), tako da porabijo skupaj $O(m^2)$ časa. Vrstica 10 porabi $O(m)$ časa, vrstica 11 $O(n+m)$ (spet s Knuth-Morris-Prattovim algoritmom), vrstica 12 le $O(n)$ časa (zmanjšujmo u_i v zanki od $i-1$ navzdol, dokler še opažamo v a_{u_i} same devetke). Vrstici 14 in 15 porabita $O(m)$ časa, ker je $u_i \le m+1$. Seštevanje v vrstici 17 porabi $O(n+m)$ časa. Tako vidimo, da je časovna zahtevnost našega novega postopka vsega skupaj $O(n+m^2)$, kar vsekakor ni slabše od $O(n(n+m))$ iz prejšnjega postopka (saj je $m \le n$, oz. če ni, vemo, da je rezultat, ki ga iščemo, kar b, tako da lahko $m \le n$ obdelamo kot poseben primer v $O(1)$ časa), lahko pa je celo precej bolje (če je m manjši od n — torej če je b krajši od a).

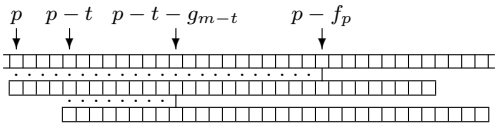
Tudi to rešitev lahko še precej izboljšamo. Za začetek izračunajmo nekaj pomožnih vrednosti, ki nam bodo prišle prav kasneje. Naj bo

$$\begin{aligned} g_p &= \max\{d : 0 \le d \le p+1,\ b_p b_{p-1} \dots b_{p-d+1} = b_m b_{m-1} \dots b_{m-d+1}\} \text{ in} \\ f_p &= \max\{d : 0 \le d \le p+1,\ d \le m+1, \\ &\qquad\qquad a_p a_{p-1} \dots a_{p-d+1} = b_m b_{m-1} \dots b_{m-d+1}\}. \end{aligned}$$

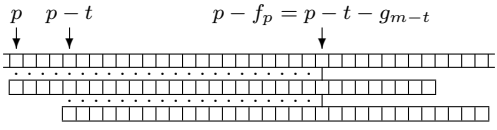
(Pri $d=0$ si mislimo, da sta $b_p b_{p-1} \dots b_{p-d+1}$ in $b_m b_{m-1} \dots b_{m-d+1}$ prazna niza.) Z besedami: če gledamo niz b od števke p naprej („naprej“ je tu mišljeno proti desni), se ujema s celotnim nizom b v levih g_p znakih (v več pa ne). Podobno, če gledamo niz a od števke p naprej, se ujema s celotnim nizom b v levih f_p znakih (v več pa ne). Iz te definicije lahko takoj vidimo, da bi lahko vse g_p izračunali z dvema gnezdenima zankama (po p in po d), podobno pa tudi vse f_p, vendar bi nam vse to vzelo $O(m^2+nm)$ časa; na srečo gre tudi hitreje. Do vrednosti g_p (za $p = 0, \dots, m$) lahko pridemo tako, da zgradimo drevo končnic (*suffix tree*) niza b; ko imamo enkrat to, ni težko določiti vseh g_p. Postopek gradnje takšnega drevesa je razmeroma zapleten in ga tu ne bomo podrobneje predstavljali (gl. op. na str. 660). Gradnja drevesa in računanje vseh g_p nam vzame $O(m)$ časa. Mimogrede, pri gradnji takega drevesa je koristno, če je na koncu niza nek znak, ki se drugod v nizu nikjer ne pojavlja; zato si mislimo, da obstaja tudi znak b_{-1}, ki je različen od vseh števk (0, 1, …, 9); zanj tudi definirajmo $g_{-1} = 0$. Kasneje bomo videli, da je koristno uvesti tudi znak a_{-1}, ki je različen od vseh števk in tudi od b_{-1}.

Zdaj ko poznamo vse g_p, lahko vrednosti f_p izračunamo v $O(n+m)$ časa tako, da malo prikrojimo Knuth-Morris-Prattov postopek za iskanje podniza b v nizu a. Niz a bomo pregledovali od leve proti desni; najprej izračunajmo f_n kar po definiciji, z zanko po d. Recimo zdaj, da že poznamo f_p za nek konkreten p. To pomeni, da je $a_p a_{p-1} \dots a_{p-f_p+1} = b_m b_{m-1} \dots b_{m-f_p+1}$ in $a_{p-f_p} \ne b_{m-f_p}$. Oglejmo si zdaj, kako lahko poceni pridemo do vrednosti f za

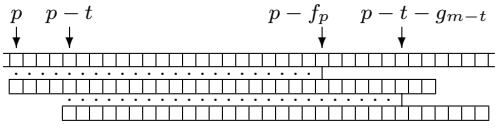
(1) Če je $g_{m-t} < f_p - t$:



(2) Če je $g_{m-t} = f_p - t$:



(3) Če je $g_{m-t} > f_p - t$:



Tri situacije, v katerih se lahko znajdemo pri računanju f_{p-t}, če že poznamo f_p. Pika med dvema znakoma pomeni, da sta enaka, črta pa, da sta različna.

Vidimo, da v teh situacijah velja naslednje:
(1) $f_{p-t} = g_{m-t}$;
(2) $f_{p-t} \geq g_{m-t}$;
(3) $f_{p-t} = f_p - t$.

---

nekaj naslednjih položajev, npr. do f_{p-t} za nek majhen pozitiven t. Vidimo, da se $a_{p-t} \dots a_0$ in $b_{m-t} \dots b_0$ ujemata vsaj v levih $f_p - t$ znakih, obenem pa vemo, da se $b_{m-t} \dots b_0$ in $b_m \dots b_0$ ujemata v levih g_{m-t} znakih, v naslednjem pa ne več. Primerjajmo zdaj g_{m-t} in $f_p - t$; ločimo tri možnosti (gl. gornjo sliko). (1) Če je $g_{m-t} < f_p - t$, lahko zaključimo, da se tudi $a_{p-t} \dots a_0$ in $b_m \dots b_0$ ujemata v levih g_{m-t} znakih, v naslednjem pa ne več; takrat je torej $f_{p-t} = g_{m-t}$. (2) Če je $g_{m-t} = f_p - t$, lahko zaključimo, da se $a_{p-t} \dots a_0$ in $b_m \dots b_0$ ujemata v levih $f_p - t = g_{m-t}$ znakih, za naslednjega pa ne vemo, če se v njem ujemata ali ne: vemo le, da je $a_{p-t-g_{m-t}} = a_{p-f_p} \neq b_{m-f_p} = b_{m-t-g_{m-t}} \neq b_{m-g_{m-t}}$, kar pa nam v splošnem ne pove nič zanesljivega o tem, ali sta si $a_{p-t-g_{m-t}}$ in $b_{m-g_{m-t}}$ enaka ali različna. Zato bomo morali nadaljevati s primerjanjem a-ja in b-ja od teh dveh položajev naprej. Lepo pri tem je, da se nam je, ko smo računali f_p, primerjanje a-ja z b-jem ustavilo pri a_{p-f_p}, kar je (zaradi $g_{m-t} = f_p - t$) ravno isti položaj kot $a_{p-t-g_{m-t}}$, na katerem bomo zdaj s primerjanjem spet začeli. Zato se z nobenim delom a-ja ne ukvarjamo po večkrat; časovna zahtevnost našega postopka bo le $O(n+m)$. (3) Če je $g_{m-t} > f_p - t$, lahko razmišljamo podobno kot pri (2); $a_{p-t} \dots a_0$ in $b_m \dots b_0$ se ujemata v levih $f_p - t$ znakih, na naslednjem mestu pa imamo $a_{p-t-(f_p-t)}$ (kar je isto kot a_{p-f_p}) in $b_{m-(f_p-t)}$; ker je $f_p - t < g_{m-t}$, je $b_{m-(f_p-t)} = b_{(m-t)-(f_p-t)}$, slednje je isto kot b_{m-f_p}, za tega pa po definiciji f_p vemo, da je različen od a_{p-f_p}. Torej lahko ravnamo enako kot v primeru (2), le da takoj opazimo neujemanje.

Postopek za računanje vrednosti f_p je torej lahko tak:

$p := n$; $f_p := 0$; **while** $a_{p-f_p} = b_{m-f_p}$ **do** $f_p := f_p + 1$;
while $p \geq 0$ **do begin**
  { *f_p že poznamo. Pregledujmo zdaj naslednje položaje, $p - t$ za $t = 1, 2, \ldots$, dokler pri njih velja primer (1) in jim lahko takoj določimo f_{p-t}.* }
  $t := 1$;
  while $g_{m-t} < f_p - t$ **do**
    begin $f_{p-t} := g_{m-t}$; $t := t + 1$ **end**;
  { *Premaknimo se na položaj $p - t$, kjer velja primer (2) ali (3); v obeh primerih vemo, da je $f_{p-t} \geq f_p - t$. Poseben primer je $f_p = 0$, ko se zgolj premaknemo na položaj $p - 1$ in postavimo f_{p-1} na 0.* }
  if $f_p = 0$ **then** $f_{p-t} := 0$ **else** $f_{p-t} := f_p - t$;
  $p := p - t$;
  { *Določimo pravo vrednost f_p na novem položaju p (bivšem $p - t$).* }
  while $a_{p-f_p} = b_{m-f_p}$ **do** $f_p := f_p + 1$;
end; {*while*}

Kot smo omenili že zgoraj, si na koncu nizov mislimo znaka a_{-1} in b_{-1}, ki sta različna od vseh ostalih znakov in še drug od drugega; to nam zagotavlja, da se zanke v gornjem postopku pravočasno in brez posebnih kompliciranj ustavijo. Gornji postopek nam izračuna tudi $f_{-1} = 0$; za g_{-1} pa smo že prej rekli, da je tudi enak 0.

Po teh predpripravah se posvetimo zanki v vrsticah 4–9 našega starega postopka Naslednik2. Ravno ta zanka namreč prinese časovno zahtevnost $O(m^2)$, saj ima $O(m)$ iteracij, v vsaki iteraciji pa nekaj operacij z zahtevnostjo $O(m)$. Oglejmo si zdaj, kako lahko s pomočjo vrednosti g_p in f_p izvedemo vsako iteracijo te zanke v času $O(1)$.

Najprej moramo primerjati d_i z b, da vidimo, ali velja $d_i > b$ ali $d_i < b$. Spomnimo se, da je d_i le podniz a-ja: $d_i = a_{i+m}a_{i+m-1}\ldots a_i$. Zato se d_i in b ujemata v levih f_{i+m} številkah, na naslednjem mestu pa se razlikujeta; tako moramo le primerjati ti dve številki, torej $a_{i+m-f_{i+m}}$ in $b_{m-f_{i+m}}$: če je večja prva, je $d_i > b$, če je večja druga, pa je $d_i < b$.

Naša zanka bo svoje delo opravila, če v vsaki iteraciji določi c_i in ugotovi, kateri od vseh c_i-jev je najmanjši. Potrebujemo torej postopek, ki bo znal v konstantnem času primerjati c_i in c_j. Seveda si tudi ne moremo privoščiti, da bi c_i in c_j eksplicitno zapisali v kakšno tabelo, saj nam potem vsaka iteracija že ne bi vzela več le konstantno mnogo časa. Pokazali bomo, da imata števili c_i in c_j tako preprosto in predvidljivo zgradbo, da ju lahko primerjamo že v konstantno mnogo časa (in to ne da bi ju zapisali eksplicitno); pri tem pa si bomo pomagali s prej izračunanimi vrednostmi g_p in f_p.

Recimo, da pri nekem konkretnem i velja $d_i > b$. Torej je število c_i oblike $(a_n a_{n-1}\ldots a_{i+m+1} + 1)b0^i$. V nizu $a_n a_{n-1}\ldots a_{i+m+1}$ je zadnjih nekaj števk mogoče enakih 9, prej ali slej pa je ena različna od 9 (če drugega ne, si mislimo, da obstaja še $a_{n+1} = 0$). Recimo, da je a_{t_i}, $t_i \leq i + m + 1$,

najbolj desna ne-devetka v tem delu a-ja. Torej je $a_n a_{n-1} \ldots a_{i+m+1} + 1 = a_n a_{n-1} \ldots a_{t_i+1} a_{t_i} 9^{\underline{t_i-1-m-1}} + 1 = a_n a_{n-1} \ldots a_{t_i+1}(a_{t_i}+1)0^{\underline{t_i-i-m-1}}$. Zato je c_i sestavljen iz petih kosov (recimo jim „A_i“, „T_i“ in tako naprej):

$$c_i \quad = \quad \overbrace{a_n a_{n-1} \ldots a_{t_i+1}}^{A_i} \quad \overbrace{(a_{t_i}+1)}^{T_i} \quad \overbrace{0^{\underline{t_i-1-m-1}}}^{N_i} \quad \overbrace{b}^{B_i} \quad \overbrace{0^{\underline{i}}}^{S_i} .$$

(Vrednosti t_i za vse i od 0 do n lahko izračunamo na začetku našega postopka v času $O(n)$.) Pri tistih i, za katere je $d_i < b$, je stvar še preprostejša; tam je c_i sestavljen le iz treh delov (A_i, B_i in S_i):

$$c_i \quad = \quad \overbrace{a_n a_{n-1} \ldots a_{i+m+1}}^{A_i} \quad \overbrace{b}^{B_i} \quad \overbrace{0^{\underline{i}}}^{S_i} .$$

Spomnimo se, kako bi prej omenjeni podprogram Primerjaj primerjal števili c_i in c_j: šel bi s števcem p od njune dolžine (to je načeloma n ali pa največ $n+1$) navzdol proti 0 in na vsakem koraku primerjal p-ti števki obeh števil, dokler ne bi na nekem mestu opazil, da se trenutni števki razlikujeta. Ker se p v vsaki ponovitvi zanke zmanjša za 1, bi takšno primerjanje trajalo v najslabšem primeru $O(n)$ časa.

Zgoraj smo videli, da sta tako c_i kot c_j sestavljena iz petih (ali celo samo treh) kosov. Trenutna vrednost p pade pri vsakem od njiju v enega od teh petih kosov. Izkaže se, da lahko v vsakem primeru, ne glede na to, katera dva kosa sta to, v konstantno mnogo časa bodisi odkrijemo položaj prvega neujemanja (in tako ugotovimo, ali je manjši c_i ali c_j) bodisi zmanjšamo p za toliko, da vsaj pri enem od c_i in c_j zdaj pade na začetek naslednjega kosa. Ker je kosov v vsakem od primerjanih števil največ 5, je jasno, da bomo morali izvesti le omejeno mnogo iteracij naše zanke, ne glede na dolžino a-ja in b-ja.

Oglejmo si konkreten primer: recimo, da je trenutni položaj p tak, da pri številu c_i pade v del A_i, pri številu c_j pa že v del N_j. Če imamo v neki tabeli za vsak položaj v a-ju pripravljen podatek o tem, koliko strnjenih ničel je desno od tega položaja (torej $h_p = \max\{d : a_p a_{p-1} \ldots a_{p-d+1} = 0^{\underline{b}}\}$ — vrednosti h_p za vse p lahko izračunamo na začetku našega postopka v času $O(n)$), lahko razmišljamo takole: pri c_j so od položaja p pa vse do konca dela N_j, torej vse do vključno položaja $j+m+1$, same ničle; to je skupaj $p-j-m$ ničel. Po drugi strani lahko v številu a od mesta p naprej vidimo h_p ničel; v številu c_j torej $\min\{h_p, p-t_i\}$ ničel, kajti po največ $p-t_i$ števkah bo dela A_i že konec in bomo prišli v T_i (tista števka tam pa je gotovo neničelna). Torej imata na naslednjih $\min\{p-j-m, h_p, p-t_i\}$ mestih tako c_i kot c_j ničlo; p lahko takoj zmanjšamo za toliko. S tem skokom smo mogoče prišli v c_i do konca dela A_i, mogoče v c_j do konca dela N_j, mogoče celo oboje, mogoče pa sicer nič od tega, kar pa pomeni, da smo odkrili mesto, kjer ima c_i že neničelno števko, v c_j pa še traja zaporedje ničel, torej imamo neujemanje in lahko s primerjavo zaključimo.

V prejšnjem odstavku smo razmišljali o primeru, ko smo v c_i znotraj A_i in v c_j znotraj N_j. Možnih je seveda še veliko drugih kombinacij, načeloma 25 (pet možnosti glede tega, v katerem delu c_i-ja smo, in pet glede tega, v katerem delu c_j-ja smo; izkaže se sicer, da so nekatere kombinacije nemogoče); s podobnim razmislekom se lahko za vsako od njih prepričamo, da lahko res v $O(1)$ časa odkrijemo neujemanje ali pa zmanjšamo p za toliko, da se v vsaj enem od števil c_i in c_j znajdemo v naslednjem kosu. Včasih si moramo pomagati z vrednostmi f_p in g_p, podobno kot smo si v prejšnjem odstavku pomagali z vrednostmi h_p.

Tako torej vidimo, da smo porabili na začetku $O(n+m)$ časa za računanje vrednosti f_p, g_p, h_p in t_i, nato pa pri vsaki izmed $O(m)$ iteracij naše glavne zanke še $O(1)$ časa za primerjavo d_i in b ter za primerjavo trenutnega c_i z največjim doslej znanim. Na koncu zanke vemo, pri katerem indeksu i; smo dobili najmanjši c_i; zdaj lahko izračunamo r_i in jo zapišemo v spremenljivko $r^\star$, nato pa izvedemo vrstice 10–17 postopka Naslednik2. Te vrstice porabijo, kot smo videli že zgoraj, le $O(n+m)$ časa, tako da zdaj tudi celoten postopek iskanja naslednika porabi le $O(n+m)$ časa. Takšna zahtevnost je v nekem smislu tudi najboljša možna, saj bi porabil $O(n+m)$ časa že tudi vsak postopek, ki bi niza a in b vsaj enkrat v celoti prebral.

R2004.X.5 Zamenjave

Odprta gesla (taka, pri katerih smo že naleteli na `$(`, na zaklepaj pa še ne) odlagajmo na sklad. Ko naletimo na zaklepaj, se zadnje odprto geslo (tisto z vrha sklada) zapre in ga zamenjamo z novo vrednostjo iz slovarja (če ga seveda najdemo v slovarju). Če je ob koncu vrstice še kaj gesel odprtih, pomeni, da manjka nekaj zaklepajev. N: 630

```
program Zamenjave;
const MaxSlovar = 1000;
      MaxGnezdenje = 50;
var Slovar: array [1..MaxSlovar, 1..2] of string;
    StGesel: integer;

  function Zamenjaj(Geslo: string): string;
  var i: integer;
  begin
    for i := 1 to StGesel do
      if Slovar[i, 1] = Geslo then
        begin Zamenjaj := Slovar[i, 2]; exit end;
    Zamenjaj := '$(' + Geslo + ')';
  end; {Zamenjaj}

var Sklad: array [0..MaxGnezdenje] of string; SP: integer;
    S: string; i, L: integer;
```

```
  T, TT: text;
begin
  { Preberimo gesla in njihove zamenjave. }
  Assign(T, 'zamenjave.in'); Reset(T); StGesel := 0;
  while true do begin
    ReadLn(T, S); L := Length(S);
    if L = 0 then break;
    i := 1; while i < L do if S[i] = ' ' then break else i := i + 1;
    StGesel := StGesel + 1;
    Slovar[StGesel, 1] := Copy(S, 1, i - 1);
    Slovar[StGesel, 2] := Copy(S, i + 1, L - i);
  end; {while}

  { Berimo besedilo, izpisujmo predelano besedilo. }
  Assign(TT, 'zamenjave.out'); Rewrite(TT);
  while not Eof(T) do begin
    ReadLn(T, S); SP := 0; Sklad[SP] := '';

    { Obdelajmo trenutno vrstico. }
    i := 1; L := Length(S);
    while i <= L do begin
      if S[i] = '$' then begin
        if i = L then { Dolar na koncu vrstice — pustimo ga kar pri miru. }
          Sklad[SP] := Sklad[SP] + S[i]
        else if S[i + 1] = '(' then begin { Začetek gesla. }
          SP := SP + 1; Sklad[SP] := ''; i := i + 1;
        end else if (S[i + 1] = '$') or (S[i + 1] = ')') then begin
          { Naleteli smo na par '$$' ali '$)'. }
          i := i + 1; Sklad[SP] := Sklad[SP] + S[i];
        end else { Napaka — pustimo dolar pri miru. }
          Sklad[SP] := Sklad[SP] + S[i];
      end else if S[i] = ')' then begin
        if SP = 0 then { Zaklepaj brez pripadajočega '$('. }
          Sklad[SP] := Sklad[SP] + S[i]
        else begin { Zamenjajmo geslo na vrhu sklada. }
          Sklad[SP - 1] := Sklad[SP - 1] + Zamenjaj(Sklad[SP]);
          SP := SP - 1;
        end; {if}
      end else { Navaden znak. }
        Sklad[SP] := Sklad[SP] + S[i];
      i := i + 1;
    end; {while}

    { Izpišimo prežvečeno vrstico. }
    Write(TT, Sklad[0]);
    { Še morebitna nedokončana gesla (če se pojavi '$(' brez pripadajočega ')'). }
    for i := 1 to SP do Write(TT, '$(', Sklad[i]);
    WriteLn(TT);
```

```
  end; {while}
  Close(T); Close(TT);
end. {Zamenjave}
```

Podprogram Zamenjaj išče zamenjavo za dano geslo tako, da po vrsti pregleduje zapise v slovarju, dokler ne naleti na pravega. Če je slovar velik, zna biti to časovno precej potratno; v tem primeru bi bilo dobro slovar hraniti v razpršeni tabeli.

R2004.X.6 Vžigalice

N: 631

Stanje igre lahko opišemo s v-terico števil $x = (x_1, \ldots, x_v)$, ki povedo, koliko vžigalic je še ostalo v posamezni vrstici. Na začetku igre je $x = a = (a_1, \ldots, a_v)$. Naj bo $N(x)$ množica vseh stanj, v katera lahko pridemo iz stanja x v eni potezi. Recimo, da je x trenutno stanje igre; naj bo $Z(x)$ logična vrednost (0 ali 1), ki nam pove, če obstaja za igralca, ki je trenutno na potezi, zanesljiva zmagovalna strategija (temu bomo rekli tudi, da je stanje zmagovalno), $P(x)$ pa logična vrednost, ki nam pove, če obstaja zanesljiva zmagovalna strategija za drugega igralca (tistega, ki trenutno ni na potezi; temu bomo rekli tudi, da je stanje pogubno, ker zagotavlja poraz tistemu, ki je na potezi, ko je igra v tem stanju).

Očitno je, da $Z(x)$ in $P(x)$ ne moreta biti oba hkrati resnična. Naloga pravi, da je vedno resničen natanko eden od njiju; prepričajmo se, da je res tako. Za končno stanje, $x = (0, 0, \ldots, 0)$, imamo $Z(x) = 1$ in $P(x) = 0$, kajti če smo mi na potezi in na mizi ni nobene vžigalice, pomeni, da je drugi igralec tik pred tem pobral zadnjo in tako izgubil igro. Drugače pa lahko razmišljamo takole: tisti, ki je na potezi, ima zagotovljeno zmago, če lahko v svoji naslednji potezi popelje igro v stanje, ki zagotavlja poraz drugega igralca; tisti, ki ni na potezi, pa ima zagotovljeno zmago šele, če mu je le-ta zagotovljena pri vseh možnih naslednikih trenutnega stanja (saj šele v tem primeru njegov nasprotnik, ki je trenutno na potezi, ne bo mogel storiti ničesar, da bi se izognil porazu). S simboli lahko to zapišemo takole: $Z(x) = \bigvee_{y \in N(x)} P(y)$ in $P(x) = \bigwedge_{y \in N(x)} Z(y)$.

Recimo zdaj, da za nekatera stanja x ne bi veljalo niti $Z(x)$ niti $P(x)$. Naj bo x med temi stanji tako z najmanjšim skupnim številom vžigalic (če je takšnih več, je vseeno, katero vzamemo). To pomeni, da za vse $y \in N(x)$ (ki imajo seveda manj vžigalic kot x) velja $Z(x) \Leftrightarrow \neg P(x)$. Ker x gotovo ni končno stanje (tisto brez vžigalic; zanj smo že videli, da velja $Z(x) \wedge \neg P(x)$), ima vsaj enega naslednika. Iz $\neg P(x)$ in dejstva, da ima x vsaj enega naslednika, sledi, da obstaja nek $y \in N(x)$, za katerega je $\neg Z(y)$. Ker ima y manj vžigalic kot x, sledi iz $\neg Z(y)$ tudi $P(y)$. To pa po definiciji Z pomeni tudi $Z(x)$, kar je v protislovju z našo predpostavko, da za x ne velja niti $Z(x)$ niti $P(x)$. Torej je zagotovilo iz besedila naloge res resnično: v vsakem stanju x je enemu od

igralcev zagotovljena zmagovalna strategija. Če velja $Z(x)$, je to tisti, ki je trenutno na potezi; če velja $P(x)$, pa tisti drugi.

Rešitev s pregledovanjem prostora stanj. Zdaj že tudi vidimo, kako bi lahko rešili nalogo: formuli za $Z(x)$ in $P(x)$ iz predprejšnjega odstavka bi lahko zapisali kot dva podprograma, ki bi v zanki pregledovala naslednike stanja x, torej vse $y \in N(x)$, in klicala drug drugega, da bi prišla do vrednosti $P(y)$ oz. $Z(y)$. Šlo bi pravzaprav tudi z enim samim podprogramom: stanje je zmagovalno natanko tedaj, ko lahko iz njega v eni potezi ustvarimo neko pogubno stanje.

```
function Z(x): boolean;
begin
  if x = (0,...,0) then return true;
  for each y ∈ N(x) do
    if not Z(y) then return true;
  return false;
end;
```

Naštevanje naslednikov trenutnega stanja bi lahko izvedli z dvema zankama — v zunanji si izberemo vrstico, iz katere bo trenutna poteza vzela vžigalice, v notranji pa si izberemo število vžigalic, ki jih bomo vzeli iz trenutne vrstice.

Slabost te rešitve je njena časovna potratnost, saj lahko pregleda veliko stanj med začetnim stanjem $(a_1, \ldots, a_v)$ in končnim stanjem $(0, \ldots, 0)$ (čeprav ne nujno vseh). Nekatera lahko pregleda celo po večkrat; temu bi se sicer lahko izognili, če bi vrednosti funkcije Z za že izračunana stanja hranili v kakšni tabeli (ali pa v razpršeni tabeli ali pa v drevesu), vendar bi s tem močno narasla poraba pomnilnika. Spomnimo se, da je iz stanja $a = (a_1, \ldots, a_v)$ načeloma mogoče doseči vsako stanje $(x_1, \ldots, x_v)$, ki ima $0 \le x_i \le a_i$ za vse $i = 1, \ldots, v$; to je z a-jem vred skupaj kar $(a_1 + 1)(a_2 + 1) \cdots (a_v + 1)$ stanj. Nekaj časa in pomnilnika bi lahko prihranili tako, da bi upoštevali, da lahko vrstice premešamo ter dodajamo ali brišemo prazne vrstice, ne da bi to na igro zares kaj vplivalo. Stanji $(3, 4, 2, 3, 2)$ in $(4, 3, 3, 2, 2)$ sta povsem enakovredni; tudi ko naštevamo naslednike tega stanja, je dovolj, če pri odvzemanju vžigalic gledamo le eno od vrstic s po tremi vžigalicami, druge pa ne, ipd.

Učinkovitejša rešitev. Nalogo lahko rešimo tudi precej hitreje, čeprav do te rešitve ni ravno preprosto priti. Označimo $x = (x_1, \ldots, x_v)$, $y = (y_1, \ldots, y_v)$, $2x = (2x_1, \ldots, 2x_v)$. Izkaže se, da je $Z(2x) = Z(x)$, razen če je $x_1 = \ldots = x_v = 1$, tedaj pa je $Z(2x) = \neg Z(x)$. Če je y sestavljen iz samih ničel in enic in je enic sodo mnogo, je $Z(2x + y) = Z(2x)$. Če pa je enic liho mnogo, je $Z(2x + y) = 1$. S temi ugotovitvami pridemo do naslednjega postopka (dokaz, da je ta rešitev pravilna, bomo videli na str. 675):

```
function Z(x): boolean;
```

begin
1 če ni v x nobena komponenta večja od 1:
2 če ima x liho mnogo enic, vrni false, sicer vrni true;
3 ponavljaj:
4 če je vsota komponent x liha, vrni true;
5 vsako komponento x deli z 2 in zaokroži navzdol;
6 če ni v x nobena komponenta večja od 1:
7 če ima x liho mnogo enic, vrni true, sicer vrni false;
end;

V vsaki iteraciji glavne zanke se število vžigalic v vsaki neprazni vrstici zmanjša vsaj za polovico, zato se izvede največ $O(\lg m)$ iteracij, če je imelo začetno stanje v vsaki vrstici največ m vžigalic.

Z nekaj pazljivosti lahko to rešitev implementiramo še bolj učinkovito. Oglejmo si, kakšne so videti operacije, ki jih izvaja naš gornji algoritem, če števila $x_1, \ldots, x_v$ predstavimo v dvojiškem zapisu.

Vsota komponent vektorja x (ki jo računamo v vrstici 4) je liha natanko tedaj, če je izmed števil $x_1, \ldots, x_v$ liho mnogo lihih; posamezno od teh števil pa je liho natanko tedaj, če je prižgan njegov najnižji bit. Za preverjanje, ali je takih števil liho ali sodo mnogo, si lahko pomagamo z operacijo xor: če xoramo vsa števila $x_1, \ldots, x_v$, je najnižji bit rezultata prižgan natanko tedaj, če je bil ta bit prižgan v liho mnogo izmed števili $x_1, \ldots, x_v$.

Vrstica 5 mora deliti vsak x_i z 2 in rezultat zaokrožiti navzdol; ta operacija preprosto pobriše najnižji bit števila x_i. Če nas bo v bodoče nekoč zanimal najnižji bit števila x_i (na primer ob izvajanju vrstice 4 v naslednji iteraciji glavne zanke), je to prav ista vrednost, ki je bila pred deljenjem z 2 na bitu 1. Torej ni treba res deliti vseh x_i z 2, pač pa je dovolj že, če si zapomnimo, na katerem bitu so zdaj tiste vrednosti, ki bi bile v najnižjem bitu, če bi deljenja res izvajali. To število se ob vsakem deljenju (torej: v vsaki iteraciji glavne zanke) poveča za 1.

V vrsticah 1 in 6 moramo preveriti, če je ostala še kakšna komponenta, večja od 1. Na začetku algoritma bi lahko pogledali, kateri je najvišji bit, ki je še prižgan v vsaj enem izmed x_i; recimo, da je to bit B. Ker ob vsakem deljenju z 2 (v vrstici 5) izgubi vsak x_i svoj najnižji bit, bo treba B deljenj, preden bo tudi največja izmed vrednosti x_i padla na 1. Tako ni težko preveriti, ali je še kateri od x_i večji od 1 ali ne: treba je le pogledati, če smo že izvedli B deljenj.

V vrsticah 2 in 7 moramo preveriti — po tistem, ko vemo, da so vse komponente vektorja x zdaj enake 0 ali 1 — ali je komponent z vrednostjo 1 liho mnogo. To je pravzaprav enako vprašanje kot v vrstici 4 in odgovor lahko spet dobimo s pomočjo operatorja xor.

Naj bo y vrednost, ki jo dobimo, če xoramo vsa števila $x_1, \ldots, x_v$. V zadnjih nekaj odstavkih smo se prepričali, da glavna zanka našega algoritma

pravzaprav ne počne drugega, kot da gleda, če je na bitih $0, \ldots, B-1$ v številu y kakšna enica (vrstica 4) — če jo najde, vrne true — drugače pa pogleda bit B števila y (vrstici 2 in 6) in vrne true ali pa false v odvisnosti od tega, ali je ta bit prižgan, in od tega, ali je B večji od 0 ali ne. Stanje x je zmagovalno, če je $B = 0$ in najvišji bit y-a ugasnjen ali pa je $B > 1$ in najvišji bit y-a prižgan. Zdaj lahko zapišemo spodnjo elegantno in učinkovito rešitev:

```
const v = ...;
type StanjeT = array [1..v] of integer;

function Zmagovalno(var x: StanjeT): boolean;
var i, y, m: integer;
begin
  m := x[1]; y := x[1];
  for i := 2 to v do begin
    if x[i] > m then m := x[i];
    y := y xor x[i];
  end; {for i}
  Zmagovalno := (m <= 1) = (y = 0);
end; {Zmagovalno}
```

V gornjem podprogramu je m največja vrednost v vektorju x. Pri $m \leq 1$ je torej $B = 0$ in y je dolg en sam bit; enak je 0, če je v stanju x sodo mnogo enic (in le takrat je x zmagovalno stanje). Pri $m > 1$ pa je $B > 0$ in stanje x je zmagovalno, če je prižgan vsaj eden od bitov v y (torej: če y ni enak 0).

Zmagovalna strategija. Če je neko stanje x zmagovalno (torej zanj gornja funkcija Zmagovalno vrne true) in smo pri tem stanju na potezi mi, vemo, da lahko igramo tako, da bomo nasprotnika prisilili v poraz. Kakšno potezo moramo najprej povleči? Pri $B = 0$ (oz. $m \leq 1$) sploh nimamo nobene izbire — v stanju x so le ničle in enice in ne moremo storiti drugega, kot da iz poljubne neprazne vrstice poberemo njeno edino vžigalico. Pri $B > 0$ (oz. $m > 1$) pa, ker je Zmagovalno vrnila true, vemo, da je $y \neq 0$. Iz trenutnega stanja moramo s svojo potezo narediti neko novo stanje, ki ne bo zmagovalno. Ker je $m > 1$, je v x neka komponenta $x_i > 1$; če je to edina komponenta, večja od 1, jo lahko v svoji potezi zmanjšamo bodisi na 1 bodisi na 0; ena od teh dveh možnosti nam bo zanesljivo dala $y = m = 1$ (katera, je odvisno od ostalih komponent stanja x: če je xor teh ostalih komponent enak 1, moramo x_i postaviti na 0, sicer pa na 1), kar pomeni, da novo stanje ne bo zmagovalno. Druga možnost pa je, da je v x več komponent, večjih od 1. Oglejmo si najvišji prižgani bit v številu y (recimo, da je to bit b); ker je y nastal z xoranjem komponent stanja x, pomeni, da ima tudi vsaj ena od komponent tega stanja tisti bit prižgan; recimo, da je to komponenta x_i. Če zdaj x_i spremenimo v $x_i \operatorname{xor} y$, se bo y postavil na 0, poleg tega pa se bo x_i zmanjšal,[119] tako da bo to veljavna poteza

[119] Zakaj je $x_i \operatorname{xor} y < x_i$? Najvišji prižgani bit v y je bit b, ki je prižgan tudi v x_i; ob

v skladu s pravili igre. Dobimo torej stanje, ki ima $y = 0$, vrednost m pa je še vedno > 1 (ker smo rekli, da x_i ni edina komponenta, večja od 1), zato novo stanje ni zmagovalno.

Dokaz pravilnosti. Strategijo iz prejšnjega odstavka lahko uporabimo tudi za eleganten dokaz, da je podprogram Zmagovalno pravilna rešitev naše naloge (spomnimo se, da tega doslej nismo dokazali — podprogram Zmagovalno temelji na trditvah, ki smo jih brez dokaza navedli na str. 672). Dokazovali bomo z indukcijo po številu vžigalic. Pri stanju z 0 vžigalicami dobimo $m = 0$ in $y = 0$ in Zmagovalno vrne true, kar je tudi prav. Pri eni vžigalici dobimo $m = 1$, $y = 1$ in funkcija pravilno vrne false. Recimo zdaj, da smo dokazali pravilnost že za vsa stanja z manj kot n vžigalicami; naj bo x neko stanje z n vžigalicami. (1) Če ima trenutno stanje $m = 1$, je vsaka vžigalica v svoji vrstici in očitno je, da je tako stanje zmagovalno natanko tedaj, ko je vrstic sodo mnogo; to pa je ravno enako pogoju $y = 0$, torej funkcija v tem primeru vrne pravilno vrednost. (2) Če pa je $m > 1$, ločimo dva primera. (2a) Pri $y \neq 0$ nam strategija iz prejšnjega odstavka kaže, kako lahko iz stanja x v eni potezi naredimo neko tako stanje, pri katerem funkcija Zmagovalno vrne false, ker pa ima to novo stanje manj vžigalic kot trenutno, sledi po induktivni predpostavki, da to stanje tudi v resnici ni zmagovalno; ker torej obstaja način, kako iz trenutnega stanja v eni potezi dobiti nezmagovalno stanje, mora biti trenutno stanje zmagovalno; in ker je $m > 1$ in $y \neq 0$, bi naša funkcija vrnila true, kar je torej popolnoma pravilno. (2b) Pri $y = 0$ pa naša funkcija vrne false; prepričajmo se, da stanje x v tem primeru res ni zmagovalno. Če bi bilo, bi se dalo iz njega v eni potezi narediti neko nezmagovalno stanje x'; ker bi imelo x' manj vžigalic od x, bi po induktivni predpostavki naša funkcija Zmagovalno za x' vrnila false. Torej bi za x' veljalo $m' = y' = 1$ ali pa $m' > 1$ in $y' = 0$. Poteza iz x v x' bi morala zmanjšati neko komponento x_i v x'_i, ostale komponente pa pustiti pri miru; torej je $y' = y \operatorname{xor} x_i \operatorname{xor} x'_i$, kar pomeni, da možnost $y' = 0$ odpade, saj je že y enak 0 in bi to pomenilo, da je $x_i = x'_i$. Torej mora biti $m' = y' = 1$; ker je bil $m > 1$, pomeni, da je bila v x večja od 1 le komponenta x_i in da smo jo zdaj zmanjšali na $x'_i \leq 1$; toda če je bila v x samo komponenta x_i večja od 1, bi moral biti najvišji prižgani bit števila x_i prižgan tudi v številu y, torej y ne bi bil enak 0. Prišli smo v protislovje, torej stanje x res ne more biti zmagovalno.

Ta dokaz nam sicer dokazuje, da je naša rešitev pravilna, ne pove pa nam kaj dosti o tem, kako bi človek do takšne rešitve sploh prišel. Dalo bi se sestaviti tudi bolj ilustrativen dokaz, ki bi lepše odražal to, kako lahko pridemo do te rešitve, če je še ne poznamo; ker pa bi bil ta dokaz malo bolj dolgovezen, ga

---

prehodu iz x_i v $x_i \operatorname{xor} y$ se ta bit ugasne, nekateri od nižjih bitov pa se lahko mogoče prižgejo; toda ugašanje bita b zmanjša vrednost števila za 2^b, prižiganje vseh nižjih bitov pa lahko vrednost poveča za največ $1 + 2 + 4 + \ldots + 2^{b-1} = 2^b - 1$, torej bo končna vrednost vendarle manjša od začetne.

tu ne bomo navajali.

Štetje stanj. Za konec si oglejmo še, koliko je vseh razporedov n vžigalic in koliko med njimi je neugodnih za prvega igralca (torej tistega, ki je prvi na potezi). Za začetek opazimo, da če lahko en razpored dobimo iz drugega le s spreminjanjem vrstnega reda vrstic ali pa z brisanjem ali dodajanjem praznih vrstic, sta z vidika naše igre oba razporeda pravzaprav čisto enakovredna: prazne vrstice na igro ne vplivajo, saj ni mogoče v njih izvesti nobene poteze, spreminjanje vrstnega reda vrstic pa na igro tudi ne vpliva, saj lahko še vedno izvajamo enake poteze kot prej (le da je vrstica, iz katere pobiramo, pač drugje kot prej). Zato se dogovorimo, da takšnih stanj ne bomo razlikovali med sabo; ali, z drugimi besedami, od vsake skupine stanj, ki se jih da dobiti drugo iz drugega le z premetavanjem vrstic in dodajanjem ali brisanjem praznih vrstic, bomo gledali le eno stanje — recimo kar tisto, v katerem ni praznih vrstic in v katerem so dolžine vrstic urejene nenaraščajoče. Primer takega stanja je na primer $(5, 3, 3, 2, 2, 2)$, medtem ko npr. stanja $(2, 5, 0, 3, 2, 2, 3)$, ki mu je sicer enakovredno, ne bomo obravnavali.

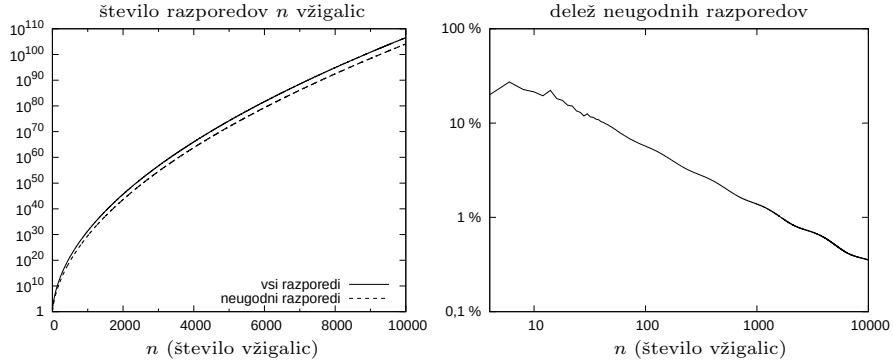
Koliko je vseh stanj z n vžigalicami? Kot pogosto pri takšnih problemih je koristno tudi zdaj ta problem malo posplošiti: vprašajmo se, koliko je vseh takih stanj z n vžigalicami, ki imajo v prvi vrstici kvečjemu k vžigalic; recimo temu $f(n, k)$. Ločimo dve možnosti: če je v prvi vrstici dejansko k vžigalic (kar je sicer možno le pri $n \geq k$), lahko ostale vrstice zgradimo na $f(n-k, k)$ načinov; če pa v prvi vrstici ni k vžigalic, pač pa manj, lahko celotno stanje zgradimo na $f(n, k-1)$ načinov. Tako smo prišli do rekurzivne zveze $f(n, k) = f(n-k, k) + f(n, k-1)$. Robni primeri so $f(0, 0) = 1$, $f(0, k) = 0$ za $k > 0$, $f(n, 0) = 0$ za $n > 0$, $f(n, k) = 0$ za $n < 0$. Funkcijo f lahko računamo z rekurzijo, lahko pa s pomočjo tabele (po naraščajočih k in pri vsakem k po vseh n, kar nam bo vzelo $O(n^2)$ časa in $O(n)$ pomnilnika). Število vseh razporedov z n vžigalicami je $f(n, n)$.[120]

Koliko pa je med stanji z n vžigalicami takih, ki so neugodna za prvega igralca (torej takih, pri katerih ima zmagovalno strategijo tisti, ki ni prvi na potezi)? Spomnimo se, kaj smo ugotovili za takšna stanja: imeti morajo $m > 1$ in $y = 0$ ali pa $m = 1$ in $y = 1$. Pri tem je y vrednost, ki jo dobimo pri xoranju vseh dolžin vrstic, m pa je dolžina najdaljše vrstice (zdaj je to kar dolžina prve vrstice, saj smo rekli, da gledamo le stanja, ki imajo vrstice urejene po nenaraščajoči dolžini). Naj bo $g(n, k, y)$ število vseh stanj, ki imajo n vžigalic, v prvi vrstici kvečjemu k vžigalic in pri katerih je xor vseh dolžin vrstic enak y. Enako kot pri f lahko tudi zdaj ločimo dve možnosti: če je v prvi vrstici k

[120] Štetje takšnih razporedov je pravzaprav isti problem kot štetje razbitij (particij) števila n (torej: na koliko načinov lahko izrazimo n kot vsoto nič ali več pozitivnih celih števil), ki smo ga srečali že pri nalogi 2002.3.3. Izkaže se na primer, da pri 1000 vžigalicah obstaja približno $2{,}4 \cdot 10^{31}$ razporedov, pri 10000 vžigalicah pa že $3{,}6 \cdot 10^{106}$ razporedov. Izkaže se, da velja $\ln f(n, n) = O(\sqrt{n})$.

vžigalic, lahko preostanek stanja zgradimo na $g(n-k, k, y \operatorname{xor} k)$ načinov; če pa je v prvi vrstici manj kot k vžigalic, lahko celo stanje zgradimo na $g(n, k-1, y)$ načinov. Torej je $g(n,k,y) = g(n-k, k, y \operatorname{xor} k) + g(n, k-1, y)$. Robni primeri so zdaj $g(0,0,0) = 1$, $g(n,0,y) = 0$ za $n \neq 0$ ali $y \neq 0$, $g(n,k,y) = 0$ za $n < 0$. Koristno je kot robni primer upoštevati tudi dejstvo, da je $g(n,k,y) = 0$, če je najvišji prižgani bit v y višji od najvišjega prižganega bita v k (ker z xoranjem samih števil, manjših ali enakih k, ne bo mogoče prižgati tistega najvišjega bita v y). Tudi funkcijo g lahko računamo z rekurzijo ali pa s pomočjo tabele (po naraščajočih k in pri vsakem k po vseh primernih n in y, kar nam bo vzelo $O(n^3)$ časa in $O(n^2)$ pomnilnika).

Vrednost $g(n,n,0)$ nam torej pove, koliko je vseh razporedov n vžigalic, ki imajo $y = 0$. Tu so všteti vsi nezmagovalni razporedi z $m > 1$; če je n pozitiven in sod, je v tem številu zajet tudi razpored, ki ima vsako vžigalico v svoji vrstici ($m = 1$, $y = 0$), ki pa je zmagovalen, zato je pravo število nezmagovalnih razporedov enako $g(n,n,0) - 1$; če pa je n lih, je pravo število nezmagovalnih razporedov $g(n,n,0) + 1$, ker v $g(n,n,0)$ ni vštet razpored, ki ima vsako vžigalico v svoji vrstici ($m = 1$, $y = 1$).



Na vodoravni osi obeh grafov je število vžigalic (n). Levi graf kaže, koliko je vseh razporedov n vžigalic v vrstice (polna črta) in koliko od teh razporedov je neugodnih za igralca, ki je prvi na potezi (črtkana črta). Desni graf kaže delež neugodnih razporedov v primerjavi z vsemi razporedi; če nanj položimo premico, vidimo, da je pri n vžigalicah delež neugodnih razporedov približno $0{,}92/n^{0{,}61}$. Pri obeh grafih so upoštevani le sodi n, saj je pri lihih n neugoden razpored vedno en sam (namreč tisti, ki ima v vsaki vrstici le po eno vžigalico).

Gornja grafa kažeta, kako narašča z n-jem število vseh razporedov, kako narašča število neugodnih razporedov in kako pada delež neugodnih razporedov v primerjavi z vsemi. Vidimo lahko, da je neugodnih razporedov v primerjavi z vsemi razmeroma malo; to je pravzaprav razumljivo, saj že iz pravil igre sledi, da če je neko stanje x nezmagovalno, so zmagovalna vsa stanja, ki jih lahko dobimo tako, da stanju x v eno od vrstic dodamo poljubno število vžigalic

(kar pa lahko storimo na veliko načinov). Če bi torej začetni razpored vžigalic izbirali naključno in imeli dva igralca, ki znata poiskati zmagovalno strategijo in igrati po njej, bi bila igra zelo nezanimiva, saj bi skoraj vedno zmagal prvi igralec. Pri 50 vžigalicah je od 204 226 možnih stanj le 18 437 stanj neugodnih za prvega igralca (to je približno 9,03 %); pri 100 vžigalicah pade ta delež na 5,69 % (10,8 od 190,6 milijonov stanj); pri 1 000 vžigalicah je neugodnih le še 1,38 % stanj ($3{,}3 \cdot 10^{29}$ od $2{,}4 \cdot 10^{31}$); pri 10 000 vžigalicah pa le še 0,35 % ($1{,}3 \cdot 10^{104}$ od $3{,}6 \cdot 10^{106}$).

R2004.X.7 Trikotniki

N: 631 Trikotnik $\triangle ABC$ je sestavljen iz treh daljic: AB, AC in BC. Takšne trikotnike lahko torej odkrijemo tako, da se postavimo v neko daljico (na primer AB) in pregledamo vse točke, ki so povezane s kakšnim od njenih krajišč (v našem primeru sta krajišči A in B); če je kakšna točka povezana z obema krajiščem (torej če imamo na primer daljici AC in BC), smo odkrili trikotnik (v našem primeru $\triangle ABC$).

Naloga pravi, da ne smemo izpisovati trikotnikov, ki so navznoter razdeljeni na več manjših trikotnikov. Če je trikotnik $\triangle ABC$ tako razdeljen na manjše trikotnike, mora biti vsaka od njegovih stranic tudi stranica enega od manjših trikotnikov; daljica AB mora biti na primer del nekega trikotnika $\triangle ABD$, točka D pa mora ležati v notranjosti trikotnika ABC (saj smo rekli, da gre za razdrobitev trikotnika $\triangle ABC$ na manjše trikotnike).

Ko torej pri daljici AB odkrivamo vse trikotnike, ki vsebujejo stranico AB, moramo pri vsakem preveriti, če vsebuje katerega od prej odkritih trikotnikov ali pa je sam vsebovan v njem — v tem primeru večjega od obeh trikotnikov zavržemo. Daljica AB je lahko vključena v največ dveh takih trikotnikih, ki nista navznoter razdeljena na več manjših (po en trikotnik na vsaki strani daljice AB).

Označimo krajišča daljic s $t_1, \ldots, t_n$. Za vsako od teh točk naj bo $N(t_i)$ množica tistih točk t_j, za katere obstaja daljica od t_i do t_j. Zdaj lahko zapišemo postopek za reševanje naloge:

za vsako daljico (t_i, t_j):
  $u :=$ **nil**; $v :=$ **nil**;
  za vsako točko $t \in N(t_i) \cap N(t_j)$:
    if $u =$ **nil then** $u := t$
    else if $t \in \triangle t_i t_j u$ **then** $u := t$
    else if $u \in \triangle t_i t_j t$ **then continue**
    else if $v =$ **nil then** $v := t$
    else if $t \in \triangle t_i t_j v$ **then** $v := t$
    else if $v \in \triangle t_i t_j t$ **then continue**
    else napaka v podatkih (daljice se križajo);

if $u \neq$ **nil then** izpiši $\triangle t_i t_j u$;
if $v \neq$ **nil then** izpiši $\triangle t_i t_j v$;

Pri izpisu moramo paziti, da ne izpišemo istega trikotnika po večkrat; pomagamo si lahko z oštevilčenjem točk. Rekli smo, da smo točke oštevilčili kot $t_1, \ldots, t_n$; če imamo trikotnik $\triangle t_i t_j t_k$, ga izpišimo le, če je $k < i$ in $k < j$. To nam zagotavlja, da ga bomo izpisali enkrat samkrat, čeprav bomo nanj naleteli trikrat. Naloga tudi pravi, naj bodo oglišča trikotnika izpisana v pozitivnem vrstnem redu; eden od vrstnih redov $t_i t_j t_k$ in $t_i t_k t_j$ je gotovo pozitiven (drugi pa negativen), le ugotoviti moramo, kateri. To lahko naredimo z vektorskim produktom (podprogram Ccw v spodnji rešitvi): točkam v mislih dodajmo še z-koordinato z vrednostjo 0 in izračunajmo vektorski produkt $(t_j - t_i) \times (t_k - t_i)$; če je njegova z-koordinata pozitivna, je tudi orientacija trikotnika $\triangle t_i t_j t_k$ pozitivna, sicer pa je negativne (za več o vektorskem produktu gl. rešitev naloge 2000.3.3, str. 424).

Z vektorskim produktom lahko tudi preverjamo, ali neka točka leži v danem trikotniku ali ne (podprogram LeziV v spodnjem programu). Če se postavimo v točko A in gledamo v smeri točke B, je z-koordinata vektorskega produkta $(B - A) \times (T - A)$ pozitivna, če je T na naši levi, in negativna, če je T na naši desni (če ležijo A, B in T na isti premici, bo tisti vektorski produkt enak 0). Če imamo pozitivno orientiran trikotnik ABC in se postavimo v eno oglišče in gledamo proti naslednjemu oglišču, imamo cel trikotnik na svoji levi, ne glede na to, v katero oglišče smo se postavili. Če leži T v notranjosti trikotnika in je ta vedno na naši levi, mora biti tudi T vedno na naši levi (oz. desni). Če pa leži T v zunanjosti trikotnika, jo bomo vsaj v enem primeru zagledali na desni strani trenutno opazovane stranice.

V notranji zanki zgoraj opisanega postopka naj bi šel t po vseh točkah iz $N(t_i) \cap N(t_j)$, torej po takih, ki so povezane tako s t_i kot s t_j. V praksi bi to lahko naredili takole: imejmo za vsako točko t_i in t_j seznam sosed te točke (torej takih, ki so z njo neposredno povezane z eno od daljic); pojdimo po enem od teh seznamov in tako preglejmo vse točke t, ki so povezane s t_i; za vsako od njih potem preverimo, če je povezana tudi s t_j. Pametno je iti po tistem od obeh seznamov, ki je krajši; pokazati je mogoče, da nam to zagotavlja, da bomo pri izvajanju celotnega postopka le $O(n)$-krat preverjali, ali je neka soseda tudi v drugem seznamu. To preverjanje lahko izvedemo na razne načine; pametno bi bilo uporabiti razpršeno tabelo, lahko pa na začetku sezname besed uredimo in potem po njih poizvedujemo z bisekcijo. Spodnji program uporablja še malo preprostejšo rešitev: namesto da bi šel le po krajšem od obeh seznamov, gre istočasno po obeh in ju (ker sta oba urejena) „zliva“ ter pri tem ugotavlja, katere točke so prisotne v obeh. Slabost te rešitve je, da je lahko počasna, če ima kakšna točka veliko sosed (potem moramo pri vsaki od teh njenih številnih daljic iti po celem njenem dolgem seznamu sosed — skupna časovna zahtevnost bo $O(n^2)$, pri bisekciji pa bi bila le $O(n \log n)$, pri razpršeni tabeli pa le $O(n)$,

če se daljice v tabeli lepo razpršijo). Primer takšnega neugodnega problema je „obroč“ točk, povezanih z neko osrednjo točko (kot špice pri kolesu). Še ena slabost spodnjega programa, ki bi tudi prišla do izraza pri točkah z veliko sosedami, je ta, da za urejanje seznamov sosed uporablja postopek urejanja z vstavljanjem; ta algoritem je prijetno preprost, ima pa to slabost, da je lahko v najslabšem primeru precej neučinkovit (porabi $O(k^2)$ časa za urejanje seznama dolžine k). V praksi opisane slabosti našega postopka najbrž niso prehude, saj pri delu z mrežami trikotnikov v praktičnih problemih ne pričakujemo, da bodo seznami sosed zelo dolgi (vsaka točka nastopa le v razmeroma majhnem številu daljic).

Precej resnejša slabost spodnjega programa pa je naslednja. Koordinate točk si zapisuje kar v tabeli Xi in Yi (koordinati točke t_i sta Xi[i] in Yi[i]). Pri vsaki novi točki je treba iti po celi tabeli, da preverimo, če smo točko videli že kdaj prej ali pa je nova. Če imamo n daljic in m točk, bomo za to sprehajanje po tabeli porabili v najslabšem primeru skupno $O(n \cdot m)$ časa. Ker je $m \geq n/3$,[121] bi naš postopek porabil $O(n^2)$ časa in to ne glede na to, kakšno mrežo n daljic mu damo.[122]

```
program TrikotnikiIzDaljic;
const MaxDaljic = 18000; MaxTock = 2 * MaxDaljic;
      Eps = 1e−6;
var Xi, Yi: array [1..MaxTock] of real;
    M: integer; { število točk (krajišč daljic) }

  { Koordinati točke i sta (Xi[i], Yi[i]). Spodnji podprogram za dani koordinati
    poišče indeks te točke; če take točke še nimamo v tabelah Xi in Yi, jo doda. }
  function StTocke(X, Y: real): integer;
  var i: integer;
  begin
```

[121] O tem se lahko prepričamo s pomočjo znane Eulerjeve formule: za vsak povezan ravninski graf z m točkami, n povezavami in f ploskvami velja $m - n + f = 2$. (Če graf ni povezan, ampak je sestavljen iz c ločenih kosov, je $m - n + f = c + 1$.) Med ploskve v f je zajeta tudi „zunanjost“, ki ima recimo z stranic (ker naše daljice tvorijo trikotnike oz. like, ki se jih da dobiti s stikanjem trikotnikov, je $z \geq 3$), ostale ploskve pa so pri grafih iz naše naloge sami trikotniki. Ker vsaka daljica pripada dvema ploskvama, sledi $2n = z + 3(f - 1)$. Iz tega (in iz $z \geq 3$) sledi $f \leq 2n/3$; ker je po Eulerjevi formuli $m \geq n - f$, sledi $m \geq n - 2n/3 = n/3$.

[122] Namesto navadne tabele bi lahko koordinate hranili v razpršeni tabeli, vendar bi bilo potem težje upoštevati zahtevo iz naloge, naj imamo dve koordinati za enaki, če se razlikujeta za manj kot $\varepsilon = 10^{-6}$. Ravnino lahko v mislih razdelimo na celice velikosti $\varepsilon \times \varepsilon$ in vidimo, da se točke, ki se po obeh koordinatah od naše (x, y) razlikujejo za manj kot ε, nahajajo bodisi v isti celici kot ona bodisi v eni od osmih okoliških celic. Pri poizvedovanju po razpršeni tabeli bi tako namesto koordinat točke uporabljali koordinate celice. (Testni primeri, ki smo jih imeli pripravljene za to nalogo, sicer niso zahtevali takšnega kompliciranja, saj so imele koordinate vseh točk le šest števk za decimalno vejico, torej bi jih lahko pred računanjem razprševalnih kod preprosto pomnožili z 10^6 in tako dobili cela števila.) Namesto razpršene tabele bi lahko uporabili tudi kakšno drevesasto strukturo, na primer k-d-drevo, štiriško drevo ali pa R-drevo (gl. rešitev naloge 2004.2.3, str. 603).

```
      for i := 1 to M do
        if (Abs(X − Xi[i]) < Eps) and (Abs(Y − Yi[i]) < Eps)
        then begin StTocke := i; exit end;
      M := M + 1; Xi[M] := X; Yi[M] := Y; StTocke := M;
    end; {StTocke}

    { Spodnja funkcija ugotovi orientacijo trikotnika ABC (pozitivna je, če so
      oglišča navedena v smeri, nasprotni smeri urinega kazalca). }
    function Ccw(A, B, C: integer): integer; { ccw = counterclockwise }
    var ABx, ABy, ACx, ACy, Z: real;
    begin
      ABx := Xi[B] − Xi[A]; ABy := Yi[B] − Yi[A]; { AB = vektor od A do B }
      ACx := Xi[C] − Xi[A]; ACy := Yi[C] − Yi[A]; { AC = vektor od A do C }
      Z := ABx * ACy − ACx * ABy; { vektorski produkt AB in AC je (0, 0, Z) }
      if Z > Eps then Ccw := 1 { pozitivna orientacija }
      else if Z < Eps then Ccw := −1 { negativna orientacija }
      else Ccw := 0; { točke so kolinearne, trikotnik je izrojen }
    end; {Ccw}

    function LeziV(T, A, B, C: integer): boolean; { Ali leži T v trikotniku ABC? }
    var i: integer;
    begin
      if Ccw(A, B, C) < 0 then begin i := B; B := C; C := i end;
      LeziV := (Ccw(A, B, T) >= 0) and (Ccw(B, C, T) >= 0)
                                                  and (Ccw(C, A, T) >= 0);
    end; {LeziV}

{ Tri je tabela vseh najdenih trikotnikov. Vsak trikotnik ima tri stranice, vsaka
  daljica pa je lahko stranica največ dveh trikotnikov, torej je število
  trikotnikov ≤ 2/3 števila daljic in spodnja tabela bo gotovo dovolj velika. }
var nTri: integer; Tri: array [1..MaxDaljic, 1..3] of integer;

    procedure ShraniTrikotnik(A, B, C: integer);
    var i: integer;
    begin
      if C = 0 then exit;
      { Vsak trikotnik bomo našli trikrat: enega od ABC in BAC;
        enega od BCA in BAC; in enega od CAB in ACB. }
      if (C > B) or (C > A) then exit;
      { Pri izpisu zahtevamo pozitivno orientacijo.
        Če je trenutno orientiran negativno, zamenjajmo dve oglišči. }
      if Ccw(A, B, C) < 0 then begin i := B; B := C; C := i end;
      { Zapomnimo si novi trikotnik. }
      nTri := nTri + 1; Tri[nTri, 1] := A; Tri[nTri, 2] := B; Tri[nTri, 3] := C;
    end; {ShraniTrikotnik}

var
    { Točka i ima StSosed[i] sosed, namreč točke Sosede[PrvaSoseda[i] + j]
      za j = 0, 1, ..., StSosed[i] − 1. }
```

```
  StSosed, PrvaSoseda: array [1..MaxTock] of integer;
  Sosede: array [1..2 * MaxDaljic] of integer;
  Krajisca: array [1..MaxDaljic, 1..2] of integer; { krajišči vsake daljice }
  N: integer; { število daljic }
  i, j, u, v, iu, iv, w, ww, Vrh1, Vrh2: integer;
  X1, Y1, X2, Y2: real; T: text;
begin {TrikotnikiIzDaljic}
  { Preberimo podatke o daljicah, izračunajmo stopnje. }
  Assign(T, 'tri.in'); Reset(T); ReadLn(T, N); M := 0;
  for u := 1 to 2 * N do StSosed[u] := 0;
  for i := 1 to N do begin
    ReadLn(T, X1, Y1, X2, Y2);
    u := StTocke(X1, Y1); v := StTocke(X2, Y2);
    Krajisca[i, 1] := u; Krajisca[i, 2] := v;
    StSosed[u] := StSosed[u] + 1;
    StSosed[v] := StSosed[v] + 1;
  end; {for i}
  Close(T);
  { Pripravimo sezname sosed. }
  PrvaSoseda[1] := 1;
  for u := 1 to M - 1 do PrvaSoseda[u + 1] := PrvaSoseda[u] + StSosed[u];
  for u := 1 to M do StSosed[u] := 0;
  for i := 1 to N do begin
    u := Krajisca[i, 1]; v := Krajisca[i, 2];
    Sosede[PrvaSoseda[u] + StSosed[u]] := v; StSosed[u] := StSosed[u] + 1;
    Sosede[PrvaSoseda[v] + StSosed[v]] := u; StSosed[v] := StSosed[v] + 1;
  end; {for i}
  { Uredimo vsak seznam sosed. }
  for u := 1 to M do begin
    for i := 1 to StSosed[u] - 1 do begin
      j := i - 1; v := Sosede[PrvaSoseda[u] + i];
      while j >= 0 do begin
        if Sosede[PrvaSoseda[u] + j] <= v then break;
        Sosede[PrvaSoseda[u] + j + 1] := Sosede[PrvaSoseda[u] + j]; j := j - 1;
      end; {while}
      Sosede[PrvaSoseda[u] + j + 1] := v;
    end; {while}
  end; {for i}
  nTri := 0;
  for i := 1 to N do begin
    u := Krajisca[i, 1]; v := Krajisca[i, 2];
    { Našli smo daljico u—v. Zlijmo seznama sosed u in v (z iu se sprehajamo
      po enem seznamu, z iv pa po drugem); vsaka w, ki je v obeh seznamih,
      tvori skupaj z u in v trikotnik. Izmed teh trikotnikov si moramo zapomniti
      tiste, ki ne vsebujejo nobenega manjšega trikotnika. Taka sta največ dva
```

```
      in njuna w-ja si bomo zapomnili v spremenljivkah Vrh1 in Vrh2. }
    iu := 0; iv := 0; Vrh1 := 0; Vrh2 := 0;
    while (iu < StSosed[u]) and (iv < StSosed[v]) do begin
      w := Sosede[PrvaSoseda[u] + iu]; ww := Sosede[PrvaSoseda[v] + iv];
      if w < ww then iu := iu + 1
      else if w > ww then iv := iv + 1
      else begin { Imamo tri daljice, ki tvorijo trikotnik uvw. }
        if Vrh1 = 0 then Vrh1 := w
        else if LeziV(w, u, v, Vrh1) then Vrh1 := w
        else if LeziV(Vrh1, u, v, w) then begin end
        else if Vrh2 = 0 then Vrh2 := w
        else if LeziV(w, u, v, Vrh2) then Vrh2 := w
        else if LeziV(Vrh2, u, v, w) then begin end
        else WriteLn('Napaka: daljice se križajo!');
        iu := iu + 1; iv := iv + 1;
      end; {if}
    end; {while}
    ShraniTrikotnik(u, v, Vrh1); ShraniTrikotnik(u, v, Vrh2);
  end; {for i}
  Assign(T, 'tri.out'); Rewrite(T); WriteLn(T, nTri);
  for i := 1 to nTri do begin
    WriteLn(T);
    for j := 0 to 3 do
      WriteLn(T, Xi[Tri[i, (j mod 3) + 1]]:0:6, ' ', Yi[Tri[i, (j mod 3) + 1]]:0:6);
  end; {for i}
  Close(T);
end. {TrikotnikiIzDaljic}
```

Viri dodatnih nalog: pretvarjanje znakov Unicode — Gorazd Božič; ribe — Gašper Fele Žorž; mobilni milijonar — Boris Horvat; ražnjič — Jure Leskovec; zamenjave — Ivo List; kocka Sierpińskega — Mojca Miklavec; graf signala, trikotniki — Marjan Šterk; ugibanje nizov — Dorian Šuc; vžigalice — Miha Vuk; stikala — Dare Zupanič; cezarjev kod — Anže Žagar; palindromi, seštevanje ulomkov — Klemen Žagar; urejanje logičnih vrednosti, nabori znakov, hiperkocka in mreža, števila zveri — Janez Brank.

Tekmovanja v poznavanju Unixa

LETO 1999, TEKMOVANJE V POZNAVANJU UNIXA

Pravila

Pri vseh nalogah lahko uporabiš ukaze ukaznih lupin (`csh`, `sh`, `bash`, `ksh`...), skriptnih jezikov (`sed`, `awk`, `perl`...) ali običajnih programov, ki sestavljajo sistem UNIX, priporočeno po standardu POSIX.1. Višjih jezikov (C, pascal, fortran...) ni dovoljeno uporabiti.

Če si v dvomu, ali si uporabil dovoljena sredstva, lahko kadarkoli povprašaš nadzorno komisijo. Odločitev nadzorne komisije je dokončna.

1999.U.1 Besedna analiza

R: 694 Naredi preprosto frekvenčno analizo besedila. Preberi datoteko in na standardni izhod izpiši seznam vseh besed in njihovih frekvenc. Seznam naj bo urejen po vrsti od najmanj frekventnih besed do najbolj frekventnih. Frekvenca pomeni, kolikokrat v datoteki se beseda pojavi. V datoteki ni drugih znakov razen presledkov in črk.

1999.U.2 `vi`

R: 696 Na sistemu z veliko uporabniki se želiš izogniti temu, da bi isto datoteko z urejevalnikom `vi` odprl več kot en uporabnik hkrati. Predlagaj rešitev! Rešitev zapiši v obliki potrebnih ukazov. Komentiraj, kaj so po tvojem mnenju prednosti in slabosti tvojega predloga. Predpostaviti smeš, da vsi uporabniki kličejo urejevalnik takole: `vi datoteka`. Urejevalnik `vi` sam po sebi ne opozori, če je neko datoteko že odprl kdorkoli drug.

1999.U.3 Premešaj

R: 699 V neki datoteki so vrstice urejene po določenem kriteriju. Ta urejenost te moti, zato želiš vrstice psevdonaključno razmešati. Napiši kodo, ki bo to storila. Bodi pozoren na učinkovitost in elegantnost svojega predloga. Psevdonaključno pomeni, da smeš uporabiti generator naključnih števil, ki ti je v tvojem orodju na voljo.

1999.U.4 Številke IP

V tekstovni datoteki so na več mestih zapisani številski naslovi IP, ki jih želiš spremeniti v polnovredno ime računalnika (FQDN, angl. fully qualified domain name). R: 700

V bazi `/etc/hosts` so po vrsticah navedeni številski naslov in njegovo polnovredno ime:

```
193.2.1.72 nanos.arnes.si
```

V datoteki razen takih zapisov ni ničesar drugega.

Številski naslov IP je lahko oblike: 0.0.0.0–255.255.255.255. Brez škode za splošnost lahko predpostaviš, da v tvoji datoteki vsak zapis oblike 0.0.0.0–999.999.999.999 predstavlja številski naslov IP in da imajo vsi v datoteki zapisani številski naslovi pripadajoča polnovredna imena v bazi. Upoštevaj še, da se naslovi IP ne stikajo z drugimi znaki razen s presledki.

LETO 2000, TEKMOVANJE V POZNAVANJU UNIXA

Pravila

Pri vseh nalogah lahko uporabiš ukaze ukaznih lupin (`csh`, `sh`, `bash`, `ksh`...), skriptnih jezikov (`sed`, `awk`, `perl`...) ali običajnih programov, ki sestavljajo sistem UNIX, priporočeno po standardu POSIX.1. Višjih jezikov (C, pascal, fortran...) ni dovoljeno uporabiti.

Če si v dvomu, ali si uporabil dovoljena sredstva, lahko kadarkoli povprašaš nadzorno komisijo. Odločitev nadzorne komisije je dokončna.

2000.U.1

Napiši programček, ki bere vhodno datoteko in jo na standarno izhodno enoto izpiše tako preurejeno, da so besede v vsaki vrstici razvrščene v obratnem vrstnem redu kot v izvorni datoteki.[123] Besede so v vrstici medsebojno ločene s po enim presledkom. Datoteko z vsebino R: 701

```
prva druga tretja
alfa beta gama delta
```

naj programček izpiše takole preurejeno:

```
tretja druga prva
delta gama beta alfa
```

Tekmovanje v poznavanju Unixa 2000 so pripravili: Aleš Košir, Jure Koren, Roman Maurer, Borut Mrak, Primož Peterlin, Marko Samastur in Boštjan Slivnik.

[123] Podobna naloga je tudi 1988.1.2 (str. 22), rešitev na str. 29.

R: 702

2000.U.2

Na datotečnem sistemu našega strežnika moramo vsako noč pobrisati vse datoteke `core`, ki so nastale čez dan. Za izvajanje ob določeni uri poskrbi ukaz `cron`, mi pa moramo napisati skripto, ki se pokliče enkrat dnevno in pobriše vse datoteke `core`. Strežnik ima samo en datotečni sistem, zato lahko iščeš od korenskega imenika naprej. Vse datoteke `core` imajo v imenu besedico `core`, zaneseš pa se lahko tudi na to, da program `file` pravilno prepozna vse tipe datotek in da imena datotek ne vsebujejo nenavadnih znakov, samo `A`–`Z`, `a`–`z`, `0`–`9` in `_`.

R: 703

2000.U.3

Opazili ste, da nekateri programi zelo slabo tvorijo datoteke v zapisu HTML, tako da datoteke vsebujejo prazne elemente, kakršen je na primer element `<B></B>`. Sestavi skripto, ki bo iz datoteke `.html` odstranil vse prazne oznake.

Pri tem smeš predpostaviti:

- oznaki za začetek in konec elementa vselej nastopata v isti vrstici;
- znaka `<` in `>` ne nastopata v datoteki nikjer, razen v oznakah elementov;
- različne oznake se skladno s standardom HTML ne križajo;
- oznake za začetek in konec elementov so vselej zapisane z enakimi črkami, na primer takole: `<oZnaKa></oZnaKa>`;
- elementi niso nikjer gnezdeni tako, kot kaže primer: `<PRVA><DRUGA></DRUGA></PRVA>`.

R: 703

2000.U.4

Zapiši vse permutacije besed, ki so podane v edini vrstici vhodne datoteke. Besede so medsebojno ločene s po enim presledkom in so različne. Permutacije lahko izpišeš v poljubnem vrstnem redu. Njihovo število je enako $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, če je n število besed.

Vse permutacije dveh besed `dan` in `noč` so videti takole:

```
dan noč
noč dan
```

Pri treh besedah `jutro`, `dan` in `noč` pa so permutacije:

```
dan jutro noč
dan noč jutro
jutro dan noč
jutro noč dan
noč dan jutro
noč jutro dan
```

LETO 2001, TEKMOVANJE V POZNAVANJU UNIXA

2001.U.1

R: 706

Napiši program, ki bo kot argument vzel imeni dveh obstoječih datotek in ne bo nič izpisal, če bo vsebina datotek popolnoma enaka; če bo vsebina različna, pa bo izpisal na standardni izhod neprazno vrstico. Predpostavi, da argumenta označujeta dve datoteki in da ti zanesljivo obstajata.

Primer:

```
./naloga1 datoteka1 datoteka2
```

2001.U.2

R: 707

Zaradi vse večjega števila uporabnikov internetnih storitev in pomanjkljivih varnostnih ukrepov se je razpaslo kar nekaj virusov, ki se razmnožujejo tako, da se razpošljejo z elektronsko pošto. Uporabnik Janez ima v imeniku pisma shranjena v zapisu `Maildir`, pri katerem je vsako sporočilo shranjeno v svoji datoteki. Vsako sporočilo ima glavo in telo. Glava sporočila je od telesa ločena s prazno vrstico. Naslov sporočila je v glavi sporočila in se prične z nizom `Subject:`, ki je čisto na začetku vrstice. Naslov se ne razteza čez več vrstic.

Napiši program, ki kot argument sprejme ime datoteke, jo pregleda in izpiše znak 1, če je sporočilo okuženo z virusom, sicer pa ne izpiše nič.

Sporočilo pa je okuženo, če se naslov sporočila prične z nizom `I LOVE YOU`.

2001.U.3

R: 707

Napiši program, ki prešteje vse izvedljive datoteke glede na vsebino spremenljivke okolja `$PATH`. Razmisli o tem, da je v poti kateri od imenikov lahko naveden večkrat. Nekatere datoteke morda lahko izvaja samo sistemski skrbnik, mi pa ne; takih ne smemo šteti. V imenikih so poleg navadnih datotek tudi simbolne povezave na druge datoteke, ki jih moramo tudi všteti. Ne smemo pa seveda šteti simbolnih povezav, ki kažejo na neobstoječe datoteke ali na datoteke, za katere nimamo dovoljenja, da bi jih poganjali.

2001.U.4

R: 709

V datoteki `stevila.txt` je $2n$ nenegativnih celih števil, vsako v svoji vrstici.

Napiši program, ki bo iz njih tvoril n parov števil tako, da bo vsota zmnožkov parov števil najmanjša.

V izhodni datoteki morata biti števili v paru ločeni s presledkom, vsak par pa mora biti v novi vrstici. Vrstni red parov v izhodni datoteki ni pomemben.

Vhodna datoteka se imenuje `stevila.txt`. Primer vhodne datoteke:

```
7
4
```

```
0
2
2
1
```

Izhodna datoteka (torej tista, ki jo mora narediti tvoj program), naj se imenuje `pari.txt`.

Primer izhodne datoteke (`pari.txt`):

```
0 7
1 4
2 2
```

LETO 2002, TEKMOVANJE V POZNAVANJU UNIXA

Navodila

Naloge boste pisali v tekstovne datoteke z urejevalnikom po svoji izbiri. Za pošiljanje rešitve naloge številka N uporabite skript `submit`N, ki mu podate ime datoteke z rešitvijo. Če rešitev nikakor ne ustreza, vam bo program javil „`Naloga je zavrnjena`“; če popolnoma ustreza, dobite 10 točk. Če rešitev ni popolnoma ustrezna, lahko dobite manj kot 10 točk. Vsako nalogo lahko pošljete poljubno mnogokrat. Seštevek svojih točk lahko pregledate z ukazom `score`. V primeru enakega skupnega števila točk bo komisija ocenjevala tudi razumljivost rešitve.

R: 710

2002.U.1

Delo skrbnice računalniških sistemov Metke je široko. Ena izmed njenih nalog je dodeljevanje internetnih naslovov računalnikom, za katere skrbi. Da bi si olajšala delo, je Metka sestavila program, ki pove, ali se izbrano podomrežje prekriva z že dodeljenim podomrežjem.

Podomrežje je opisano s skupino štirih polj, ločenih s piko. Polja so opisana s števili od vključno 0 do vključno 255 ali pa z intervalom, ki vključuje ta števila. Interval je označen z znakom minus (`-`) med dvema številoma. Namesto kateregakoli polja je lahko znak zvezdica (`*`), ki označuje vsa števila z intervala.

Primeri tako opisanih podomrežij so:

```
192.168.1.1
192.168.0.1-3
0-255.*.255.*
```

Pomagajte Metki in sestavite program, ki v ukazni vrstici sprejme kot parametra dve podomrežji, opisani na zgornji način.

Program naj vrne izhodno vrednost (*exit status*):

Tekmovanje v poznavanju Unixa 2002 so pripravili: Saša Divjak, Jure Koren, Aleš Košir, Rok Kaver, Rok Papež, Primož Peterlin, Marko Samastur, Andraž Tori in Miha Tomšič.

0, če območji nimata skupnih naslovov (podomrežji sta tuji), kot na primer podomrežji 192.168.1–30.64 in 192.168.31–35.0;

1, če imata podomrežji natanko en skupni naslov (presek množic je natanko en element), kot na primer 10.0.0.* in 10.0.0.1;

2, če imata območji več kot en skupni naslov, kot na primer 10.*.*.* in 10.0–12.3.4;

Privzameš lahko, da so podatki pravilni.

2002.U.2

R: 711

Vrstice v besedilnih datotekah so v sistemih Unix zaključene z znakom LF. V nekaterih sistemih, na primer MS-DOS in Windows, so vrstice zaključene z dvema zaporednima znakoma CR in LF. Napiši program, ki bo v datoteki, podani z imenom v ukazni vrstici, zamenjal vse konce vrstic iz zaporedja CR LF v LF. Privzameš lahko, da znaka CR in LF vselej nastopata v paru.

V pomoč: znak CR je desetiško 13, predstavljen pa je tudi kot `^M` ali `\r`, znak LF je desetiško 10 ali `^J` ali `\n`.

Program naj se izvede takole:

```
naloga2 ime_datoteke
```

Ko se program konča, morajo biti zaključki vrstic v datoteki zamenjani. Na disku ne smejo ostati morebitne pomožne datoteke.

2002.U.3

R: 712

V računalniku hkrati teče več procesov. Vsi izvirajo iz procesa `init`. Vsak proces pa ima lahko več sinov. Procesi tako tvorijo drevesno strukturo. Procesna veriga so procesi od procesa `init` pa do zadnjega procesa, ki je brez potomca. Napiši program, ki globino najdaljše procesne verige vrne kot izhodno vrednost.

Primer:

```
init-+
     |-cron
     |-gpm
     |-httpd---10*[httpd]
     `-in.identd---in.identd---3*[in.identd]
```

Najdaljša veriga je dolžine 4.

2002.U.4

R: 712

Sčasoma se je nabralo več datotek, v katerih so zapisani statistični podatki. Ker je za datoteke skrbelo več oseb, ki so datoteke poimenovali po svoje, imena niso sistematično urejena in iz njih ni razvidno, kateremu časovnemu obdobju pripadajo.

Da bi datoteke lahko kronološko uredili, potrebujemo program, ki primerja dve datoteki in pove, katera je starejša.

Napišite program, ki prejme kot parameter dvoje imen in vrne kot izhodno vrednost:

0 — če sta datoteki enako stari,
1 — če je prva datoteka starejša od druge,
2 — če je druga datoteka starejša od prve,
3 — če je kaj narobe.

Starost datoteke je določena s trenutkom zadnje spremembe podatkov v njej.

LETO 2003, TEKMOVANJE V POZNAVANJU UNIXA

Nasvet

Pri vaših rešitvah bo komisija za ocenjevanje upoštevala poleg pravilnosti in robustnosti tudi njihovo elegantnost in preprostost.

R: 713

2003.U.1

V okolju UNIX je obdelava znakovnih podatkov zelo pomembna. Strežniški programi običajno vodijo datoteko dogodkov — dnevnik (*logfile*).

Napišite program `preglej`, ki mu podamo kot parametre imena treh datotek po vrsti, kot kaže zgled:

```
preglej datoteka1.log regexp-da.dat regexp-ne.dat
```

Prva datoteka je dnevniška z običajnim besedilom (*text file*) in vsebuje zapise dogodkov. Drugi datoteki vsebujeta vsaka po eno vrstico z regularnim izrazom. Dnevnik je potrebno prebrskati in izpisati vse vrstice, ki ustrezajo regularnemu izrazu iz prve datoteke ter ne ustrezajo regularnemu izrazu iz druge datoteke. Če morebiti katera od podanih datotek ne obstaja, naj program izpiše besedilo „`NAPAKA`“ v vrstici na standardni izhod.

Namig: Privzamete lahko, da je dnevniška datoteka vselej dostopna in na voljo, da se med tekom skripte ne dodajajo novi zapisi v dnevnik in da se skripte nikoli ne poganja večkrat hkrati.

R: 713

2003.U.2

Nekatere datoteke z večpredstavnimi vsebinami imajo glavo, ki se začne z nizom „`RIFF`“, temu pa lahko sledijo poljubni podatki, ki se končajo z nizom „`data`“. Za nizom „`data`“ je preostanek datoteke.

Tekmovanje v poznavanju Unixa 2003 so pripravili: Aleš Košir, Saša Divjak, Gašper Fele-Žorž, Boris Gašperin, Jure Koren, Rok Papež, Primož Peterlin, Marko Samastur, Andraž Tori in Miha Tomšič.

Napiši program, ki mu kot parameter podaš ime tako oblikovane datoteke, program pa bo iz nje izrezal vse podatke med nizoma „`RIFF`“ in „`data`“, vključno s tema nizoma. Če na začetku datoteke ni niza „`RIFF`“, naj ne odstani podatkov.

2003.U.3

Sistemi Linux v navideznem datotečnem sistemu `/proc` hranijo mnoge podatke o stanju sistema in programov, ki tečejo v njem. Z branjem teh podatkov znajo programi prikazati stanje sistema na različne načine. R: 714

Napišite program, ki na standardni izhod izpiše polno pot do izvršilne datoteke očeta in njeno ime. Oče je proces, ki je pognal program, ki ste ga napisali. Če kot zgled v ukazni lupini `bash` poženemo nek program `preskus` s parametri

```
preskus -v test -o izhod.dat
```

in bi ta program hotel izpisati pot do izvršilne datoteke svojega očeta, bi moral izpisati pot do lupine `/bin/bash`. Če pa bi napisali svoj program za izpisovanje poti do izvršilne datoteke očeta in ga pognali iz programa

```
preskus -v test -o izhod.dat
```

bi moral naš program izpisati pot, kjer je na disku shranjena izvršilna datoteka programa `preskus`, denimo `/usr/local/bin/preskus`. V primeru

```
/usr/bin/preskus
```

bi to bil niz

```
/usr/bin/preskus
```

V primeru

```
./preskus
```

pa je to lahko nekaj takega:

```
/home/uporabnik/preskus
```

2003.U.4

Nekateri programi so napisani tako, da delujejo kot filtri (berejo podatke s standardnega vhoda in pišejo na standardni izhod), drugim pa moramo podati vhodno in izhodno datoteko. R: 716

Program `sort`, denimo, zna delati celo na oba načina.

```
sort datoteka1 -o datoteka2
sort datoteka1 > datoteka2
```

Razliko opazimo takrat, ko sta `datoteka1` in `datoteka2` ista datoteka. Takrat druga oblika ukaza poreže datoteko, preden je urejanje končano in izgubimo njeno vsebino.

Napiši program `prepis`, ki bo prepisal vhodno datoteko na izhodno, podobno kot `cat datoteka1 > datoteka2`, na začetek datoteke pa bo dodal niz „`PREPIS`“.

Če podamo le en parameter, bo program podano datoteko izpisal na standardni izhod. Če mu podamo izhodno datoteko, bo izhod pisal nanjo. V tem primeru se bo tudi pametno odzval, kadar sta podani datoteki ista datoteka, kar pomeni, da bo vhodni datoteki na njen začetek dodal zahtevani niz.

LETO 2004, TEKMOVANJE V POZNAVANJU UNIXA

R: 718

2004.U.1

Z leti rabe računalnika ste ugotovili, da je uporabniški vmesnik s tipkovnico za normalno mislečega uporabnika neprimeren, saj se uporabnik ves čas moti pri tipkanju.

Najpogostejša napaka pri tem je zamenjava sosednjih črk v besedi, seveda poleg napak neknjižne rabe besed.

Da bi uporabniku pri tem pomagali, sestavite program, ki bo za vneseno besedo izpisal vse možnosti zamenjave dveh sosednjih črk, recimo za *bal*: *abl*, *bal*, *bla*.[124]

Program naj besede prebere iz vhodne datoteke, ki je navedena kot parameter v ukazni vrstici. V vsaki vrstici datoteke je podana ena beseda. Datoteka ne vsebuje presledkov ali nečrkovnih znakov, le črke in znake za konec vrstice. Izpisane besede naj bodo urejene bo abecednem vrstnem redu. Morebitne prazne vrstice v vhodni datoteki preskoči.

R: 719

2004.U.2

Preštejte število vseh internetnih sej, ki so zabeležene v dnevniški datoteki strežnika Apache. Privzamete lahko, da je v dnevniški datoteki na prvih mestih podano število IP računalnika, ki je sodeloval pri seji. Sledi poljubno število presledkov, nato čas dostopa, zapisan kot celo število po dogovoru standard epoch time. Ta meri število pretečenih sekund od 1. januarja 1970. Časi v datoteki le naraščajo. Kot eno sejo obravnavajte vse dostope z nekega računalnika, med katerimi ni več kot 30 minut premora. Ime datoteke naj bo dano vašemu programu kot parameter v ukazni vrstici.

Dnevniška datoteka je videti takole:

Tekmovanje v poznavanju Unixa 2004 so pripravili: Aleš Košir, Saša Divjak, Boris Gašperin, Rok Papež, Andraž Sraka, Boštjan Müller, Jure Čuhalev, Primož Bratanič, Špela Kraner, Andraž Tori, Primož Peterlin in Miha Tomšič.

[124] Kot vidimo iz tega primera, je dovoljena tudi možnost, da sploh ne izvedemo nobene zamenjave in pustimo prvotni niz pri miru. Če je mogoče na več načinov priti do istega niza (npr. če se v prvotni besedi pojavita skupaj dve enaki črki), tak niz tolikokrat tudi izpiši.

```
123.123.123.123 100000
123.123.123.123   100001
123.123.123.123 100001
192.168.0.0     100002
192.168.0.1 100000000
123.123.123.123     10000000
```

2004.U.3

Uredite vrstice vhodne datoteke v obratnem vrstnem redu in rezultat shranite v izhodno datoteko. R: 720

Vaša skripta naj sprejme dva parametra, prvi parameter je ime vhodne datoteke, drugi pa ime izhodne datoteke, v katero shranite rezultat.

Pričakujte, da vrstice v podani vhodni datoteki vsebujejo le črke slovenske abecede, in to brez šumnikov in presledkov. Če je več enakih vrstic, izpišite le eno. Vrstice v vhodni datoteki se vedno začno s črko.

Denimo, da zaradi težav v sistemu pri tem ne morete uporabiti programov `sort`, `perl`, `sed`, `awk`, `python`, `php` in morate najti nadomestno rešitev.

Obratni vrstni red pomeni obratno abecedno urejanje (sortiranje). Na primer: vrstice, v katerih so le znaki *A, B, V, Z, a, b, v, z*, se tako izpišejo kot *z, v, b, a, Z, V, B, A*.

2004.U.4

Napišite skripto, ki bo pripravila sliko za spletno galerijo. Skripta naj sprejme kot prvi parameter ime datoteke z vhodno sliko, kot drugi parameter ime izhodne slike, kot tretji parameter njeno širino (velikost x), četrti parameter pa je največja dovoljena velikost slike. R: 720

Sintaksa:

pomanjsaj ⟨*ime vhodne slike*⟩ ⟨*ime izhodne slike*⟩
⟨*širina izhodne slike*⟩ ⟨*največja velikost izhodne slike v bajtih*⟩

Skripta mora vhodno sliko pomanjšati na določeno širino. Pri tem mora ohraniti velikostno razmerje slike. Slika ne sme biti večja od predpisane velikosti, manjša pa je lahko, vendar (če je le mogoče) ne za več kot 5 %. Sliko nato zapišite v izbrano izhodno datoteko. Privzamete lahko, da slike s tem programom vselej pomanjšujete, ne povečujete. Predpostavite lahko tudi, da vaša skripta dobi za parametre smiselne vrednosti, ki ji ne bodo zastavljale nerešljivega problema (npr. zahtevale slike velikosti 10000×10000, obenem pa omejile velikost datoteke na 1 bajt).

Namig: Pri reševanju si pomagajte z ukazom `convert`.

REŠITVE NALOG PRVEGA TEKMOVANJA IZ UNIXA

N: 684

R1999.U.1

Pomagali si bomo s programi `tr`, `sort` in `uniq`, ki jih bomo povezali s cevmi (kar pomeni, da ukazna lupina ob poganjanju teh programov poskrbi za to, da se standardni izhod enega programa spelje na standardni vhod naslednjega in tako naprej). Rešitev je lahko takšna:

```
cat datoteka.txt | tr " " "\n" | sort | uniq -c | sort -n
```

Program `cat` samo bere vhodno datoteko in jo piše na svoj standardni izhod. Cev bo poskrbela, da pridejo ti podatki na standardni vhod programa `tr`, ki smo mu naročili, naj vse presledke spremeni v znake za konec vrstice. Tako pride vsaka beseda v samostojno vrstico; dobljeno besedilo pošljemo programu `sort`, ki uredi vrstice po abecedi. Zdaj torej pridejo vse pojavitve posamezne besede skupaj. Program `uniq` bere svoj vhod in če je več zaporednih vrstic enakih, izpiše od takšne skupine le eno vrstico; s stikalom `-c` smo zahtevali, naj izpiše še število vrstic v skupini. Zdaj torej dobimo v vsaki vrstici niz oblike „`123 bla`“, ki nam pove, da se je beseda `bla` v vhodni datoteki pojavila 123-krat. Posledica tistega prvega urejanja je tudi to, da so besede tu navedene po abecedi; ker pa bi jih radi uredili po frekvenci, pokličimo `sort` še enkrat. Tokrat mu s stikalom `-n` povemo, naj ignorira morebitne presledke na začetku vrstice (ki jih je mogoče izpisal `uniq`) in nato začetek vrstice gleda kot število, ne kot niz (drugače bi namreč lahko ugotovil, da je na primer `10` manjše od `2`, ker bi tadva niza primerjal znak po znak in videl, da je `1` leksikografsko pred `2`).

Opisana rešitev se zanaša na dejstvo, da so v vhodni datoteki le črke in presledki (kot zagotavlja besedilo naloge). Če bi bilo v datoteki še kaj drugega, na primer ločila, bi bilo razbijanje na besede malo bolj zapleteno; za prvo silo bi lahko na primer vse ne-črke spremenili v presledke:

```
sed "s/[^A-Za-z]/ /g"
```

Tu smo uporabili program `sed`; ukaz `s/`*vzorec*$_1$`/`*vzorec*$_2$`/`*zastavice* zamenja pojavitve vzorca 1 z vzorcem 2; zastavica `g` zahteva zamenjavo vseh pojavitev (namesto samo prve). Regularni izraz `[^A-Za-z]` se ujame z vsakim znakom, ki ni ena od črk *A, . . . , Z* ali *a, . . . , z*.

Naša rešitev je zaenkrat tudi občutljiva na razlike med velikimi in malimi črkami; tako sta na primer *Bla* in *bla* zanjo dve povsem različni besedi. Če bi hoteli pred obdelavo spremeniti velike črke v male, bi si spet lahko pomagali s programom `tr`:

```
tr [:upper:] [:lower:]
```

`tr` v splošnem vzame kot parametra dva niza, nato pa bere standardni vhod in vsako pojavitev kakšne črke prvega niza zamenja z istoležno črko drugega niza; rezultat izpisuje na standardni izhod. Parameter [:upper:] je enakovreden nizu „`ABC...XYZ`“, podobno pa [:lower:] nizu „`abc...xyz`“.

Še ena morebitna slabost naše rešitve bi se pojavila pri delu z dolgimi besedili. Recimo, da imamo besedilo z n besedami, od katerih je k različnih. Naša rešitev bi za urejanje (prvi klic programa `sort`) porabila $O(n)$ dodatnega pomnilnika (ali prostora na disku) in, če uporablja `sort` katerega od splošnonamenskih postopkov za urejanje, tudi $O(n \log n)$ časa. Pri dovolj velikih n bi torej lahko postal učinkovitejši naslednji postopek: besedilo berimo vrstico po vrstico in ga sproti režimo na besede, te pa shranjujmo v razpršeni tabeli, kjer ob vsaki besedi tudi piše, kolikokrat se pojavlja. V razpršeni tabeli je torej vsaka različna beseda omenjena le enkrat, zato je poraba pomnilnika le $O(k)$. Urejanje potrebujemo zdaj le na koncu, da uredimo besede po frekvenci; ker imamo takrat vsako besedo v seznamu omenjeno le enkrat, porabimo za to le $O(k \log k)$ časa. Če prištejemo še čas, potreben za branje vhodnega besedila, imamo skupno zahtevnost $O(n + k \log k)$. Lepo pri tem je, da je k (število različnih besed) v praksi precej manjši od n (števila vseh besed), saj se mnoge besede pojavljajo po večkrat (in to nekatere zelo velikokrat). Znani Heapsov zakon ugotavlja, da je k približno enak $c \cdot n^d$ za neki konstanti c in d (ki sta odvisni od jezika in vrste besedil, s katerimi delamo); glavno je, da je d običajno precej manjši od 1. Za poskus smo vzeli zbirko 806 791 Reutersovih člankov in opazovali, kako se povečuje k, če gledamo vse več člankov (in s tem povečujemo n); izkazalo se je, da je $k \approx 39 \cdot n^{0{,}48}$ (cela zbirka ima približno 182 milijonov besed, od tega 380 tisoč različnih). Tukaj lahko torej rečemo, da je $k = O(\sqrt{n})$.

Sledi primer rešitve z razpršeno tabelo:

```
import sys
frekvence = {}
for vrstica in sys.stdin:
  for beseda in vrstica.split():
    try: frekvence[beseda] += 1
    except: frekvence[beseda] = 1
seznam = [(frekvence[beseda], beseda) for beseda in frekvence]
seznam.sort()
for (frekvenca, beseda) in seznam: print "%s %d" % (beseda, frekvenca)
```

Za razbijanje vrstice na besede (pri presledkih) smo uporabili pythonovo funkcijo split. Pri dostopu do razpršene tabele moramo posebej obravnavati primer, ko na neko besedo naletimo prvič; takrat je v razpršeni tabeli še ni. Stavek `frekvence[beseda] += 1` sproži takrat izjemo (*exception*), ker bi moral najprej prebrati iz razpršene tabele vrednost, ki pripada tej besedi, da bi jo nato lahko povečal za 1. To izjemo prestrežemo (stavek **try**...**except**) in v tem pri-

meru preprosto vpišemo besedo v razpršeno tabelo z začetno frekvenco 1 (ker smo pravkar videli njeno prvo pojavitev). Namesto te rešitve s **try**...**except** bi lahko tudi eksplicitno preverili, če je beseda že v razpršeni tabeli:

```
if beseda in frekvence: frekvence[beseda] += 1
else: frekvence[beseda] = 1
```

Ali pa bi uporabili metodo get, ki ji lahko povemo, kakšno vrednost naj vrne, če besede še ni v tabeli:

```
frekvence[beseda] = frekvence.get(beseda, 0) + 1
```

Vendar se izkaže, da je zadnja različica približno 15 % počasnejša.

Za primerjavo smo pognali obe rešitvi, torej tisto s `sort` in `uniq` ter tisto z razpršeno tabelo, na nekaj dolgih angleških besedilih. Izkaže se, da je rešitev z razpršeno tabelo hitrejša šele pri besedilih, dolgih več deset milijonov znakov, vendar je to verjetno deloma tudi posledica neučinkovitosti pythonovega interpreterja.

Besedilo	Dolžina	Št. besed vseh	Št. besed različnih	Čas izvajanja `sort` + `uniq`	Čas izvajanja razp. tabela
Gibbonova *Zgodovina*	9,0 MB	1,5 M	55 K	4,3 s	5,9 s
Dickensova dela	18,9 MB	3,6 M	39 K	10,1 s	9,9 s
Reutersovi članki 1/8	145 MB	22,9 M	160 K	81 s	59 s
Reutersovi članki	1147 MB	183 M	381 K	1129 s	462 s

Besedila so ista, kot smo jih že uporabili pri poskusih v rešitvi naloge 1989.2.3 (str. 58).

N: 684

R1999.U.2

Pri tej nalogi moramo poznati ali iznajti pojem zaklepanja. Ko en uporabnik dela z datoteko, mora biti za druge nekako „zaklenjena“, tako da ne bodo mogli do nje (oz. bodo lahko vsaj opazili, da bi bilo bolje, če je ne bi odpirali).

Rešitev z nevsiljenim zaklepanjem. Izvršilno datoteko `vi` preimenujmo v `vi_old` in jo zamenjajmo z našo skripto:

```
#!/usr/bin/perl
use Fcntl ':flock';   # definicija konstant LOCK_*
$preimenovan_vi = "vi_old";
$ime_datoteke = $ARGV[0];
$ime_lock_datoteke = $ime_datoteke . ".lock";
open(FH, ">$ime_lock_datoteke");
$return = flock(FH, LOCK_EX | LOCK_NB);
if (! $return) {
```

```
  print "Čakam, da se datoteka \"$ime_datoteke\"odklene.\n"
  print "Za prekinitev pritisnite CTRL+C.\n";
  flock(FH, LOCK_EX);
}
system($preimenovan_vi . " " . $ime_datoteke);
flock(FH, LOCK_UN);
close(FH);
```

Skripta uporablja sistemski klic flock, ki zagotavlja nevsiljeno (advisory) zaklepanje datotek (torej lahko nek drug proces takšno datoteko še vseeno odpre, npr. s funkcijo open, četudi jo je nek drug proces ekskluzivno zaklenil s flock). Pred klicem urejevalnika besedila `vi` se tako ustvari datoteka-ključavnica, ki nakazuje, da je datoteka, ki jo urejamo, zaklenjena. Klicu flock podamo zastavici LOCK_EX, ki zahteva izključni dostop do ključavnice, in LOCK_NB, ki mu pove, naj ne čaka, da se bo ključavnica sprostila, če je jo je zasegel že kdo drug. Zato lahko iz vrednosti, ki jo flock vrne, ugotovimo, če je uspel ključavnico zaseči ali ne; če je ni, uporabnika obvestimo, da bo treba čakati, in pokličemo flock še enkrat, tokrat brez LOCK_NB.

Zaklepno datoteko moramo odpreti v pisalnem načinu (predznak > ob odpiranju datoteke), kar datoteko tudi ustvari, če še ne obstaja. Ta način sicer ob odpiranju tudi poreže datoteko na dolžino 0 bytov (za razliko od >>), vendar nas to ne bo motilo, ker vanjo tako ali tako ne bomo ničesar pisali; glavno je, da ostane to še vseeno ista datoteka (ker če bi > obstoječo datoteko na primer pobrisal in ustvaril novo, bi bilo to za zaklepanje seveda neuporabno: ko bi nek program datoteko zaklenil, drugi tega ne bi opazili, ker sploh ne bi gledali iste datoteke).

Omeniti velja, da rešitev deluje le pod sistemi, ki imajo implementiran sistemski klic flock (linux in večina drugih unixov). Prav tako rešitev ne deluje na oddaljeno priklopljenih datotečnih sistemih NFS (in sorodnih); tam je potrebno datoteko zakleniti preko sistemskega klica fcntl.

Po svoje bi bilo lepo, če bi naš program na koncu tudi pobrisal zaklepno datoteko, da se nam ne bi take datoteke kopičile na disku, vendar pa bi utegnile biti s tem težave. „Črni scenarij“ bi bil takšen: program A odpre datoteko, jo zaklene in požene `vi`; program B jo odpre in čaka; program A se vrne iz `vi` in datoteko odklene, zapre in pobriše; B jo zaklene in požene `vi`; program C jo odpre, vidi, da je prosta, in jo tudi zaklene in požene `vi`. Kaj se je zgodilo? Ko je A pobrisal datoteko, je operacijski sistem še ni zares pobrisal, ker jo je imel B še odprto, vendar pa jo je „skril“ pred drugimi: ko poskuša C odpreti datoteko s tem imenom, dobi v resnici že novo datoteko, ne pa tiste, ki jo ima B še odprto (in zaklenjeno).

Rešitev s pomočjo dostopnih pravic. Za zaseganje zaklepne datoteke lahko namesto funkcije flock uporabimo tudi običajne unixove dostopne pravice do datoteke. Zaklepno datoteko poskusimo ustvariti tako, da bomo imeli bralni

in pisalni dostop do nje samo mi, drugi uporabniki pa ne. Če datoteko ureja že kdo drug, nam to ne bo uspelo, saj ima tisti drugi uporabnik že ekskluzivni dostop do zaklepne datoteke; tako bomo vsaj vedeli, pri čem smo. Drugače pa bomo zaklepno datoteko uspešno ustvarili in tako tudi vedeli, da lahko poženemo `vi`.

Pomembno je, da ne gremo najprej preverjat, če datoteka že obstaja, in jo šele nato poskusimo ustvariti; če bi počeli tako, bi nas lahko med našim preverjanjem in ustvarjanjem datoteke prehitel kdo drug, ki bi ravno tako opazil, da še ne obstaja, in jo nato ustvaril, še preden bi jo ustvarili mi. Namesto tega bomo datoteko kar takoj poskusili ustvariti, nato pa bomo samo pogledali, če se je to posrečilo ali ne.

```
#!/bin/bash
if ( umask u=rw,g=,o= ; touch $1.lock > /dev/null 2>&1 )
then
  vi_old $1
  rm -f $1.lock
else
  echo "Datoteka $1 je trenutno zaklenjena."
fi
```

Za ustvarjanje zaklepne datoteke smo si pomagali s programom `touch`, ki ustvari prazno datoteko, če ta prej še ni obstajala, sicer pa ji le postavi čas zadnje spremembe na trenutni čas. Pred tem smo z lupininim vgrajenim ukazom `umask` zahtevali, naj se datoteke ustvarja tako, da jih bomo lahko mi (`u`, *user*) brali in pisali (`rw`), drugi iz naše skupine (`g`, *group*) in ostali uporabniki (`o`, *others*) pa je ne bodo smeli niti brati niti pisati. S tem zagotovimo, da če nam bo datoteko uspelo ustvariti, je drugi uporabniki (razen administratorja) ne bodo mogli povoziti.

Nastavitve, ki jih podamo prek klica `umask`, bi v splošnem veljale vse do naslednje spremembe, ne le za prvi naslednji ukaz. Ker pa hočemo, da bi bila ta sprememba pri našem programu le začasna (da bi vplivala samo na program `touch`), smo `umask` in `touch` ovili v oklepaje in jima s tem dodelili ločeno podlupino, iz katere se sprememba `umask` ne vidi navzven.

Izpis programa `touch` nas ne zanima, zato smo njegov standardni tok za napake preusmerili na običajni standardni izhod (`2>&1`), tega pa na `/dev/null`. Vrednost, ki jo vrne `touch` in z njo pove, če se je zaklepne datoteke uspel „dotakniti“ ali ne, pa bomo uporabili kot pogoj v stavku **if**.

Ko se vrnemo iz programa `vi`, moramo zaklepno datoteko pobrisati, kajti dokler obstaja, je datoteka z vidika vseh ostalih uporabnikov zaklenjena. Programu `rm` s stikalom `-f` naročimo, naj nas nič ne sprašuje in naj se tudi ne zmeni za to, če bi mogoče zaklepna datoteka ne obstajala.

Lepo pri tej rešitvi je, da je preprostejša od prve in da zaklepne datoteke na koncu pobriše za sabo. Za razliko od prve pa nas ta rešitev ne varuje

pred tem, da bi isti uporabnik odprl neko datoteko iz več različnih procesov. Še posebej nerodno pri tem je, da bi prvi od teh procesov, ko bi se vrnil iz programa `vi`, zaklepno datoteko pobrisal in bi bila potem z vidika ostalih uporabnikov datoteka odklenjena, čeprav je v resnici mogoče še odprta v drugih procesih prvega uporabnika.

Mimogrede omenimo še, da za razliko od prvotne različice programa `vi` nekatere novejše različice, na primer zelo razširjeni `vim` („`vi` improved“), že same po sebi skrbijo tudi za zaklepne datoteke.

R1999.U.3

N: 684

Če je datoteka dovolj majhna, jo lahko kar celo preberemo v pomnilnik, premešamo njene vrstice in jih izpišemo v novem vrstnem redu. Primer rešitve v pythonu:

```
import sys, random
vrstice = sys.stdin.readlines()
random.shuffle(vrstice)
for s in vrstice: sys.stdout.write(s)
```

Pravzaprav bi morali paziti še na možnost, da se zadnja vrstica vhodne datoteke mogoče ne konča z znakom za konec vrstice. Ker ta vrstica po mešanju najbrž ne bo več zadnja, bi ji morali znak za konec vrstice dodati, da se ne bo v izhodni datoteki sprijela z naslednjo.

Za mešanje vrstic smo zgoraj uporabili pythonovo funkcijo shuffle iz modula random. Oglejmo si še, kako bi lahko premešali vrstice brez takšne funkcije:

```
import sys, random
vrstice = sys.stdin.readlines()
n = len(vrstice)
while n > 0:
  # Izpisati bo treba še vrstice[0], ..., vrstice[n − 1] (v naključnem vrstnem redu).
  i = random.randrange(n)    # Naključno število iz množice {0, 1, ..., n − 1}.
  sys.stdout.write(vrstice[i])
  vrstice[i] = vrstice[n − 1]; n = n − 1
```

V vsaki iteraciji glavne zanke si naključno izberemo eno od n vrstic, ki jih doslej še nismo izpisali. Izbrano vrstico izpišemo, nato pa na njeno mesto v tabeli vrstica postavimo zadnjo še neizpisano vrstico, torej niz vrstica[n − 1]. Potem lahko n zmanjšamo za 1 in postopek se nadaljuje. Z indukcijo se lahko hitro prepričamo, da ima pri takšnem postopku res vsaka permutacija naše tabele vrstic enake možnosti, da bo izbrana (če le randrange(n) res vrača vsa števila od 0 do n − 1 z enako verjetnostjo).

Slabo pri tem postopku je, da mora prebrati vse vrstice v pomnilnik, torej ga ne bi mogli uporabiti na zelo velikih datotekah. Tu bi bilo pametno narediti novo kopijo datoteke, pri kateri bi na začetek vsake vrstice vrinili neko primerno veliko psevdonaključno število. Vrstice te datoteke potem uredimo,

v urejeni datoteki porežimo števila, ki smo jih prej vrinili na začetek vsake vrstice, pa nam ostanejo vrstice prvotne datoteke v premešanem vrstnem redu. Za urejanje bi morali uporabiti kakšnega od algoritmov zunanjega (eksternega) urejanja, ki si pomagajo s pomožnimi datotekami in zato ne porabijo veliko pomnilnika; prepustimo to kar standardnemu programu `sort`. Vse opisane korake lahko opravimo kar iz ukazne lupine:

```
awk '{ print rand(), $0 }' urejena.txt | sort -n | cut -d " " -f 2-
```

Program `awk` bere vhodno datoteko (`urejena.txt`) vrstico za vrstico in na vsaki vrstici izvede stavek, ki smo mu ga podali v zavitih oklepajih. Stavek **print** v `awk`u izpiše svoje argumente, vmes po en presledek, na koncu pa prazno vrstico. Funkcija `rand`, ki je že vgrajena v `awk`, vrne naključno število med 0 in 1; v spremenljivki `$0` pa nam `awk` hrani vsebino trenutne vrstice.

Za urejanje uporabimo program `sort`, ki mu s stikalom `-n` naročimo, naj začetke vrstic pri urejanju gleda kot števila (iz enakih razlogov kot pri 1. nalogi, glej str. 694). Na koncu bi radi tista naključna števila z začetkov vrstic pobrisali, kar lahko naredimo s programom `cut`. Ta naj razreže vsako vrstico pri presledkih (`-d " "`) in izpiše vse kose od drugega naprej (`-f 2-`); završemo torej le prvi kos, ki vsebuje naključno število, ki smo ga dodali pred urejanjem.

N: 685

R1999.U.4

Najprej preberimo datoteko `/etc/hosts` in si pripravimo razpršeno tabelo, ki bo preslikovala številske naslove v imena. Potem lahko beremo vhodno datoteko (spodnji program bere kar standardni vhod) in v njej iščemo pojavitve številskih naslovov IP ter jih zamenjujemo z imeni računalnikov. Pri iskanju naslovov IP si lahko pomagamo z regularnim izrazom, saj dobro poznamo zgradbo takega naslova: sestavljen je iz štirih skupin števk, vsaka ima največ tri števke, med skupinami pa so pike; poleg tega nam naloga zagotavlja še, da se naslovi IP v vhodni datoteki ne stikajo z drugimi znaki razen s presledki.

```
#!/usr/bin/perl
open(HOSTS_FILE, "/etc/hosts");
while ($line = <HOSTS_FILE>) {
  chomp $line;
  ($ip, $fqdn) = split(/ +/, $line);
  $hostbyip {$ip} = $fqdn;
}
close(HOSTS_FILE);
while (<>) {
  s/\b(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\b/$hostbyip{$1}/g;
  print;
}
```

Pri vsaki vrstici datoteke `/etc/hosts` najprej s perlovo funkcijo chomp odrežemo znak za konec vrstice, nato pa jo s split razcepimo pri presledku (ali presledkih). Tako dobimo dva kosa, številski naslov IP in pripadajoče ime, ki ju nato vpišemo v tabelo hostbyip.

Za iskanje in zamenjevanje številskih naslovov z imeni uporabljamo operator s/*vzorec*$_1$/*vzorec*$_2$/*zastavice*, ki zamenja pojavitve vzorca 1 z vzorcem 2; zastavica `g` zahteva zamenjavo vseh pojavitev, ne pa le prve. Pri regularnem izrazu, ki naj bi odkrival številske naslove IP, upoštevajmo naslednje: pred piko je treba postaviti poševnico \, ker drugače pika sama po sebi deluje kot metaznak, ki se ujame s poljubnim znakom pregledovanega niza; metaznak `\d` se ujame s katero koli številko; metaznak `\b` se ujame z nizom dolžine 0, vendar le na robovih besed. Tisti del regularnega izraza, ki se bo ujel s številskim naslovom, postavimo v oklepaje; tako se bomo lahko na ta del vhodnega niza sklicevali še iz zamenjavnega vzorca (s spremenljivko $1). Zamenjavni vzorec $hostbyip{$1} se torej razširi v niz, ki v razpršeni tabeli hostbyip pripada ključu $1; ker je slednji ravno številski naslov IP, se bo zamenjavni vzorec razširil v pripadajoče ime računalnika.

REŠITVE NALOG DRUGEGA TEKMOVANJA IZ UNIXA

R2000.U.1

Rešitev v `awk`: N: 685

```
{
  for (i = NF; i > 0; i--)
    printf "%s ", $i;
  print "";
}
```

`awk` si misli, da je vhod sestavljen iz zaporedja zapisov (*records*), ločenih z ločilnim znakom; če mu ne povemo drugače, je to znak za konec vrstice, torej so zapisi kar vrstice besedila. Skripta je sestavljena iz pravil (*rules*), vsako pravilo pa iz vzorca (*pattern*) in dejanja (*action*), ki naj se izvede, če trenutni zapis ustreza vzorcu. Naša zgornja skripta ima le eno pravilo, ki pa nima vzorca, zato se izvede na vsakem zapisu. `awk` razdeli vsak zapis na polja (*fields*); če mu ne povemo drugače, razdeli pri presledkih, torej je vsako polje pravzaprav ena beseda. Pri vsakem zapisu postavi vrednost spremenljivke NF na število polj; do posameznih polj lahko pridemo z izrazi $1, $2, ..., $NF. Stavek printf izpiše dane vrednosti v skladu z danim opisom formata, podobno kot funkcija printf v C/C++. Stavek print pa izpiše svoje argumente, ločene s presledki, na koncu pa izpiše še znak za konec vrstice. Naša „akcija“ gre torej z zanko **for** po vseh besedah v obratnem vrstnem redu in vsako izpiše, med njimi presledke, nazadnje pa še znak za konec vrstice. Ker jo `awk` izvede pri vsaki vrstici posebej, je rezultat ravno to, kar je naloga zahtevala.

Če je vrstica ena sama, gre tudi z ukazom v lupini:

```
tr " " "\n" | tac | tr "\n" " "
```

Program `tr` bere podatke s standardnega vhoda, spreminja nekatere znake in spremenjeno besedilo izpisuje na standardni vhod. Niza, ki ju dobi kot parametra, mu povesta, kako naj spreminja znake (vse pojavitve i-tega znaka prvega niza zamenja z i-tim znakom drugega niza).

Program `tac` prebere vsebino dane datoteke (oz. standardnega vhoda, če mu ne podamo imena datoteke) in si misli, da je sestavljena iz „zapisov“, ki se končajo z določenim ločilnim nizom. Te zapise potem izpiše na standardni izhod v obratnem vrstnem redu. Privzeta vrednost ločilnega niza je `"\n"`, torej `tac` takrat preprosto obrne vrstni red vrstic v vhodnem besedilu.

Naša gornja rešitev torej najprej zamenja presledke z znaki za konec vrstice, tako da vsaka beseda pride na samostojno vrstico; nato s `tac` zamenja vrstni red vrstic; na koncu pa znake za konec vrstice spremeni v presledke, tako da iz vsega skupaj spet nastane ena sama vrstice, med besedami pa so presledki. V primeru, če ima vhodno besedilo več vrstic, ta rešitev seveda ne deluje, saj bi besede z vseh vrstic staknil skupaj v eno samo dolgo vrstico. Malo nerodno je tudi to, da na koncu svojega izpisa ta rešitev ne doda znaka za konec vrstice, pač pa le še en presledek (saj je zadnji klic `tr` vse konce vrstic spremenil v presledke).

N: 686

R2000.U.2

Datoteke, ki nas zanimajo, lahko poiščemo s programom `find`. Naročili mu bomo, naj išče vse od korenskega imenika (`/`) navzdol, da nas zanimajo le navadne datoteke (`-type f`) in da mora v imenu biti besedica `core` (`-name "*core*"`). Program `find` izpiše poti do najdenih datotek na standardni izhod, po eno datoteko v vsaki vrstici. To beremo v zanki z lupininim ukazom `read`; v vsaki iteraciji te zanke preberemo po eno vrstico in jo shranimo v spremenljivko `$f`. Ker ima marsikakšna datoteka v imenu besedo `core`, pa vendarle ne gre za core dump, moramo pred brisanjem še preveriti, če je datoteka res tega tipa. To lahko naredimo s programom `file`, ki izpiše na standardni vhod vrstico oblike „*ime datoteke*`:` *opis*“. Pobrisali bomo le tiste, pri katerih tudi *opis* vsebuje besedo `core`. To preverimo s programom `grep` in primernim regularnim izrazom; stikalo `-q` pa mu naroči, naj ničesar ne izpisuje, saj potrebujemo od njega le povratno vrednost, da jo bomo lahko preverili z lupininim stavkom **if**.

```
find / -type f -name "core*" |
while read f
do
  if file $f | grep -q ^[^:]*:.*core
  then
    rm -f $f
```

```
  fi
done
```

Lahko bi uporabili tudi program `cut`, mu naročili, naj razreže vrstico, ki jo je izpisal `file`, pri dvopičjih (`-d :`) in izpiše vse od vključno drugega kosa naprej (`-f 2-`), kajti prvi kos je ime datoteke.

```
  if file $f | cut -d : -f 2- | grep -q core
```

R2000.U.3

Naloga je zelo primerna za reševanje s programom `sed`. N: 686
Ta bere standardni vhod vrstico za vrstico in izvaja ukaze, ki smo mu jih navedli. Z njimi lahko trenutno vrstico preoblikujemo, na koncu pa `sed` izpiše dobljeni niz in se loti naslednje vrstice. Uporabili bomo ukaz `s:`*vzorec*$_1$`:`*vzorec*$_2$`:`*zastavice*, s katerim zamenjamo pojavitve vzorca 1 z vzorcem 2. Z zastavico `g` zahtevamo, naj se zamenjajo vse pojavitve, ne pa samo prva. Mi bi radi prazne elemente pobrisali, torej bo vzorec 2 kar prazen niz, vzorec 1 pa se mora ujemati z nizi oblike `<ime></ime>`. To, da bo `ime` obakrat enako, lahko zagotovimo tako, da njegovo prvo pojavitev v regularnem izrazu obdamo z oklepaji `\(...\)`, nato pa se nanjo sklicujemo z `\1`. Niz `\1` v regularnem izrazu namreč zahteva, naj se na tem mestu pojavi natančno isti podniz, kakršen se je že ujel s tistim delom regularnega izraza, ki je znotraj prvega para oklepajev `\(...\)`.

```
sed "s:<\([a-zA-Z]*\)></\1>::g"
```

R2000.U.4

Do vseh permutacij n besed lahko pridemo z naslednjim N: 686
razmislekom. Naj bo w prva izmed teh besed; vse permutacije naših n besed lahko razdelimo v n skupin glede na to, katero mesto ima v permutaciji beseda w. Če vzamemo eno od teh skupin in zbrišemo besedo w iz vseh permutacij v njej, dobimo ravno vse permutacije preostalih $n-1$ besed, vsako natanko po enkrat. Torej lahko z rekurzivnim klicem pripravimo najprej vse permutacije $n-1$ besed, nato pa besedo w vrinemo v vsako od njih na vseh n možnih položajev (kot prvo, drugo, ..., n-to). Tako bomo dobili ravno vse permutacije vseh n besed. Robni primer, pri katerem se rekurzija ustavi, nastopi takrat, ko imamo le še eno samo besedo in je pri njej možna tudi ena sama permutacija. Pravzaprav bi lahko definirali tudi permutacijo 0 besed (kot prazen seznam) in ustavili rekurzijo šele tam.

Rešitev v perlu:

```
#!/usr/bin/perl -n
permutiraj([split], []);
sub permutiraj {
  my @elementi = @{ $_[0] };
  my @permutacije = @{ $_[1] };
```

```
  unless (@elementi) {
    print "@permutacije\n";
  } else {
    my(@noviElementi, @novePermutacije, $i);
    foreach $i (0 .. $#elementi) {
      @noviElementi = @elementi;
      @novePermutacije = @permutacije;
      unshift(@novePermutacije, splice(@noviElementi, $i, 1));
      permutiraj([@noviElementi], [@novePermutacije]);
    }
  }
}
```

Rešitev v pythonu:

```
def permut1(s):
  if s == []: return [s]
  else: return [u[:i] + [s[0]] + u[i:] for u in permut1(s[1:]) for i in range(len(s))]

import sys
for s in permut1(sys.stdin.readline().split()):
  print " ".join(s)
```

Funkcija `permut1(s)` vrne vse permutacije seznama `s`; torej, če je `s` seznam n elementov, vrne `permut1(s)` seznam $n!$ seznamov, od katerih ima vsak po n elementov.

Še nekaj pojasnil o pythonovi sintaksi in uporabljenih funkcijah. Funkcija `range(n)` vrne seznam [0, 1, 2, ..., n − 1]. Izraz `[f(x, y) for x in L1 for y in L2]` sestavi seznam, v katerem je po en element, namreč `f(x, y)`, za vsak par `(x, y)`, pri katerem je `x` iz seznama `L1` in `y` iz seznama `L2`. Izraz `u[:i]` pomeni seznam, v katerem je prvih `i` elementov seznama `u`; izraz `u[i:]` pa seznam, v katerem so vsi elementi seznama `u` razen prvih `i`. Izraz `[]` pomeni prazen seznam, izraz `[x]` pa seznam, ki ima le en sam element, namreč `x`. Sezname lahko stikamo z operatorjem +. Objekt `sys.stdin` predstavlja standardni vhod; metoda `readline` prebere naslednjo vrstico in jo vrne kot niz; metoda `x.split` pa vrne seznam nizov, ki jih dobi tako, da niz `x` razreže pri vseh presledkih. S klicem `x.join(s)` pa dobimo niz, v katerem so staknjeni skupaj vsi nizi iz seznama `s`, med njimi pa je niz `x` (v našem primeru presledek).

Slabost gornje rešitve je, da eksplicitno sestavi seznam vseh $n!$ permutacij danega seznama. To utegne biti potratno s pomnilnikom, klicatelj funkcije `permut1` pa mogoče sploh ne potrebuje seznama vseh permutacij — mogoče mu je dovolj že to, da jih dobiva eno za drugo in lahko z vsako nekaj naredi. V našem primeru je že tako, saj jih moramo le izpisovati in lahko na vsako pozabimo, čim jo izpišemo.

Zato bi bilo lepo, če bi lahko funkcijo `permut1` izvajali „po koščkih“ — vsakič le toliko, da bi nam izračunala naslednjo permutacijo; nato bi njeno izvajanje

zamrznili, obdelali trenutno permutacijo, nato z izvajanjem funkcije permut1 nadaljevali do naslednje permutacije in tako dalje. Podprogramu, ki ga lahko uporabljamo na ta način, pogosto pravimo „korutina“ (*coroutine*). Korutine podpira tudi python, le da se tam imenujejo „generatorji“.

```
def permut2(s):
  if s == []: yield s
  else:
    for u in permut2(s[1:]):
      for i in range(len(s)):
        yield u[:i] + [s[0]] + u[i:]
import sys
for s in permut2(sys.stdin.readline().split()):
  print " ".join(s)
```

Stavek **yield** namesto **return** pythonu pove, da to ni navaden podprogram, pač pa generatorska funkcija. Ko pokličemo permut2(s), se ne začnejo izvajati stavki v funkciji permut2, pač pa je rezultat tega klica *generator* — nek objekt, ki hrani podatke o tem, do kod je že prišlo izvajanje podprograma permut2 in kakšne so trenutne vrednosti njegovih lokalnih spremenljivk. Generator pa ima tudi metodo next, ki ob vsakem klicu nadaljuje z izvajanjem podprograma permut2 do naslednjega stavka **yield** in vrne vrednost, ki jo je permut2 navedel v tem stavku **yield**. Zato lahko generator uporabimo v stavku **for** in bo na primer **for** u **in** permut2(...) v vsaki iteraciji zanke priredil u-ju naslednjo vrednost, ki jo vrne permut2(...) prek stavka **yield**. Pri našem programu se tako zdaj pravzaprav vedno izvaja kar n vzporednih korutin, ki prek stavka **yield** sestavijo naslednjo permutacijo, nikoli pa ne obstaja v pomnilniku hkrati seznam vseh permutacij. Poraba pomnilnika je zato le še $O(n)$, ne več $O(n!)$.

Še en način, kako priti do vseh permutacij, pa je naslednji: vse permutacije n elementov dobimo tako, da na vse možne načine izberemo enega od njih in ga postavimo na prvo mesto, nato pa ostala mesta zapolnimo z vsemi permutacijami ostalih $n-1$ elementov. To je pravzaprav enak razmislek kot prej, le zapisan na malo drugačen način; je pa dobro izhodišče za naslednjo rekurzivno rešitev.

```
def permut3(zePostavljene, ostale):
  if ostale == []:
    print " ".join(zePostavljene)
  else:
    for i in range(len(ostale)):
      permut3(zePostavljene + [ostale[i]], ostale[:i] + ostale[i + 1:])
import sys
permut3([], sys.stdin.readline().split())
```

Ta rešitev pa porabi veliko časa za rezanje in stikanje seznamov. Zato bo bolje, če imamo ves čas le en sam seznam in elemente samo premeščamo po njem:

```
def permut4(tabela, stZePostavljenih):
  if stZePostavljenih == len(tabela):
    print " ".join(tabela)
  else:
    for i in range(stZePostavljenih, len(tabela)):
      (tabela[i], tabela[stZePostavljenih]) = (tabela[stZePostavljenih], tabela[i])
      permut4(tabela, Postavljenih + 1)
      (tabela[i], tabela[stZePostavljenih]) = (tabela[stZePostavljenih], tabela[i])
import sys
permut4(sys.stdin.readline().split(), 0)
```

Tukaj torej podprogram permut4 predpostavi, da je prvih stZePostavljenih elementov tabele tabela že fiksiranih na svojih mestih, zdaj pa je treba na vse možne načine premešati preostale elemente. To naredimo tako, da z zanko izberemo vsakič po enega od preostalih, ga postavimo na indeks stZePostavljenih in ga razglasimo za fiksiranega; z rekurzivnim klicem potem pripravimo vse permutacije preostalih elementov. V pythonu lahko dve vrednosti zamenjamo s prireditvijo oblike (a, b) = (b, a), ki „hkrati“ priredi staro vrednost b-ja a-ju in staro vrednost a-ja b-ju; na ta način povlečemo enega od elementov na prvo mesto in ga po vrnitvi iz rekurzivnega klica postavimo nazaj.

Za primerjavo smo pognali na istem računalniku vse štiri predstavljene pythonovske rešitve na seznamu desetih besed (izpis pa preusmerili v datoteko). permut1 je porabil 64 s, permut2 31 s, permut3 61 s, permut4 pa 45 s. Videti je torej, da sta permut1 in permut3 počasna zaradi preveč prekladanja seznamov; razlika v hitrosti med permut2 in permut4 pa je mogoče posledica tega, da je hitreje nadaljevati z izvajanjem generatorja kot pa začeti s povsem novim rekurzivnim klicem (ker je pri slednjem več knjigovodskega dela).

REŠITVE NALOG TRETJEGA TEKMOVANJA IZ UNIXA

N: 687

R2001.U.1

Pomagali si bomo s programom `diff`. Ta primerja dve datoteki in na standardni izhod izpiše podatke o tem, kje (v katerih vrsticah) in kako se razlikujeta. Če sta datoteki enaki, ne izpiše ničesar. Njegov izpis lahko pošljemo programu `wc`, ki zna šteti vrstice, besede in znake v svojem standardnem vhodu; s stikalom `-c` mu povemo, naj izpiše le število znakov. Spodnja skripta za lupino `bash` lahko potem to število prebere; če je enako 0, pomeni, da `diff` ni izpisal ničesar in sta datoteki enaki, sicer pa sta različni.

```
#!/bin/bash
diff $1 $2 | wc -c | (
  read dolzina;
  if [ $dolzina -gt 0 ]
  then echo "različni"; fi
)
```

Spremenljivki $1 in $2 predstavljata prva dva parametra, ki ju je naš program dobil iz ukazne vrstice. Z internim lupininim ukazom read lahko preberemo število, ki ga je izpisal wc. V stavku **if** uporabimo operator -gt, ki gleda na operanda kot na števili in pove, če je levo večje od desnega. Klic read in stavek **if** morata biti skupaj v oklepajih, sicer stavek **if** ne bi videl vrednosti, ki jo je read vpisal v spremenljivko dolzina. Lahko pa bi namesto tega uporabili obrnjene narekovaje (*backquotes*), ki izvedejo ukaze med narekovaji in izhod teh ukazov shranijo v spremenljivko:

```
#!/bin/bash
dolzina=`diff $1 $2 | wc -c`
if [ $dolzina -gt 0 ]
then echo "različni"; fi
```

R2001.U.2

Spodnji program v pythonu bere vhodno datoteko po vrsticah; če naleti na prazno vrstico, neha; če pa naleti na vrstico `Subject:`, preveri, če se za tem nizom (in morebitnimi presledki) pojavi besedilo „`I LOVE YOU`“. N: 687

```
import sys
for s in file(sys.argv[1], "rt"):
  if s == "\n": break
  if s.startswith("Subject:") and s[8:].strip().startswith("I LOVE YOU"):
    print 1; break
```

R2001.U.3

V okoljski spremenljivki $PATH so našteti imeniki, ločeni z dvopičji; s pythonovo funkcijo split lahko razbijemo ta niz v seznam imen posameznih imenikov. Potem se lotimo vsakega imenika posebej; uporabimo funkcijo realpath, ki zamenja imena simbolnih povezav s tistim, na kar te povezave kažejo. Za lažje preverjanje, če smo si nek imenik že ogledali, bomo imena že obdelanih imenikov hranili v razpršeni tabeli pregledaniImeniki. Pri vsakem imeniku potem pregledamo vse datoteke v njej (funkcija os.listdir vrne seznam imen datotek); z realpath spet poskrbimo za simbolne povezave. Potem moramo le še preveriti, če je tista stvar v resnici navadna datoteka (isfile) in če je z našega stališča izvršljiva. Pomagali si bomo s funkcijo os.stat, ki vrne strukturo s koristnimi podatki o datoteki. V polju st_mode so zastavice, ki povedo, kdo lahko datoteko bere, piše in izvaja (prav N: 687

iste, kot jih lahko spreminjamo s programom `chmod`); v poljih st_uid in st_gid pa sta uporabniška številka lastnika datoteke ter številka skupine, ki ji lastnik pripada. To dvoje lahko primerjamo s svojo številko in številko skupine (getuid, getgid) in tako vidimo, katere zastavice v st_mode veljajo za nas. Da ne bi iste datoteke šteli po večkrat, hranimo imena že odkritih datotek v razpršeni tabeli datoteke.

```
import os, os.path, stat

imeniki = os.environ["PATH"]
pregledaniImeniki = {}    # da ne bi po večkrat pregledovali celih imenikov
datoteke = {}             # množica vseh že odkritih izvršljivih datotek
# Naša uporabniška številka in skupina — to bomo uporabljali
# za preverjanje, če bi lahko neko datoteko izvedli.
uid = os.getuid(); gid = os.getgid()

# Preglejmo vse imenike.
for s in imeniki.split(':'):
  imenik = os.path.realpath(s)                  # prava pot do tega imenika
  if imenik in pregledaniImeniki: continue      # tega smo že pregledali
  pregledaniImeniki[imenik] = 1
  if not os.path.isdir(imenik): continue        # to sploh ni imenik

  # Preglejmo vse datoteke v tem imeniku.
  for ime in os.listdir(imenik):
    polnoIme = os.path.realpath(os.path.join(imenik, ime))
    if not os.path.isfile(polnoIme): continue   # najbrž je podimenik
    st = os.stat(polnoIme)
    if polnoIme in datoteke: continue           # to datoteko smo že videli
    # Preverimo zdaj, če smemo to datoteko izvajati.
    izvrsljiva = False
    if st.st_mode & stat.S_IXUSR and st.st_uid == uid: izvrsljiva = True
    if st.st_mode & stat.S_IXGRP and st.st_gid == gid: izvrsljiva = True
    if st.st_mode & stat.S_IXOTH: izvrsljiva = True
    if izvrsljiva: datoteke[polnoIme] = 1

print len(datoteke)
```

V primeru, če kaže na isto datoteko več trdih povezav (ne pa simbolnih), bi gornji program štel vsako povezavo posebej, saj bi realpath pustil imena takih povezav pri miru. Če bi se hoteli izogniti tudi takemu podvajanju, bi lahko v tabeli datoteke namesto imen hranili pare (st.st_dev, st.st_ino), ki enolično identificirajo posamezno datoteko. Pri tem je st_ino številka datoteke znotraj datotečnega sistema (*inode number*), st.st_dev pa pove, v katerem datotečnem sistemu se ta datoteka nahaja.

R2001.U.4

N: 687

Recimo, da imamo dve majhni števili (x_1, x_2) in dve veliki števili (X_1, X_2). Je bolje vzeti zmnožek obeh majhnih in zmnožek obeh velikih ali dva mešana zmnožka s po enim majhnim in enim velikim? Recimo, da je $x_1 = x_2 = x$ in $X_1 = X_2 = kx$; potem nam da prva možnost vsoto $x_1x_2 + X_1X_2 = (k^2+1)x^2$, druga pa $x_1X_1 + x_2X_2 = 2kx^2$. Ker je (če k ni premajhen) $k^2 + 1$ precej večje od $2k$, nam bo dala manjši rezultat druga možnost.

Opažanje iz tega primera lahko posplošimo: če hočemo čim manjšo vsoto zmnožkov, je bolje množiti velika števila z majhnimi kot pa posebej velika med sabo in majhna med sabo. Tega načela se bomo najdosledneje držali, če števila kar uredimo naraščajoče in nato zmnožimo najmanjše in največje, pa drugo najmanjše in drugo največje in tako naprej.

Prepričajmo se, da s tem res dobimo najmanjšo vsoto zmnožkov. Označimo naša števila v naraščajočem vrstnem redu z $a_1, \ldots, a_{2n}$, torej tako, da je $a_1 \le a_2 \le \cdots \le a_{2n}$. Naš postopek bi vzel zmnožke

$$a_1a_{2n} + a_2a_{2n-1} + \ldots + a_ia_{2n-i+1} + \ldots + a_na_{n+1}.$$

Recimo pa, da je mogoče z neko drugo razdelitvijo teh števil na pare dobiti manjšo vsoto zmnožkov. Ta razdelitev se z našo mogoče v prvih nekaj parih ujema, prej ali slej pa se mora od nje razlikovati; recimo, da so pri tej drugi razdelitvi tudi prisotni pari $a_1a_{2n}, \ldots, a_{i-1}a_{2n-i+2}$, število a_i pa ni v paru z a_{2n-i+1} (kot pri naši razporeditvi), pač pa z nekim a_j. Ker smo števila $a_1, \ldots, a_{i-1}$ in $a_{2n-i+2}, \ldots, a_{2n}$ že porabili, poleg tega pa tudi ne more biti $j = 2n - i + 1$ (saj bi potem to ne bilo nič drugače kot pri naši razporeditvi), mora biti $i < j < 2n - i + 1$, poleg tega pa mora biti število a_{2n-i+1} pri tej drugi razporeditvi v paru z nekim a_k, ne pa z a_i kot pri naši; in za k mora iz enakih razlogov kot za j veljati $i < k < 2n - i + 1$. V opazovani razporeditvi torej nastopata para a_ia_j in $a_{2n-i+1}a_k$; pa recimo zdaj, da bi elementa a_j in a_k zamenjali. S tem bi vsota zmnožkov izgubila člena a_ia_j in $a_{2n-i+1}a_k$, pridobila pa bi a_ia_{2n-i+1} in a_ja_k. Zato se poveča za

$$a_ia_{2n-i+1} + a_ja_k - a_ia_j - a_{2n-i+1}a_k = (a_{2n-i+1} - a_j)(a_i - a_k).$$

Zaradi $j < 2n-1+1$ je $a_j \le a_{2n-1+1}$, tako da je prvi faktor v tem izrazu nenegativen; zaradi $i < k$ pa je $a_i < a_k$, tako da je drugi faktor nepozitiven; celotna sprememba vsote je torej nepozitivna. Z drugimi besedami, tisto domnevno boljšo razporeditev, ki se je z našo ujemala le v prvih $i - 1$ parih, v i-tem paru pa ne, se je dalo predelati tako, da se ujema z našo tudi v i-tem paru, pri tem pa se ji ni vsota zmnožkov nič povečala, ampak je ostala ali enaka ali pa se je celo zmanjšala! S takšnim razmislekom bi lahko nadaljevali in korak za korakom spreminjali tisto razporeditev tako, da bi na koncu postala enaka naši; in ker se ji ni vsota zmnožkov pri tem nikoli povečala, pomeni, da ni

naša razporeditev nič slabša od tiste prvotne. Torej je naša razporeditev res najboljša možna.

Zapišimo še program v pythonu:

```
stevila = [int(vrstica) for vrstica in file("stevila.txt")]
stevila.sort()
f = file("pari.txt", "wt")
for i in range(len(stevila) // 2):
  f.write("%d %d\n" % (stevila[i], stevila[-i - 1]))
```

REŠITVE NALOG ČETRTEGA TEKMOVANJA IZ UNIXA

N: 688

R2002.U.1

Interval števil predstavimo z urejenim parom (od, do); spodnji program ima funkcijo interval, da predela niz znakov v takšen urejen par. Pri tem moramo niz `"*"` obravnavati posebej in vrniti interval od 0 do 255. Sicer pa dani niz s funkcijo split razcepimo pri znaku `-` in vsak kos predelajmo v celo število; rezultat je seznam L z enim ali dvema elementoma, odvisno od tega, ali je prvotni niz vseboval znak `-` ali ne. V vsakem primeru nam torej prvi element (L[0]) pove spodnjo mejo, zadnji element (L[−1]) pa zgornjo mejo intervala.

Niz oblike `0-255.*.255.*` bomo s funkcijo split razrezali pri vseh pikah in vsak kos pretvorili v urejen par, kot je to opisano v prejšnjem odstavku. V spodnjem programu to naredi funkcija intervali. (Izraz [f(x) **for** x **in** L] sestavi seznam vrednosti f(x) za vse x iz seznama L.)

Funkcija presek izračuna, koliko števil vsebuje presek dveh intervalov. Presek intervala od a_1 do a_2 in intervala od b_1 do b_2 je interval od $c_1 := \max\{a_1, b_1\}$ do $c_2 := \min\{a_2, b_2\}$, ki vsebuje $c_2 - c_1 + 1$ elementov. Če pa je presek prazen, bo ta vrednost negativna (ker bo $c_2 < c_1$) in moramo vrniti 0.

Produkt več števil lahko elegantno računamo s funkcijo, kot je produkt v spodnjem programu. Pomagamo si s pythonovo vgrajeno funkcijo reduce(f, L, a), ki vrne vrednost $\mathsf{f}(\cdots\mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{L}[0]), \mathsf{L}[1])\cdots, \mathsf{L}[\mathsf{len}(\mathsf{L}) - 1])$. Če torej hočemo produkt elementov seznama L, mora biti f funkcija, ki sprejme dva argumenta in vrne njun zmnožek; ravno takšno funkcijo pa sestavi izraz **lambda** x, y: x * y.

Glavni del programa pretvori oba dana niza v seznama intervalov; zdaj moramo za vsak par istoležnih intervalov izračunati, koliko števil je v njunem preseku. To lahko elegantno naredimo s pythonovo funkcijo map(f, L1, L2), ki vrne seznam vrednosti f(L1[i], L2[i]) za vse i od 0 do dolžine seznamov − 1.

Število naslovov, ki so skupni obema danima podomrežjema, je kar produkt velikosti presekov po posameznih komponentah. Če je ta produkt enak 0 ali 1, je to že tudi kar vrednost, ki jo mora vrniti naš program; če pa je produkt večji ali enak 2, vrnemo 2.

```
def interval(s):
  if s == "*": return (0, 255)
  L = [int(t) for t in s.split('-')]
  return (L[0], L[-1])
def intervali(s): return [interval(t) for t in s.split('.')]
def presek(a, b): return max(0, min(a[1], b[1]) - max(a[0], b[0]) + 1)
def produkt(faktorji): return reduce(lambda x, y: x * y, faktorji, 1)

import sys
preseki = map(presek, intervali(sys.argv[1]), intervali(sys.argv[2]))
sys.exit(min(2, produkt(preseki)))
```

R2002.U.2 Primer rešitve z bashem in perlom:

N: 689

```
#!/bin/bash

uporaba() {
  echo "Uporaba: $0 datoteka" 1>&2
  echo " Program na mestu izreže iz podane datoteke vse znake CR" 1>&2
  echo " (predstavljeni desetiško kot 15, kot ^M ali \r)." 1>&2
}

if [ "$#" != 1 ]; then
  uporaba
  exit 1
fi

perl -pi -e "s/\r//g" "$1"
```

Skripta v lupini `bash` vidi prvi parameter ukazne vrstice kot spremenljivko `$1`, število parametrov pa kot `$#`. Ko se prepričamo, da smo dobili točno en parameter, pokličemo `perl`, da res pobriše znake CR iz dane datoteke. Pri tem mu naročimo, naj dani program ponavlja v zanki, po enkrat za vsako vrstico vhodne datoteke, in izpisuje spremenjene vrstice (stikalo `-p`); na koncu naj dobljene izhodne podatke napiše kar čez vhodno datoteko (stikalo `-i`). Naš „program“ v perlu je tu dolg eno samo vrstico in ga podamo kar prek stikala `-e`. Stavek `s/.../.../g` zamenja vse pojavitve prvega vzorca z drugim; v našem primeru torej zamenja vse pojavitve znaka CR s praznim nizom in jih tako pobriše.

Lahko pa uporabimo tudi program `tr` in mu s stikalom `-d` naročimo, naj pobriše vse pojavitve določenih znakov (v našem primeru znaka CR). Ker pa dela `tr` le s standardnim vhodom in izhodom, moramo sami poskrbeti za pomožno datoteko. Da ne bi več uporabnikov ali procesov hkrati uporabljalo iste pomožne datoteke, dodajmo v njeno ime tudi številko trenutnega procesa, ki jo v `bash`u dobimo v spremenljivki `$$`. Na koncu s programom `mv` zapišemo pomožno datoteko čez prvotno.

```
tr -d '\r' < "$1" > "/tmp/$1-$$"
mv "/tmp/$1-$$" "$1"
```

N: 689

R2002.U.3

Podatke o procesih dobimo od programa `ps`; hočemo podatke o *vseh* procesih (stikali `a` in `x`), brez imen stolpcev v prvi vrstici (stikalo `h`); obliko izpisa mu določimo sami (stikalo `o`), in sicer hočemo za vsak proces njegovo številko (`pid`) in številko njegovega očeta (`ppid`).

Naš program bo bral, kar je `ps` izpisal; v vsaki vrstici imamo podatke o enem procesu. V razpršeni tabeli otroci bomo za vsak proces vzdrževali seznam njegovih otrok. Procesi tvorijo drevo, čigar koren je proces `init` s številko 1 (torej prav takšno drevo, kot ga izpiše program `pstree` in je prikazano pri besedilu naloge na str. 689). Globino lahko računamo rekurzivno: globina drevesa je za eno večja kot globina najglobljega izmed njegovih poddreves. Na koncu vrnemo globina(1), torej globino celotnega drevesa procesov.

```
#!/usr/bin/python
import sys, os

otroci = {}

def globina(proces):
  if not proces in otroci: return 1    # nima otrok
  return 1 + max([globina(otrok) for otrok in otroci[proces]])

for vrstica in os.popen("ps axho pid,ppid", "r"):
  s = vrstica.split()
  proces = s[0]; oce = s[1]
  if not oce in otroci: otroci[oce] = []
  otroci[oce].append(proces)
sys.exit(globina(1))
```

N: 689

R2002.U.4

Pomagali si bomo s pogojnimi stavki v lupini `bash`. V spremenljivkah `$1` in `$2` dobimo prva dva parametra iz ukazne vrstice, v `$#` pa število teh parametrov. V primerjalnih izrazih lahko z operatorjem `-f` preverimo, če je določen niz res ime kakšne datoteke; operator `-o` deluje kot logični ali, operator `!` pa pomeni negacijo. Z operatorjem `-ot` pa preverimo, če je neka datoteka starejša od druge.

```
#!/bin/bash
if (( $# != 2 )); then
  exit 3
fi

if [ ! -f "$1" -o ! -f "$2" ]; then
  exit 3
fi
```

```
if [ "$1" -ot "$2" ]; then
  exit 1
elif [ "$2" -ot "$1" ]; then
  exit 2
else
  exit 0
fi
```

REŠITVE NALOG PETEGA TEKMOVANJA IZ UNIXA

R2003.U.1

N: 690

Pri tej nalogi nam bo prišel prav program `egrep`, ki prebere neko datoteko in izpiše le tiste njene vrstice, ki ustrezajo določenemu regularnemu izrazu. Pognali ga bomo dvakrat, najprej zato, da bo obdržal vrstice, ki ustrezajo prvemu izrazu, nato pa bomo te vrstice še enkrat pognali skozi `egrep` in obdržali le tiste izmed njih, ki *ne* ustrezajo drugemu izrazu (stikalo `-v`). Programu `egrep` lahko s stikalom `-f` povemo, naj regularni izraz prebere iz določene datoteke.

Naša skripta za `bash` lahko do parametrov, ki jih je dobila iz ukazne vrstice, dostopa prek spremenljivk `$1`, `$2` in `$3`. To, če nek niz res predstavlja ime neke datoteke, lahko preverimo z operatorjem `-f`; vse tri pogoje združimo z operatorjem `-a`, ki pomeni logični in.

```
#!/bin/bash
if [ -f "$1" -a -f "$2" -a -f "$3" ]; then
  egrep -f "$2" "$1" | egrep -v -f "$3"
else
  echo "NAPAKA"
fi
```

R2003.U.2

N: 690

Spodnja rešitev (v perlu) prebere kar celo datoteko v pomnilnik (njeno ime je prvi parameter iz ukazne vrstice in do njega pridemo z `$ARGV[0]`) in potem z regularnim izrazom preveri, če je v njej prisotna glava. Operator `s/`*vzorec*$_1$`/`*vzorec*$_2$`/`*zastavice* zamenja pojavitev vzorca 1 z vzorcem 2; v našem primeru pokrije vzorec 1 celo glavo, vzorec 2 pa je prazen in tako se glave znebimo. Zastavica `s` na koncu pa zahteva, naj obravnava interpreter cel niz kot eno samo dolgo vrstico; to potrebujemo, saj bi se utegnili znotraj glave pojavljati tudi znaki za konec vrstice. Pozorni moramo biti tudi na naslednje: glava se konča že pri prvi pojavitvi niza `data`; znak `*` v regularnem izrazu pa načeloma poskuša pokriti čim več besedila („požrešno ujemanje“, *greedy matching*), torej bi šel tu do zadnje pojavitve niza `data` v opazovanem nizu. Če hočemo, da pokrije čim manj besedila (torej le do prve pojavitve niza `data`), moramo za zvezdico postaviti še `?`.

```
#!/usr/bin/perl

use strict;
use warnings;

open FH, $ARGV[0];
$_= join('', (<FH>));          # Preberemo celo datoteko.
close FH;

if (s/^RIFF.*?data//s) {       # Zbrišimo glavo, če je prisotna.
   open FH, '>', $ARGV[0];     # Če je bila glava prisotna, shranimo
   print FH;                   # preostanek podatkov nazaj v datoteko.
   close FH;
}
```

Naloga pravi, da če se datoteka ne začne na `RIFF`, naj program ne reže ničesar; ne pove pa, kaj storiti v primeru, če se začne na `RIFF`, vendar kasneje ne vsebuje niza `data`. Gornji program bi jo pustil pri miru; verjetno je to vendarle bolje, kot pa če bi pobrisali celo datoteko.

Morebitna slabost gornje rešitve je, da prebere celo datoteko v pomnilnik. To utegne biti nerodno, če je datoteka velika (kar ni pri multimedijskih datotekah nič neobičajnega). Za take primere bi bilo bolje, če bi datoteko brali po koščkih in vsebino, ki sledi nizu `data`, sproti prepisovali na začetek datoteke (podobno kot v rešitvi naloge 2003.U.4), na koncu pa bi datoteko skrajšali s funkcijo truncate.

N: 691

R2003.U.3

Za vsak proces obstaja navidezni imenik `/proc/`*pid*, pri čemer je *pid* številka procesa. V tem imeniku je med drugim datoteka z imenom `exe`, ki je simbolna povezava na izvršilno datoteko tistega procesa. V lupini `bash` lahko prek spremenljivke `$PPID` dobimo številko procesa-očeta. Ime prave izvršilne datoteke, kamor kaže naša simbolna povezava, lahko izvemo od programa `ls`, če zahtevamo izčrpnejši izpis (stikalo −l). Vrstico, ki jo `ls` izpiše, razbijmo pri vseh presledkih (`cut −d ' '`); izkaže se, da je ime datoteke, kamor kaže simbolna povezava, potem ravno enajsta komponenta vrstice. Ker pa lahko ime vsebuje tudi presledke, je bolje izpisati vse komponente od enajste naprej (stikalo `−f 11−`). Težava je le ta, da `cut` prereže pri vsakem presledku, v izpisu programa `ls` pa je včasih po več presledkov skupaj in bi zato `cut` tam vmes ustvaril še neko število praznih komponent; to število je nepredvidljivo, ker ne vemo, koliko presledkov je `ls` vrinil zaradi poravnavanja stolpcev pri izpisu (odvisno je npr. od tega, koliko števk je porabil za izpis dolžine datoteke). Dobro bi bilo torej spremeniti vsako zaporedje presledkov v en sam presledek. To lahko naredimo s programom `tr`, če uporabimo stikalo `−s`. Druga možnost je, da si izpis programa `ls` shranimo v neko spremenljivko in jo podamo programu `echo`: iz posameznih komponent bodo nastali posamezni argumenti programu `echo`, interpreterju lupine

pa je čisto vseeno, s koliko presledki so ločeni argumenti; `echo` bo med dvema argumentoma vedno izpisal en presledek.

```
#!/bin/bash
povezava=`ls -l /proc/$PPID/exe`
echo $povezava | cut -d ' ' -f 11-
```

ali pa

```
#!/bin/bash
ls -l /proc/$PPID/exe | tr -s ' ' | cut -d ' ' -f 11-
```

Gornja rešitev ima še majhno slabost: če se v imenu datoteke, na katero kaže opazovana simbolna povezava, kdaj pojavlja po več zaporednih presledkov, bo naš program tam izpisal en sam presledek, ker pač v `ls`-jevem izpisu nadomesti vsa zaporedja presledkov s po enim samim. Na srečo pa se v imenih datotek le redko pojavi več zaporednih presledkov.

Lahko bi si pomagali z dejstvom, da v izpisu programa `ls` pred imenom datoteke, na katero kaže simbolna povezava, stoji niz `"-> "`. S `sed`om lahko pobrišemo vse do vključno te puščice in presledka (stikalo `-n` je zato, da ne bo `sed` izpisal še prvotnega niza, kakršen je bil pred brisanjem):

```
#!/bin/bash
ls -l /proc/$PPID/exe | sed -n "s/^.*-> //p"
```

Slabost te rešitve je, da `sed` ujemanje z regularnimi izrazi preverja „požrešno“ (*greedy matching*) — zvezdica poskuša pobrati čim daljši kos niza. Če bi se torej v imenu očetovskega procesa pojavil niz `"-> "`, bi `sed` pobrisal še del tega imena, vse do zadnje pojavitve niza `"-> "`.

Še ena možnost je, da namesto `sed`a uporabimo `awk`; na začetku nastavimo njegovo spremenljivko `FS` in mu s tem naročimo, naj vrstico, ki jo je izpisal `ls`, razreže pri vseh puščicah. Vrstica tako razpade na `NF` delov (*i*-tega dobimo v spremenljivki `$i`), mi pa moramo izpisati vse razen prvega.

```
#!/bin/bash
ls -l /proc/$PPID/exe | awk '
  BEGIN { FS = "-> " }
  {
    for (i = 2; i < NF; i++)
      printf "%s-> ", $i;
    print $NF;
  }'
```

Še lažje in bolj elegantno gre na primer v perlu, saj imamo funkcijo `readlink`, ki nam pove, kam kaže simbolna povezava. Očetovo številko dobimo s funkcijo `getppid`, nize pa stikamo z operatorjem `.` (pika).

```
#!/usr/bin/perl
print readlink("/proc/" . getppid . "/exe") . "\n";
```

N: 691

R2003.U.4

Za stikanje datotek lahko uporabimo program `cat`. S programom `echo` mu pošljemo niz „`PREPIS`“ (brez znaka za konec vrstice, zato stikalo `-n`) in nato programu `cat` naročimo, naj stakne to, kar je prišlo s standardnega vhoda (`-`), z vsebino vhodne datoteke. Rezultat bi lahko zapisovali kar v izhodno datoteko, vendar pa bi to v primerih, ko sta vhodna in izhodna datoteka ena in ista, pomenilo, da bomo vhodne podatke najbrž izgubili, še preden bomo vse sploh prebrali. Zato raje uporabimo pomožno datoteko in jo potem preimenujmo v ime, ki smo ga dobili kot ime izhodne datoteke. Ime pomožne datoteke pripravimo s programom `mktemp`, ki znake `X` na koncu danega argumenta zamenja z naključnimi številkami in pazi na to, da datoteka s takšnim imenom še ne obstaja; dobljeno ime potem izpiše na svoj standardni izhod.

```
#!/bin/bash
pomozna=`mktemp "$2.XXXXXX"`
echo -n PREPIS | cat - "$1" > "$pomozna"
mv "$pomozna" "$2"
```

Slabost te rešitve je, da je včasih lahko potratna s prostorom. Če sta vhodna in izhodna datoteka različni, bi lahko pisali kar naravnost v izhodno datoteko, tako pa so tik pred klicem `mv` prisotne na disku vse tri: vhodna, pomožna (ki je dolga približno toliko kot vhodna, pravzaprav šest znakov daljša) in še stara izhodna. V primerih, ko sta vhodna in izhodna datoteka ena in ista, pa uporaba pomožne datoteke pomeni, da bosta pred klicem `mv` prisotni na disku dve kopiji vhodne (prvotna in tista pomožna z nizom „`PREPIS`“).

Varčnejša rešitev bi najprej preverila, če sta vhodna in izhodna datoteka ena in ista; če je res tako, naj odpre to datoteko za branje in pisanje obenem, nato pa prebira iz nje podatke po koščkih in se po vsakem branju pomakne po datoteki nazaj ter povozi ravnokar prebrane podatke s tistim, kar bo moralo biti na tem mestu zapisano v izhodni datoteki. Ker je vsebina izhodne datoteke v primerjavi z vsebino vhodne „zamaknjena“ za šest znakov (ker je v izhodni na začetku še niz „`PREPIS`“), nam vedno ostane šest znakov, ki smo jih že prebrali iz vhodne datoteke ter jih pri zadnjem pisanju tudi že povozili z drugimi podatki; te obdrži spodnji program v nizu `buf` in bodo prišli kot prvi na vrsto pri naslednjem pisanju v datoteko (takoj za naslednjim branjem).

```
import sys, os, os.path

def IstaDatoteka(ime1, ime2):
  if ime1 == ime2: return True
  # Mogoče pa sta to trdi povezavi na isto datoteko.
```

```
  st1 = os.stat(ime1); st2 = os.stat(ime2)
  return st1.st_dev == st2.st_dev and st1.st_ino == st2.st_ino

vhodIme = os.path.realpath(sys.argv[1])
pazi = False
if len(sys.argv) <= 2:
  # Izpisovali bomo na standardni izhod.
  izhod = sys.stdout
else:
  # Preverimo, če je izhodna datoteka ista kot vhodna.
  izhodIme = os.path.realpath(sys.argv[2])
  pazi = IstaDatoteka(vhodIme, izhodIme)
  if pazi:
    # Je — odprimo jo le enkrat, za branje in pisanje.
    # „vhod“ in „izhod“ bosta le dve referenci na isti objekt.
    vhod = file(vhodIme, "r+b"); izhod = vhod
  else:
    # Izhodna datoteka ni ista kot vhodna; odprimo izhodno za pisanje
    # (in uničimo morebitno obstoječo datoteko s tem imenom).
    izhod = file(izhodIme, "wb")
if not pazi:
  # Če sta vhodna in izhodna datoteka različni,
  # odprimo zdaj še vhodno, vendar le za branje.
  vhod = file(vhodIme, "rb")

buf = "PREPIS"
bufLen = 1024 * 1024
while len(buf) > 0:
  # Zapomimo si trenutni položaj v datoteki.
  if pazi: pos = izhod.tell()
  # Preberimo nekaj novih podatkov.
  buf = buf + vhod.read(bufLen)
  # Če je vhodna datoteka ista kot izhodna, se postavimo nazaj na položaj
  # „pos“, da bomo pri pisanju povozili pravkar prebrane podatke.
  if pazi: izhod.seek(pos)
  # Zapišimo nekaj podatkov.
  izhod.write(buf[:bufLen])
  # Če mešamo branja in pisanja nad isto datoteko, lahko pride včasih do težav,
  # npr. da vhod.read() vrne tisto, kar smo ravnokar zapisali na stari položaj,
  # namesto da bi prebral nove podatke. Rešitev je, da med pisanjem in branjem
  # pokličemo izhod.flush() ali pa vhod.seek(vhod.tell()).
  if pazi: vhod.seek(vhod.tell())
  # Obdržimo podatke, ki jih še nismo zapisali.
  buf = buf[bufLen:]
```

Za ugotavljanje, če se dani imeni nanašata na eno in isto datoteko, smo uporabili najprej funkcijo realpath, ki sledi simbolnim povezavam; nato pa, če sta

imeni tudi po tem različni, pogledamo za vsako ime par (st.st_dev, st.st_ino), ki enolično identificira posamezno datoteko (glej rešitev naloge 2001.U.3 na str. 707). Tako odkrijemo še primere, ko dobimo dve „trdi povezavi“ na isto datoteko.

REŠITVE NALOG ŠESTEGA TEKMOVANJA IZ UNIXA

N: 692

R2004.U.1

Primer rešitve v pythonu:

```
import sys
seznam = []
for s in file(sys.argv[1]):          # Prebirajmo vhodno datoteko po vrsticah.
  s = s.strip()                      # Odrežimo znak za konec vrstice.
  if not s: continue                 # Preskočimo morebitne prazne vrstice.
  seznam.append(s)                   # Dodajmo prvotno besedo.
  for i in range(len(s) - 1):        # Dodajmo besede, dobljene z zamenjavami.
    seznam.append(s[:i] + s[i + 1] + s[i] + s[i + 2:])
seznam.sort()                        # Uredimo rezultate po abecedi
for s in seznam: print s             # in jih izpišimo.
```

Kot zanimivost povejmo, da je program vsaj pri naših poskusih (z veliko vhodno datoteko, ki je vsebovala milijon in pol angleških besed) več kot polovico časa porabil za urejanje seznama rezultatov. Izpis pa lahko še malo pospešimo, če zamenjamo zadnjo vrstico s

```
print "\n".join(seznam)
```

Tako program stakne vse besede v en dolg niz, vmes postavi znake za konec vrstice in potem izpiše vse v enem kosu. Seveda pa zato porabimo malo več pomnilnika.

Majhna slabost te rešitve je, da gradi seznam vseh rezultatov v pomnilniku, kar utegne biti problematično, če je vhodna datoteka zelo dolga. V tem primeru lahko rezultate sproti izpisujemo v neko pomožno datoteko in nato uporabimo program `sort`, ki naj bi znal urejati tudi zelo velike datoteke (pri tem si pomaga z dodatnimi pomožnimi datotekami, če je to potrebno). Spodaj je rešitev z `awk`om in ukazno lupino. Pomožno datoteko ustvarimo s programom `mktemp`, ki poišče primerno ime, ki še ne obstaja. Klic `sort` na koncu pomožno datoteko uredi in izpiše, nato pa jo z `rm` še pobrišemo.

```
#!/bin/bash
TMP=`mktemp -t premetavanje.XXXXXXXXXX`
while read BESEDA; do
  echo "$BESEDA" | awk '{
    for (i = 1; i <= length($1); i++) {
```

```
        beg = substr($1, 1, i − 1);
        r1 = substr($1, i, 1);
        r2 = substr($1, i + 1, 1);
        end = substr($1, i + 2, length($1));
        print beg r2 r1 end ;
      }
  }' >> "$TMP"
done < "$1"

sort "$TMP"
rm −f "$TMP"
```

R2004.U.2

Primer rešitve v pythonu:

N: 692

```
import sys
zadnjiDostop = {}; stSej = 0
for vrstica in file(sys.argv[1]):
  vrstica = vrstica.split(); ip = vrstica[0]; cas = int(vrstica[1])
  if ip not in zadnjiDostop or zadnjiDostop[ip] < cas − 1800: stSej += 1
  zadnjiDostop[ip] = cas
print stSej
```

V razpršeni tabeli zadnjiDostop imamo za vsak naslov IP zapisan čas zadnjega dostopa s tega naslova. Če naletimo na naslov, ki ga v tabeli še ni, ali pa sicer je, vendar je njegov zadnji dostop že prestar, vemo, da se je začela nova seja.

Če so v vhodni datoteki sami različni IPji, bo naša razpršena tabela na koncu hranila praktično že celotno vsebino vhodne datoteke. Če je vhodna datoteka zelo dolga, nas torej lahko skrbi, da bo naš program porabil preveč pomnilnika. Podobno kot pri prejšnji nalogi lahko vhodno datoteko tudi tu najprej uredimo; tako pridejo vrstice, ki se nanašajo na isti IP, skupaj in so tudi urejene po naraščajočem času dostopa. Zdaj je za prepoznavanje novih sej dovolj, če primerjamo po dve zaporedni vrstici.

```
#!/bin/bash
sort −s −k 1,1 $1 | python −c 'import sys
prejsnjiIP = None; stSej = 0
for vrstica in sys.stdin:
  vrstica = vrstica.split(); ip = vrstica[0]; cas = int(vrstica[1])
  if ip != prejsnjiIP or cas > prejsnjiCas + 1800: stSej += 1
  prejsnjiIP = ip; prejsnjiCas = cas
print stSej'
```

Program `sort` lahko razbije vsako vrstico na „polja“, ločena s presledki (ali čim drugim, če mu s parametrom `−t` naročimo drugače); mi bomo s parametrom `−k 1,1` zahtevali, naj za urejanje uporabi le prvo polje, torej naslov IP. Pri vrsticah z enakim naslovom IP pa moramo ohraniti njihov dosedanji medsebojni

vrstni red, tako da bodo ostale urejene po naraščajočem času dostopa; potrebujemo torej stabilno urejanje, kar povemo s stikalom `-s`. Druga možnost je, da bi eksplicitno zahtevali tudi urejanje po drugem polju, vendar pa moramo vrednosti v njem gledati kot števila in ne kot nize; lahko bi torej rekli:

```
sort -k 1,1 -k 2,2n $1 | ...
```

N: 693

R2004.U.3

Lahko si pomagamo z ukazno lupino. Z ukazom `read` berimo vhodno datoteko `$1` po vrsticah. Trenutno vrstico dobimo v spremenljivki `$IME` in jo podamo programu programu `touch`, da ustvari prazno datoteko s tem imenom. Nato s programom `ls` izpišimo imena nastalih datotek; stikalo `-r` zahteva obrnjeni abecedni vrstni red, stikalo `-w 1` pa ga prisili, da izpiše vsako ime v svojo vrstico. Tako torej dobimo seznam nizov, urejen v obrnjenem abecednem vrstnem redu, in ga lahko shranimo v izhodno datoteko `$2`.

Vse skupaj raje počnimo v nekem pomožnem direktoriju (`$TMPDIR`), da bomo na koncu lažje počistili za sabo (`rm`). Pomožni direktorij ustvarimo s programom `mktemp`, ki sam zamenja niz `XX...X` s takšnimi znaki, da nastalo ime še ni v rabi; s stikalom `-d` zahtevamo, naj ustvari direktorij, ne pa navadne datoteke, s stikalom `-t` pa, naj se nahaja pod pomožnim direktorijem (običajno `/tmp`). Uporabljeno ime izpiše `mktemp` na svoj standardni izhod in ga lahko prestrežemo v spremenljivko `$TMPDIR`.

```
#!/bin/bash
TMPDIR=`mktemp -t -d urejanje.XXXXXXXXXX`
while read IME; do
  touch "$TMPDIR/$IME" 2> /dev/null
done < "$1"
ls "$TMPDIR" -w 1 -r > "$2"
rm -rf "$TMPDIR"
```

N: 693

R2004.U.4

Spodaj je primer rešitve v ukazni lupini. Za začetek s programom `identify` ugotovimo velikost vhodne slike. `identify` izpiše več podatkov o sliki, ločenih s presledki; na tretjem mestu je niz oblike *širina*`x`*višina*, tako da lahko do širine in višine pridemo s pomočjo `sed`a in `awk`a. Potem z lupininim vgrajenim ukazom `let` izračunajmo višino izhodne slike, da bo razmerje višine in širine enako kot pri vhodni.

Sliko bomo pretvarjali s programom `convert`; s parametrom `-resize` mu naročimo spremembo velikosti slike, s parametrom `-quality` pa lahko vplivamo na velikost izhodne datoteke. Če je izhodni format JPEG, ima lahko `-quality` vrednosti od 0 do 100. Pri manjših vrednostih bo izhodna datoteka manjša, vendar bo slika zato tudi bolj popačena. Do primerne nastavitve pridemo s poskušanjem; če je kvaliteta 100 prevelika, jo zmanjšujemo, dokler ne dobimo dovolj majhne datoteke. Da ne bo trajalo predolgo, jo zmanjšujmo v korakih

po 10, nato pa jo po potrebi še povečajmo do največje dopustne vrednosti. Tega bi se lahko lotili tudi kako drugače, npr. z bisekcijo.

Za ugotavljanje velikosti izhodne datoteke uporabimo ukaz `ls -l`, ki kot peto polje izpiše dolžino datoteke; to lahko izluščimo z `awk`om. Izraz `` `ukaz` `` se pri izvajanju skripte nadomesti z nizom, ki ga izpiše ukaz `ukaz` na svoj standardni izhod. Spomnimo se še, da se pri preverjanju pogojev obnaša operator `-a` kot logični in, operatorja `-lt` in `-gt` pa kot primerjalna operatorja $<$ in $>$. Za računanje aritmetičnih izrazov uporabljamo lupinin vgrajeni ukaz `let`.

```
#!/bin/bash

# Določimo velikost vhodne in izhodne slike.
Sirina=`identify "$1" | awk '{print $3}' | sed 's/x/ /g' | awk '{print $1}'`
Visina=`identify "$1" | awk '{print $3}' | sed 's/x/ /g' | awk '{print $2}'`
NovaSirina=$3
let NovaVisina=$Visina*$NovaSirina/$Sirina

# Začnimo z največjo možno kakovostjo.
Q=100
convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $Q $2

# Zmanjšujmo kakovost v korakih po 10, dokler ne dobimo dovolj majhne slike.
while [ $Q -gt 0  -a  `ls -l $2 | awk '{print $5}'` -gt $4 ]; do
  let Q=$Q-10
  convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $Q $2
done

# Povečujmo kakovost, dokler je slika še dovolj majhna.
while [ $Q -lt 100 ]; do
  let NovaQ=$Q+1
  convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $NovaQ $2
  if [ `ls -l $2 | awk '{print $5}'` -gt $4 ]; then
    break; fi    # NovaQ je že prevelika.
  Q=$NovaQ
done

# Pripravimo končno verzijo slike.
convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $Q $2
```

Tematsko kazalo

Stvarno kazalo

Številke strani v ležečem tisku se nanašajo na pojavitve v rešitvah nalog, številke v običajnem tisku pa na pojavitve v nalogah samih ali pa v ostalem besedilu.