25. državno tekmovanje v znanju računalništva (2001)

NALOGE ZA PRVO SKUPINO

2001.1.1 Tipkanje

R: 15

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:

```
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

R: 16

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).

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: 17 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.[1] 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
```

[1] 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: 19

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: 31

2001.2.2 Psevdo-tetris

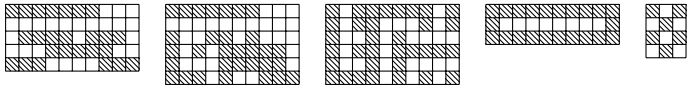
R: 34 Dana je igralna površina, ki je pravokotna karirasta mreža z Y_P vrsticami in X_P stolpci. Vsako od $X_P \times Y_P$ polj (kvadratkov) te mreže je lahko „polno“ ali pa „prazno“.

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.[2]

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;
```

[2] 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: 35

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.

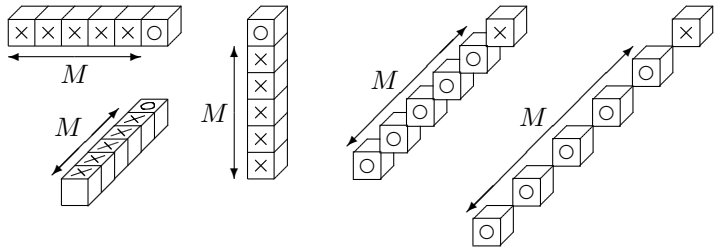
type SmerT = (Stop, Noter, Ven);
procedure Motor(IzbranaSmer: SmerT); **external**;
: S tem podprogramom lahko upravljamo elektromotor: parameter je lahko: Stop (ugasne motor), Noter (vključi motor v smer zapiranja predalčka), Ven (vključi motor v smer odpiranja predalčka).

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.

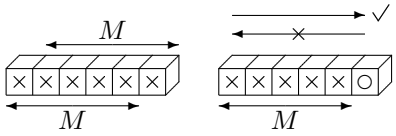
2001.2.4 3-D križci in krožci

R: 36 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: 37

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.[3]

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: 39

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 ($1 \leq N \leq 15$), tvoj program pa naj v *izhodno datoteko* izpiše vrstico, ki vsebuje število oklepajskih izrazov reda N.

[3] 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: 42 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: 45

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.

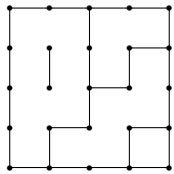
V *vhodni datoteki* sta v prvi vrstici najprej napisani dimenziji M in N (v tem vrstnem redu; velja $1 \leq M \leq 100$, $1 \leq N \leq 100$), potem pa sledi N vrstic; v vsaki vrstici je M dvomestnih števil (ločenih s presledki), ki predstavljajo stanje pregrad okoli posameznih celic. (Vrstice so podane 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. 12.

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: 51

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.[4]

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.[5] 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`

[4] 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.

[5] *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: 53

$$P(\mathbf{x}, \mathbf{y}) = \frac{x_1y_1 + x_2y_2 + \ldots + x_ny_n}{\sqrt{x_1^2 + \ldots + x_n^2}\sqrt{y_1^2 + \ldots + y_n^2}}.$$

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
```

LETO 2001, TEKMOVANJE V POZNAVANJU UNIXA

R: 55

2001.U.1

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
```

R: 55

2001.U.2

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`.

R: 56

2001.U.3

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.

R: 57

2001.U.4

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
```

REŠITVE NALOG ZA PRVO SKUPINO

R2001.1.1 Tipkanje

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). N: 1

```
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

N: 1 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.

```
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}
```

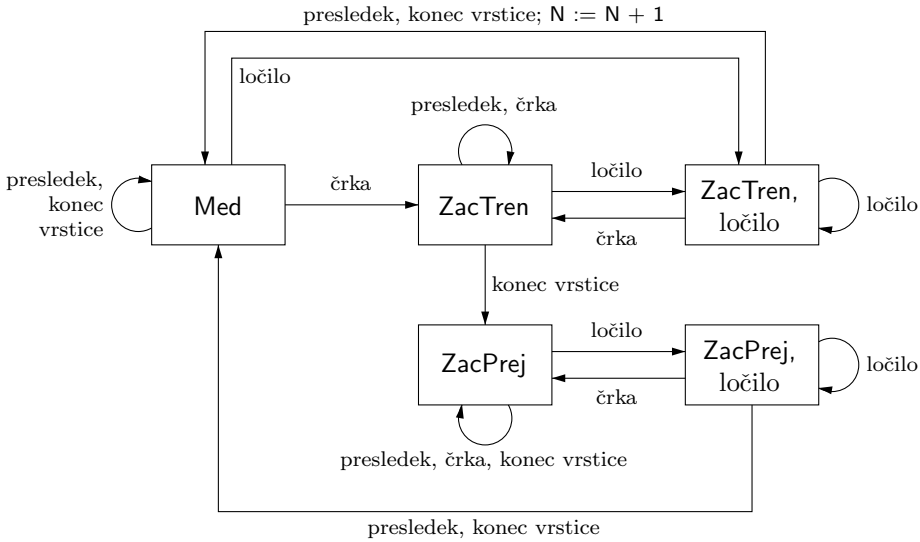
R2001.1.3 Besedilo v stolpcu

N: 2

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. 18); 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).



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).

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
```

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

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. N: 3

```
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 $Y \leq y$, 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 $X \geq x$ in $y \geq Y$; toda zdaj sta x in y enaka in bi za takšne trojice veljalo $X \geq Y$, 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.[6] (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 $\leq b$, mora biti $k^2 + l^2 \leq b$ in zato $k \leq \sqrt{b}$. Zdaj torej ni treba drugega, kot da z dvema gnezdenima zankama pregledamo vse primerne pare (k, l) do $k = \lfloor \sqrt{b} \rfloor$.

[6] Glej npr. Jože Grasselli, *Diofantske enačbe*, 1984, § 10.

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 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[7]). 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[8]

$$\triangle(z) := (2d_1 + 1)(2d_2 + 1) \cdots (2d_r + 1) - 1.$$

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 $d_1, \ldots, d_r$).

[7] Glej npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 33.2 v prvi izdaji, 31.2 v drugi.

[8] 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).

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. }
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 $z \ge q^2$, 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.[9] 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

[9] 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.

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, 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,[10] da je vsota v oklepajih približno enaka $\ln\ln P$, tako da ima celoten postopek $O(m \ln\ln m)$ računskih operacij.[11]

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 $O(m/\ln\ln m)$.)

[10] 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).

[11] 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).

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 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.[12]

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

[12] 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 $n/(2\pi)$.

---

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 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: 3

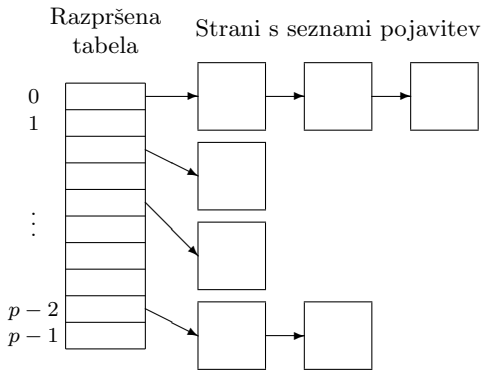
Glede na vrsto poizvedb, ki jih želimo izvajati, je še najkoristneje, če datotekam dodelimo neke preproste številske oznake in si za vsako besedo pripravimo seznam datotek, v katerih se pojavlja. Te sezname bi seveda hranili na disku in pri vsaki poizvedbi naložili le tiste, ki se tičejo iskanih besed. 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. 32). 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.[13] Če pa bi hoteli imeti le iskalnik za popolnoma statičen in vnaprej znan nabor datotek, nam niti ne bi bilo treba komplicirati

[13] 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).

z 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 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 zadosto-

val 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.[14]

R2001.2.2 Psevdo-tetris

N: 4 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).

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.

[14] 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.

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).

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

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).

N: 5

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: 6 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

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.

N: 9

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

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).

N: 9

```
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}
```

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.

Tabela na str. 42 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.[15]

R2001.3.3 Parlament

N: 10 Spodnja rešitev preprosto našteje vse koalicije in med njimi poišče najšibkejšo. 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
```

[15] Glej npr. MathWorld *s. v.* "Catalan Number" in *The On-Line Encyclopedia of Integer Sequences*, A000108.

```
        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 $(p-1) \operatorname{div} 2$.[16] 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
```

[16] 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 $\lceil p/d \rceil = \lfloor (p+d-1)/d \rfloor$.

```
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. }
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: 11

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;
```

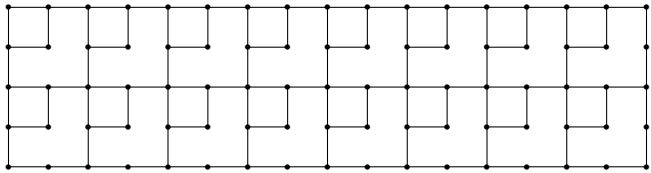
```
  { Preglejmo celo mrežo. }
  for y := 1 to N do begin
    for x := 1 to M do begin
      Read(T, Celica); Celica := Celica and (Zgoraj or Levo);
      if Celica = 0 then
        { Celica ima zidova zgoraj in levo; ustanovimo zanjo novo kletko. }
        begin StBarv := StBarv + 1; k := StBarv end
      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. 47) 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 $O(mn \min\{m, n\})$.



Primer mreže, kjer se izvaja zlivanje pri vsaki četrti celici.

Lahko pa bi uporabili znano podatkovno strukturo za disjunktne množice,[17] ki zagotavlja, da bo pri c celicah in $O(c)$ operacijah nad njimi skupna 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.

[17] 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).

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: 12

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: 13

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}
```

REŠITVE NALOG TRETJEGA TEKMOVANJA IZ UNIXA

R2001.U.1

N: 14

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

N: 14

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`“.

```
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
```

N: 14

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 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: 14

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 $a_j \leq a_{2n-1+1}$, 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]))
```

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).