27. državno tekmovanje v znanju računalništva (2003)

NALOGE ZA PRVO SKUPINO

2003.1.1 Dva kupa števil

Rešitev: str. 15

Napiši podprogram

```
procedure Razdeli(N: integer);
```

ali, v C-ju:

```
void Razdeli(int N);
```

ki bo števila $1, 2, 3, \ldots, N-1, N$ (kjer je N vhodni parameter podprograma Razdeli) razdelil na dva kupa in to tako, da bosta vsoti števil na enem in na drugem kupu čim bolj podobni; če pa je možnih glede tega več enako dobrih razporedov, poišči tistega, ki ima na obeh kupih čim bolj enako število števil.

Podprogram naj izpiše vsebino vsakega od kupov in vsoto števil na njem. Možnih je več rešitev. Dovolj je, da najdeš eno izmed njih.

Primer: Razdeli(9) lahko izpiše

```
1. kup: 2, 5, 6, 9  Vsota: 22
2. kup: 1, 3, 4, 7, 8  Vsota: 23
```

2003.1.2 „Pet čevljev merim, palcev pet,“

Rešitev: str. 18

. . . je tarnal mlad trgovec, ko je čez lužo odprl trgovino z blagom. Sprva se mu niti sanjalo ni, da mu utegne pretvarjanje enot povzročati toliko preglavic.

„Kako že gre? *1 liga* so *3 milje*, *1 milja* je *8 furlongov*, *1 furlong* je *220 jardov*, *1 jard* so *3 čevlji*, *1 čevelj* so *3 dlani*, *1 dlan* so *4 palci*,“ je drdral v mislih. Uh, saj to je še šlo, a ko je nekaj kupcev naročilo blago, se je pri seštevanju in naročanju blaga v tovarni krepko uštel.

Pomagaj mu **sestaviti program**, ki s standardnega vhoda prebere deset vrstic s po sedmimi števili (ki pomenijo zaporedoma število lig, milj, furlongov, jardov, čevljev, dlani in palcev blaga, ki so jih posamezne stranke naročile), nato pa še vrstico sedmih števil, ki povedo količino blaga, naročenega v tovarni. Na standardni izhod naj nato izpiše eno od naslednjih sporočil:

```
Naročil si *** preveč blaga.
Naročil si *** premalo blaga.
Naročil si ravno prav blaga.
```

Namesto zvezdic naj se izpiše količina preveč ali premalo naročenega blaga. Ta naj bo izražena tako, da uporabiš čim večje enote. Dolžino 5 000 jardov bi moral na primer napisati kot 2 milji, 6 furlongov in 160 jardov (ne pa kot 5 000 jardov ali pa kot 2 milji in 1 320 jardov ali pa kot 22 furlongov in 160 jardov ali pa celo kot 15 000 čevljev ali kaj podobnega). Količino zapiši kar s sedmimi števili, ločenimi s presledki, torej v enaki obliki, v kakršni so zapisane posamezne količine tudi v vhodnih podatkih. Dolžino 5 000 jardov bi tako zapisal kot `0 2 6 160 0 0 0`.

2003.1.3 Glasovanje

Rešitev: str. 19

Na šolah vsako leto v vsakem razredu učenci izvolijo predsednika razreda. Šola, na katero hodi tudi Miha, je to leto uvedla nov, bolj zaupen način volitev. Vsak učenec je na list napisal številko kandidata, za katerega je glasoval, nato pa je Miha števila iz vseh volilnih lističev pretipkal v računalnik. Sedaj pa potrebujejo tebe, da jim **sestaviš podprogram**, ki bo na podlagi teh števil izrisal histogram, iz katerega bo lepo razvidno, koliko glasov je dobil posamezen kandidat. Kandidati so oštevilčeni s številkami od 1 do StKandidatov, zagotovo pa je StKandidatov manjše ali enako MaxStKandidatov.

Tvoj podprogram naj bo takšne oblike:

```
const MaxStVolilcev = ...; MaxStKandidatov = ...;
type GlasoviT = array [1..MaxStVolilcev] of integer;

procedure Histogram(StKandidatov, StVolilcev: integer;
                    Glasovi: GlasoviT);
begin
  ...
end;
```

oziroma, v C-ju:

```
#define MaxStKandidatov ...

void Histogram(int StKandidatov, int StVolilcev, int Glasovi[ ])
{
  ...
}
```

Primer: za 7 kandidatov, 10 volilcev in glasove $\langle 1, 3, 2, 4, 1, 4, 7, 6, 1, 2 \rangle$ naj podprogram izpiše:

```
1:***
2:**
3:*
4:**
5:
6:*
7:*
```

2003.1.4 Radar

Rešitev: str. 20

Na cestnem odseku je postavljen radar, ki stalno beleži hitrosti mimovozečih vozil in jih zapisuje v datoteko, vsako meritev v svojo vrstico. Nabralo se je veliko število meritev, sedaj pa nas zanima dvajset najvišjih izmerjenih hitrosti.

Napiši program, ki prebere datoteko s podatki in izpiše dvajset največjih prebranih števil (ni nujno, da so v izpisu urejena po velikosti). Podatkov je preveč, da bi lahko vse shranili v pomnilnik.

NALOGE ZA DRUGO SKUPINO

2003.2.1 Križanka

Rešitev: str. 23

Napiši podprogram, ki dobi kot vhod križanko, na izhod pa izpiše besede, ki se pojavljajo v križanki, skupaj z njihovimi položaji.

Križanka je sestavljena iz m vrstic in n stolpcev. V vsakem polju je ali velika tiskana črka angleške abecede ali pa znak `'*'`, ki pomeni, da to polje razmejuje besede. Za besedo šteje strnjeno zaporedje vsaj dveh črk, ki je napisano navpično ali vodoravno in je na začetku in koncu ločeno od ostalih črk v križanki z znakom `'*'` ali z robom križanke.

Vsa polja, na katerih se začnejo besede, oštevilčimo od 1 naprej. Štejemo po vrsticah od zgoraj navzdol, v vsaki vrstici pa od leve proti desni (glej primer).

Tvoj podprogram naj ustreza naslednjim deklaracijam:

```
const Visina = ...; Sirina = ...;
type KrizankaT = array [1..Visina, 1..Sirina] of char;
procedure IzpisiBesede(Krizanka: KrizankaT);
```

ali pa

```
#define Visina ...
#define Sirina ...
void IzpisiBesede(char Krizanka[Visina][Sirina]);
```

Primer: recimo, da imamo podano naslednjo križanko.

```
SOLAR
*LOK*
SIPON
IMATO
RPR*V
```

Najprej oštevilčimo vsa polja (od zgoraj navzdol, v vsaki vrstici od leve proti desni):

```
1234.
*5..*
6...7
8....
9....
```

Program mora v tem primeru izpisati:

```
1, vodoravno: SOLAR
1, navpično : -
2, vodoravno: -
2, navpično : OLIMP
3, vodoravno: -
3, navpično : LOPAR
4, vodoravno: -
4, navpično : AKOT
5, vodoravno: LOK
5, navpično : -
6, vodoravno: SIPON
6, navpično : SIR
7, vodoravno: -
7, navpično : NOV
8, vodoravno: IMATO
8, navpično : -
9, vodoravno: RPR
9, navpično : -
```

2003.2.2 Števke

Rešitev: str. 24

V danem naravnem številu je *zadnja neničelna števka* najbolj desna neničelna števka v desetiškem zapisu števila. Na primer, zadnja neničelna števka števila 123 je 3, pri številu 45600 je to 6, pri številu 100 pa 1.

Napiši podprogram

```
procedure Stevke(M, N: integer; var Rezultat: array [1..9] of integer);
```

ali, v C:

```
void Stevke(int M, int N, int Rezultat[9]);
```

ki za vsako števko od 1 do 9 ugotovi, kolikokrat se pojavi kot zadnja neničelna števka v zaporedju

$$M, M+1, M+2, \ldots, N-2, N-1, N.$$

Podprogram naj zapiše rezultat v tabelo Rezultat. Na primer, če je $M = 118$ in $N = 122$, pomeni, da obravnavamo zaporedje 118, 119, 120, 121, 122. Zadnje neničelne števke so 8, 9, 2, 1 in 2. Torej mora tabela Rezultat vsebovati elemente 1, 2, 0, 0, 0, 0, 0, 1, 1, ker enica, osmica in devetka nastopajo kot zadnja neničelna števka enkrat, dvojka pa dvakrat.

Pozor: Razlika med številoma N in M je lahko tako velika, da je bolje, če ne obravnavamo vsakega števila med M in N posebej, ker bi podprogram predolgo tekel. Tvoj podprogram naj bo učinkovit!

2003.2.3 Različnost nizov

Rešitev: str. 25

Pogosto je koristno definirati neko mero različnosti med nizi znakov. Takšne mere radi definirajo tako, da predpišejo nek nabor operacij, ki jih je nad nizi dovoljeno izvajati, vsaki operaciji pripišejo tudi neko ceno, nato pa definirajo razdaljo med dvema nizoma kot ceno najcenejšega zaporedja operacij, ki predela prvi niz v drugega. (Cena zaporedja operacij je definirana kar kot vsota cen vseh posameznih operacij v njem.)

Tako bomo storili tudi pri tej nalogi. Omejili se bomo na nize, ki jih sestavljajo same male črke angleške abecede. Nad njimi dovolimo tri operacije:

(1) Dodajanje znaka: poljubno črko vrinemo na poljubno mesto v nizu, lahko tudi na začetek ali na konec. Če hočemo na primer v *zxc* dodati *q*, lahko dobimo *qzxc*, *zqxc*, *zxqc* ali pa *zxcq*.

(2) Brisanje znaka: zbrišemo lahko poljuben znak niza. Iz *xyzzy* bi lahko na primer z enim brisanjem dobili *yzzy*, *xzzy*, *xyzy* ali pa *xyzz*.

(3) Premikanje znaka: en znak lahko premaknemo na poljubno drugo mesto v nizu; učinek je tak, kot da bi ga najprej zbrisali in nato dodali na neko drugo mesto v nizu. Iz *spqr* bi lahko s premikanjem znaka *s* dobili *psqr*, *pqsr* in *pqrs*.

Cena vsakega dodajanja in brisanja naj bo 1, cena premikanja pa 0. **Napiši funkcijo** Razdalja, ki za dana dva niza izračuna ceno najcenejšega zaporedja operacij, ki predela prvi niz v drugega. Funkcija naj bo takšne oblike:

```
function Razdalja(S, T: string): integer;
```

ali, v C-ju:

```
int Razdalja(const char *S, const char *T);
```

2003.2.4 Pošiljanje sporočil

Rešitev: str. 26

V računalniškem omrežju podjetja MiSmoSoft je veliko računalnikov. Sistemski inženir je dobil nalogo, naj čim hitreje pošlje sporočilo na vse računalnike v omrežju. Vsi računalniki v omrežju imajo sposobnost prejemanja in pošiljanja sporočil. Pomagaj sistemcu ter mu **opiši postopek**, ki z uporabo večjega števila računalnikov čim hitreje pošlje sporočilo do vseh računalnikov v omrežju. Pri tem upoštevaj, da pošiljanje sporočila iz računalnika popolnoma zaposli ta računalnik za 1 sekundo (pošiljanje N sporočil iz enega računalnika traja torej N sekund). Pošiljanje sporočil iz več računalnikov je popolnoma neodvisno in hkratno. Vsak računalnik v omrežju ima enolično določen naslov, ki je 32-bitna številka (naslov IP); vsak računalnik tudi hrani naslove vseh računalnikov v mreži. Trajanje procesiranja sporočil na računalniku lahko zanemarimo. Iščemo torej nakrajši čas, v katerem obvestimo vse računalnike v omrežju.

Začetek pošiljanja sporočil sproži sistemski inženir, ki pošlje sporočilo na en računalnik v omrežju (to označi v sporočilu). Tvoja naloga je napisati algoritem, ki čim hitreje pošlje sporočila do vseh računalnikov. Ta algoritem se bo v nespremenjeni obliki izvajal na vseh računalnikih v omrežju. Bodi pozoren na to, da se bo začel tvoj algoritem na posameznem računalniku izvajati takrat, ko ta računalnik prvič prejme kakšno sporočilo.

Zahtevani algoritem zapiši v telo funkcije, ki se pokliče ob prejemu sporočila. Bodi pozoren na temeljit opis algoritma in podatkov, ki jih pošiljaš. Računalnik velja za obveščenega, ko prejme kakršnokoli sporočilo. Prejemanje več sporočil je dovoljeno, vendar brezpomensko.

Na voljo imaš naslednje deklaracije in podprograme:

```
const StRacunalnikov = ...; { Število računalnikov v omrežju. }
type NaslovT = ...; { Naslov računalnika v omrežju (npr. IP-številka). }

type SporociloT = record
  { Pri sporočilu, ki ga bo poslal sistemski inženir prvemu
    računalniku, bo spodnja vrednost gotovo true. Pri ostalih
    sporočilih je pač taka, kakršno pripravi računalnik-pošiljatelj. }
  PrvoSporocilo: boolean;
  { Tu lahko dopolniš to strukturo še s svojimi polji. }
  ...
end;

{ Pošlje sporočilo S na računalnik z naslovom Prejemnik.
  Ta podprogram potrebuje za svojo izvršitev eno sekundo. }
procedure PosljiSporocilo(S: SporociloT; Prejemnik: NaslovT); external;

{ V tabelo Naslovi vpiše naslove vseh računalnikov v mreži
  (vključno z našim), v naraščajočem vrstnem redu. }
type NasloviT = array [1..StRacunalnikov] of NaslovT;
procedure NasloviVsehRacunalnikov(var Naslovi: NasloviT); external;
```

```
{ Vrne naslov našega računalnika. }
function NasNaslov: NaslovT; external;

{ Ko naš računalnik prejme kakšno sporočilo, bo sistem
  poklical naš spodnji podprogram. Kot parametra dobi sporočilo
  in pošiljateljev naslov. }
procedure ObPrejemuSporocila(S: SporociloT; Posiljatelj: NaslovT);
begin
  { Tu vpiši svoj postopek. }
  ...
end; {ObPrejemuSporocila}
```

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Na začetku tekmovanja smo tekmovalcem najprej razdelili naslednja navodila. Nekaj minut kasneje so dobili tudi besedilo nalog, za reševanje pa so imeli slabe tri ure časa.

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*`.in` in izpisujejo svoje rezultate v *imenaloge*`.out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil mapo s svojim uporabniškim imenom ter navodili, ki jih pravkar prebiraš. Ko boš sedel pred računalnik, boš dobil nadaljna navodila za prijavo v sistem.

Na vsakem računalniku imaš na voljo enoto (disk) `U:`, na kateri lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku Pascal, C ali C++, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal in GNU C/C++. Za delo lahko uporabiš `turbo` (Turbo Pascal), `fp` oz. `ppc386` (FreePascal), `tc` (Turbo C), ali `gcc/g++` (GNU C/C++ — command line compiler). Ves potreben softver lahko najdeš na `c:\Programi` ter v meniju `Start` pod `Programs` in `Prevajalniki`.

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `rtk.exe`, ki ga lahko uporabiš za preverjanje svojih rešitev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

```
rtk imenaloge.pas
rtk imenaloge.c
rtk imenaloge.cpp
```

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Na spletni strani boš dobil obvestilo o tem, ali je program na testne primere odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Izjema je 1. naloga, kjer boste dobili vhodne datoteke že z nalogo, oddajati pa boste morali izhodne datoteke. Te se oddaja s klicem

`rtk` *ImeIzhodneDatoteke*

Imena datotek bodo podana v opisu naloge. Oddaja se vsako izhodno datoteko posebej in ni nujno, da oddaš datoteke za vse testne primere.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z drugimi datotekami kot z vhodno in izhodno. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov, prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi od 0 do 10 točk (praviloma 10, če je izpisal popolnoma pravilen odgovor, sicer pa 0; izjema je 2. naloga, ki dopuča tudi delno pravilne rešitve), nato pa se te točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M-3(N-1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk.

Izjema pri opisanem načinu točkovanja je 1. naloga, pri kateri se oddaja po eno izhodno datoteko za vsak testni primer. Za vsako dobiš od 0 do 10 točk. Skupno število točk pri tej nalogi dobimo tako, da za vsak testni primer upoštevamo najboljšo od izhodnih datotek, ki jih je tekmovalec oddal za ta primer. Število oddaj pri tej nalogi ne zmanjšuje števila točk.

Skupno število točk tekmovalca je vsota po vseh nalogah. Tekmovalce razvrstimo po skupnem številu točk.

Vsak tekmovalec se mora sam zase odločiti o tem, katerim nalogam bo posvetil svoj čas, v kakšnem vrstnem redu jih bo reševal in podobno. Verjetno je priporočljivo najprej reševati lažje naloge.

Poskusna naloga (ne šteje k tekmovanju) `poskus.in`, `poskus.out`

Napiši program, ki iz vhodne datoteke prebere eno celo število (le-to je v prvi vrstici, okoli njega ni nobenih dodatnih presledkov ipd.) in izpiše njegov desetkratnik v izhodno datoteko.

Primer vhodne datoteke:

```
123
```

Ustrezna izhodna datoteka:

```
1230
```

Primer rešitve:

```
program PoskusnaNaloga;
var T: text; i: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * i); Close(T);
end. {PoskusnaNaloga}
```

```
#include <stdio.h>
int main() {
    FILE *f = fopen("poskus.in", "rt");
    int i; fscanf(f, "%d", &i); fclose(f);
    f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * i);
    fclose(f); return 0;
}
```

```
#include <fstream.h>
int main() {
    ifstream ifs("poskus.in"); int i; ifs >> i;
    ofstream ofs("poskus.out"); ofs << 10 * i;
    return 0;
}
```

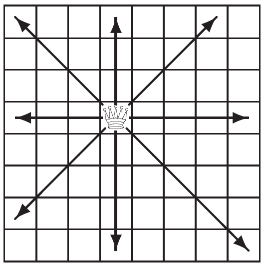
NALOGE ZA TRETJO SKUPINO

2003.3.1 Napadalne kraljice

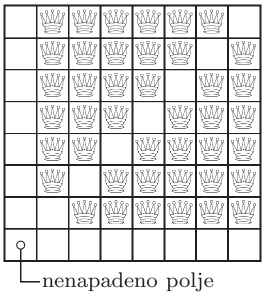
`dame01.out, dame02.out, ..., dame10.out`

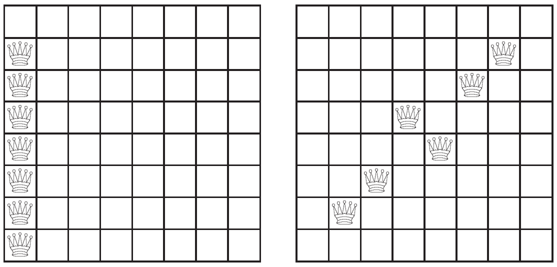
Rešitev: str. 28

Igra šah se igra na igralni plošči (šahovnici) z 8×8 polji. V igri nastopa več vrst figur, od katerih pa bo nas pri tej nalogi zanimala samo kraljica (dama). Kraljica se lahko premika po igralni plošči v osmih smereh (naravnost in po diagonalah) in sicer od svojega položaja pa vse do roba igralne plošče (glej zgornjo levo šahovnico na sliki). Za polja, na katera bi se neka kraljica načeloma lahko premaknila, pravimo, da jih „napada". Kraljica napada tudi polje, na katerem že zdaj stoji. Če postavimo na šahovnico več kraljic, pravimo, da je neko polje napadeno, če ga napada vsaj ena od kraljic na šahovnici. Na šahovnici običajne



Kraljica in polja, ki jih napada.





Napadena so vsa polja.

velikosti, torej z 8×8 polji, lahko na primer s sedmimi ali celo s šestimi kraljicami že napademo vsa polja šahovnice — če te kraljice primerno razporedimo (glej spodnji dve šahovnici na sliki). Seveda tudi večje število kraljic samo po sebi še ne zagotavlja, da bodo napadena vsa polja (zgornja desna šahovnica na sliki).

Ta problem lahko še malo posplošimo, če se namesto običajne šahovnice z 8×8 polji zanimamo tudi za šahovnice drugih velikosti. Zanimalo nas bo naslednje: če imamo šahovnico velikosti $n \times n$ in bi radi nanjo postavili k kraljic, kako naj jih postavimo, da bo napadenih čim več polj?

Pri tej nalogi ne boš oddajal izvorne kode programa, pač pa boš dobil deset vhodnih datotek, za vsako od njih pa moraš oddati po eno izhodno datoteko. Vsaka od *vhodnih datotek* vsebuje dve števili, naprej n in potem k, ločeni s po enim presledkom. V *izhodni datoteki* podaj nek razpored k kraljic na šahovnico velikosti $n \times n$, pri katerem je napadenih čim več polj. Prva vrstica izhodne datoteke naj vsebuje tri števila, ločena s po enim presledkom: najprej n, nato k in nato še skupno število napadenih polj pri razporedu kraljic. Sledi naj še n vrstic, ki predstavljajo razpored kraljic na šahovnici. Vsaka od teh vrstic naj ima n znakov `'.'` ali `'#'`. Pike predstavljajo prazna polja, znaki `'#'` pa polja, na katerih stoji kakšna kraljica. Znakov `'#'` mora biti točno k, torej ni dovoljeno, da bi na kakšnem polju šahovnice stalo po več kraljic.

Primer vhodne datoteke:

```
11 5
```

Primer pripadajoče izhodne datoteke:

```
11 5 117
#..........
...........
...........
.......#...
...#.......
...........
...........
........#..
....#......
...........
...........
```

Pozor: najti je mogoče tudi boljše rešitve. Z drugimi besedami, pet kraljic lahko na šahovnico 11×11 razporedimo tudi tako, da je napadenih več kot 117 polj.

Tule pa je deset vhodnih primerov, za katere moraš poiskati rešitve:

n	k	Oddaj datoteko s tem imenom
8	5	`dame01.out`
10	5	`dame02.out`
12	5	`dame03.out`
14	7	`dame04.out`
15	7	`dame05.out`
17	9	`dame06.out`
19	9	`dame07.out`
19	10	`dame08.out`
20	11	`dame09.out`
21	10	`dame10.out`

Posamezno rešitev oddaš tako, da pokličeš program `rtk` in mu podaš kot parameter ime izhodne datoteke:

`rtk dame`XX`.out`

kjer je XX eden od nizov `01`, `02`, …, `10`.

Točkovanje. Za vsak testni primer, pri katerem si oddal kakšno izhodno datoteko, lahko dobiš od 0 do 10 točk. Če oblika izhodne datoteke ni takšna, kot je predpisano v nalogi (ali pa se ne ujema z vhodnimi podatki za ta testni primer), dobiš 0 točk. Sicer pa je število točk odvisno od tega, kako dobra je tvoja rešitev v primerjavi z najboljšo rešitvijo, ki jo je našla tekmovalna komisija. Naj bo t število napadenih polj v tvoji rešitvi, m pa v najboljši rešitvi, ki jo je uspela najti komisija.

Če je. . .	. . . dobiš toliko točk:
$t \geq m$	10
$t = m - 1$	9
$t = m - 2$	8
$m - 5 \leq t < m - 2$	7
$m - 10 \leq t < m - 5$	5
$m - 20 \leq t < m - 10$	4
$t < m - 20$	3

Za vsak testni primer lahko oddaš tudi več različnih izhodnih datotek; v tem primeru se šteje najboljša med njimi. Število oddaj pa sámo po sebi ne vpliva na to, koliko točk dobiš.

2003.3.2 **Smučarji** `smucarji.in`, `smucarji.out`

Rešitev: str. 33

Na posamezni smučarski tekmi za svetovni pokal dobi vsak tekmovalec določeno število točk (0 ali več). Število prejetih točk je odvisno od njegove uvrstitve na tej tekmi. Mednarodna smučarska zveza vodi tudi razvrstitev v skupnem seštevku, kjer je vsak tekmovalec predstavljen z vsoto točk, ki jih je dobil na vseh tekmah trenutne sezone.

Ko manjka do konca sezone le še ena tekma, se opazovalci radi sprašujejo, katera je najslabša ali najboljša možna uvrstitev v skupnem seštevku, ki bi jo nek tekmovalec utegnil dobiti po zadnji tekmi. Recimo na primer, da se na posamezni tekmi dobi 100 točk za prvo mesto, 50 za drugo, 25 za tretje, za ostala pa še manj. Recimo še, da v skupnem seštevku trenutno vodi tekmovalec A, drugouvrščeni pa je tekmovalec B, ki za A-jem zaostaja za 60 točk. Iz tega že lahko sklepamo, da bo A po zadnji tekmi v skupnem seštevku ali prvi ali drugi, gotovo pa ne slabši. Da bi ga kdo v skupnem seštevku prehitel, bi namreč potreboval vsaj 60 točk, toliko pa lahko na eni tekmi dobi samo zmagovalec. Torej lahko A-ja v skupnem seštevku prehiti največ en tekmovalec (zmagovalec zadnje tekme, če se A na njej uvrsti dovolj slabo).

Tvoj program bo v vhodni datoteki dobil naslednje podatke. (Vsi podatki so cela števila, vsako je v samostojni vrstici, okoli njih ni presledkov, praznih vrstic ali česa podobnega.)

- V prvi vrstici je m, število smučarjev, ki na posamezni tekmi dobijo kaj točk. Tisti, ki se uvrstijo na $(m + 1)$-vo ali slabše mesto, ne dobijo nič točk.
- V naslednjih m vrsticah je za vsako uvrstitev od prve do m-te podano število točk, ki jih dobi tekmovalec za to uvrstitev. Če število točk, ki jih prejme i-touvrščeni, označimo s t_i, lahko predpostaviš, da velja: $100\,000 \geq t_1 > t_2 > t_3 > \cdots > t_{m-1} > t_m > 0$.
- V naslednji vrstici je n, število smučarjev v skupnem seštevku svetovnega pokala.
- V naslednjih n vrsticah je podano za vsakega od teh smučarjev njegovo število točk v skupnem seštevku pred zadnjo tekmo sezone; najprej za vodilnega v skupnem seštevku, nato za drugega, itd., nazadnje pa za zadnjega. Če je a_i število točk, ki jih ima i-ti najboljši v skupnem seštevku, lahko predpostaviš, da velja: $100\,000 \geq a_1 \geq a_2 \geq a_3 \cdots \geq a_{n-1} \geq a_n \geq 0$.

Tako m kot n sta pozitivni celi števili, ki nista manjši od 1 in nista večji od 3 000.

V izhodno datoteko naj tvoj program zapiše n vrstic. V vsaki naj bosta dve pozitivni celi števili. Prvo število v i-ti vrstici naj bo najboljši možni položaj, ki ga bi utegnil imeti v skupnem seštevku po zadnji tekmi tisti tekmovalec, ki je v skupnem seštevku pred zadnjo tekmo na i-tem mestu. Drugo število v i-ti vrstici naj bo najslabši možni položaj, ki bi ga utegnil imeti ta tekmovalec v skupnem seštevku po zadnji tekmi.

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:
`6`	`1 3`
`50`	`1 3`
`25`	`1 4`
`20`	`2 8`
`10`	`3 10`
`5`	`3 10`
`3`	`4 10`
`10`	`4 10`
`1000`	`4 10`
`980`	`4 10`
`971`	
`930`	
`925`	
`923`	
`920`	
`915`	
`912`	
`910`	

Dogovorimo se še, da na zadnji tekmi sezone velja posebno pravilo: ne more se zgoditi, da bi si dva ali več tekmovalcev delilo isto mesto (torej tudi ne morejo dobiti enakega števila točk). Če bi imelo več tekmovalcev na stotinko enak čas, jih bodo pač razvrstili s pomočjo žreba in zakulisne kuhinje. (Lahko pa si več ljudi deli mesto v skupnem seštevku. Če sta dva sedma, je tisti takoj za njima deveti in podobno.) Mogoče pa je, da kak tekmovalec (lahko celo zelo veliko tekmovalcev) med tekmo odstopi (ali pa sploh ne štartajo ali pa so diskvalificirani) in takšni ne dobijo na tej tekmi nobenih točk.

Če pravilno izračunaš le najboljše možne uvrstitve, pri najslabših pa je kakšna napaka, dobiš pri tistem testnem primeru le tri točke; če pravilno izračunaš najslabše, ne pa najboljših, dobiš pri tistem testnem primeru sedem točk; če je pravilno oboje, dobiš pri tistem testnem primeru vseh deset točk.

2003.3.3 **Vplivi** `vplivi.in`, `vplivi.out`

Rešitev: str. 36

Kajti če si je kdajkoli kdo zaslužil našo grmado, si to ti. Jutri te bom sežgal. Dixi.
Veliki inkvizitor v *Bratih Karamazovih*

Véliki inkvizitor vodi skupino n inkvizitorjev (v tem številu je zajet tudi sam). Ker vseh procesov proti krivovercem ne more neposredno nadzirati sam, obenem pa ne bi rad, da bi kak heretik ušel grmadi, bi rad občasno poslal do posameznega inkvizitorja kak nasvet, s katerim bi lahko vplival na njegovo razsodbo. Nerodno pa je, da je pripravljen vsak inkvizitor upoštevati nasvete le nekaterih drugih (veliki inkvizitor je lahko med njimi ali pa tudi ne), pa še pri tem ne vplivajo nanj vsi enako močno. Zato ga zanima, kako močan vpliv ima lahko na vsakega izmed preostalih inkvizitorjev, če si pametno izbere obveščevalne verige.

Oštevilčimo inkvizitorje s števili $1, 2, \ldots, n$. Veliki inkvizitor seveda dobi številko 1. Vpliv inkvizitorja i na inkvizitorja j označimo s $p(i, j)$. To je celo število, večje ali enako 0. (Vplivi niso nujno obojestranski: $p(i, j)$ ni nujno enak $p(j, i)$.) Če vpliva inkvizitor i_0 na inkvizitorja i_1, ta na i_2, ta na i_3, …, i_{k-1} pa vpliva na i_k, pravimo temu *pot* od i_0 do i_k in definiramo vpliv te poti kot $\min\{p(i_0, i_1), p(i_1, i_2), \ldots, p(i_{k-1}, i_k)\}$. *Najvplivnejša pot* od i do j je taka pot od i do j, ki ima vsaj tolikšen vpliv kot vsaka druga pot od i do j.

Zagotovljeno je, da od velikega inkvizitorja do vsakega drugega obstaja vsaj ena pot z vplivom, večjim od 0. Veliki inkvizitor te prosi, da mu za vsakega inkvizitorja i ($i = 2, \ldots, n$) izračunaš vpliv najvplivnejše poti od 1 do i. (Mogoče sicer obstaja več različnih najvplivnejših poti od 1 do i, ampak že iz definicije je očitno, da imajo vse enak vpliv.)

Vhodna datoteka: v prvi vrstici je n (število vseh inkvizitorjev, vključno z velikim inkvizitorjem), v drugi neko pozitivno celo število m, sledi pa m vrstic, ki navajajo vse neničelne vplive. V vsaki od teh vrstic so tri števila: i, j in $p(i, j)$, ločena s po enim presledkom. Za vse te $p(i, j)$ v datoteki vedno velja: $1 \leq p(i, j) \leq 1\,000\,000\,000$. Če pa za nek par inkvizitorjev (i, j) v datoteki ni ustrezne vrstice, velja, da je $p(i, j) = 0$. Zagotovljeno je tudi, da je $p(i, i) = 0$ za vsak i. Predpostaviš lahko, da velja $1 \leq n \leq 1\,000$ in $1 \leq m \leq 200\,000$.

Izhodna datoteka naj ima $n - 1$ vrstic. V prvi vrstici naj bo vpliv najvplivnejše poti od 1 do 2, v drugi vpliv najvplivnejše poti od 1 do 3 in tako naprej. V zadnji vrstici naj bo torej vpliv najvplivnejše poti od 1 do n.

Primer vhodne datoteke:

```
5
7
1 2 10
1 3 5
2 4 8
3 5 7
4 3 9
2 5 2
5 4 12
```

Pripadajoča izhodna datoteka:

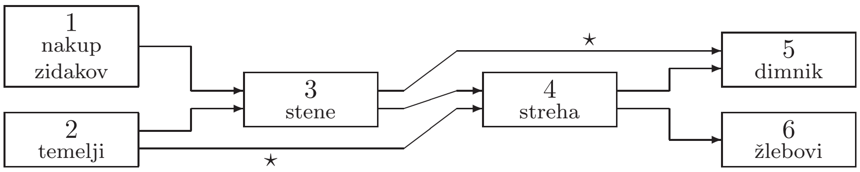
```
10
8
8
7
```

2003.3.4 Vodenje projektov `projekti.in`, `projekti.out`

Rešitev: str. 41

Gradbeno podjetje Zidarstvo Polde, d. d., uporablja računalniško podprto načrtovanje projektov. Projekt opišejo kot množico aktivnosti, vsaka aktivnost pa je lahko odvisna od poljubno mnogo drugih aktivnosti, ki morajo biti končane, preden se lahko začne. Direktor Polde si zaradi lažje predstave rad nariše sliko celotnega projekta (glej primer), vendar njegovi podrejeni pri navajanju odvisnosti med aktivnostmi radi pretiravajo in navajajo odvečne podatke (kot na primer odvisnosti $\star$ na spodnji sliki), ki naredijo sliko nepregledno.

Polde te prosi za pomoč pri odstranjevanju odvečnih podatkov. Odvečne odvisnosti so vse, ki že izhajajo iz drugih navedenih odvisnosti. V podatkih je odvisnost med dvema aktivnostima navedena *največ enkrat.* Nobena aktivnost tudi nikoli ni posredno ali neposredno odvisna od same sebe.



Oblika *vhodnih podatkov* je: v prvi vrstici je število aktivnosti, n; v drugi vrstici je število odvisnosti, m; sledi m vrstic, ki vsebujejo po dve števili: prvo je številka neke aktivnosti, drugo pa številka neke aktivnosti, ki je odvisna od prve. Veljalo bo $1 \leq n \leq 1000$, $0 \leq m \leq 10000$.

Tvoja naloga je izločiti največje možno število parov $\langle$*aktivnost, odvisna aktivnost*$\rangle$ tako, da bo zahtevano zaporedje aktivnosti še vedno enako. (Z drugimi besedami, če je bila neka aktivnost prvotno odvisna (posredno ali neposredno) od neke druge aktivnosti, mora biti tudi po končanem izločanju parov še vedno vsaj posredno odvisna od nje.) Na zgornjem primeru sta to odvisnosti, označeni z zvezdico ($\star$). **Izločene** odvisnosti izpiši v *izhodno datoteko* kot pare števil (najprej številka aktivnosti, nato številka odvisne aktivnosti). Vrstni red, v katerem izpišeš izločene odvisnosti, ni pomemben, ne smeš pa nobene odvisnosti navesti več kot enkrat.

Primer vhodne datoteke (zgornja slika):

```
6
7
1 3
2 3
3 4
2 4
4 5
3 5
4 6
```

Ena od možnih pripadajočih izhodnih datotek:

```
2 4
3 5
```

2003.3.5 Knjižnica `knjige.in`, `knjige.out`

Rešitev: str. 45

V neki knjižnici imajo ogromno starih knjig, ki so zanimive predvsem zaradi zgodovinske vrednosti. Ker imajo zanje na voljo le en regal, morajo med njimi narediti nek izbor tako, da jih čimveč razvrstijo na police, preostale pa prepustijo muzeju za kulturno dediščino. Knjige so različno debele, potrebno pa jih je razvrstiti v kronološkem vrstnem redu izdaje — od leve proti desni in potem naprej na naslednji polici... Tako smejo biti na prvi polici na primer le knjige, izdane v letih od 1900 do 1903, na drugi le tiste od 1903 do 1908 itd. (letnice so tu podane le kot primer!). Z drugimi besedami, vse knjige na eni polici morajo imeti zgodnejši datum izdaje kot knjige na naslednji polici.

Tvoja naloga je, da najdeš največje možno število knjig, ki jih še lahko spravijo v regal. Podano imaš število knjig n, število polic m in širino regala d (to je v bistvu širina vsake posamezne police) ter seznam debelin knjig, podanih v centimetrih. Ta seznam je že urejen v kronološkem vrstnem redu izdaje, tako da datumov izdaj ne potrebuješ.

Vhod: najprej prebereš trojico celih števil N, M in d, za katere bo zagotovo veljalo $1 \leq N \leq 1000$, $1 \leq M \leq 100$ ter $1 \leq d \leq 100$, nato pa še zaporedje N celih števil d_i (za katera velja $1 \leq d_i \leq 10$), ki predstavlja debeline knjig, podane v kronološkem vrstnem redu izdaje. Nobena knjiga ni debelejša od širine police: za vsak i velja $d_i \leq d$.

Izhod: izpiši največje število knjig iz danega zaporedja knjig, ki jih pod danimi pogoji še lahko spravijo v regal.

Primer vhodne datoteke:

```
10 3 5
1 4 2 1 3 4 1 2 2 1
```

Pravilni odgovor za to vhodno datoteko:

```
8
```

Primer takšne razporeditve osmih knjig na tri police je:

$\underline{1}\ 4\ \underline{2\ 1}\ \ \big|\ \ \underline{3}\ 4\ \underline{1}\ \ \big|\ \ \underline{2\ 2\ 1}.$

Podčrtane so debeline tistih knjig, ki smo jih res uvrstili na police; navpični črti določata meji med policami.

LETO 2003, TEKMOVANJE V POZNAVANJU UNIXA

Nasvet

Pri vaših rešitvah bo komisija za ocenjevanje upoštevala poleg pravilnosti in robustnosti tudi njihovo elegantnost in preprostost.

Rešitev: str. 47

2003.U.1

V okolju UNIX je obdelava znakovnih podatkov zelo pomembna. Strežniški programi običajno vodijo datoteko dogodkov — dnevnik (*logfile*).

Napišite program `preglej`, ki mu podamo kot parametre imena treh datotek po vrsti, kot kaže zgled:

```
preglej datoteka1.log regexp-da.dat regexp-ne.dat
```

Prva datoteka je dnevniška z običajnim besedilom (*text file*) in vsebuje zapise dogodkov. Drugi datoteki vsebujeta vsaka po eno vrstico z regularnim izrazom. Dnevnik je potrebno prebrskati in izpisati vse vrstice, ki ustrezajo regularnemu izrazu iz prve datoteke ter ne ustrezajo regularnemu izrazu iz druge datoteke. Če morebiti katera od podanih datotek ne obstaja, naj program izpiše besedilo „`NAPAKA`“ v vrstici na standardni izhod.

Namig: Privzamete lahko, da je dnevniška datoteka vselej dostopna in na voljo, da se med tekom skripte ne dodajajo novi zapisi v dnevnik in da se skripte nikoli ne poganja večkrat hkrati.

Rešitev: str. 47

2003.U.2

Nekatere datoteke z večpredstavnimi vsebinami imajo glavo, ki se začne z nizom „`RIFF`“, temu pa lahko sledijo poljubni podatki, ki se končajo z nizom „`data`“. Za nizom „`data`“ je preostanek datoteke.

Napiši program, ki mu kot parameter podaš ime tako oblikovane datoteke, program pa bo iz nje izrezal vse podatke med nizoma „`RIFF`“ in „`data`“, vključno s tema nizoma. Če na začetku datoteke ni niza „`RIFF`“, naj ne odstani podatkov.

Rešitev: str. 48

2003.U.3

Sistemi Linux v navideznem datotečnem sistemu `/proc` hranijo mnoge podatke o stanju sistema in programov, ki tečejo v njem. Z branjem teh podatkov znajo programi prikazati stanje sistema na različne načine.

Napišite program, ki na standardni izhod izpiše polno pot do izvršilne datoteke očeta in njeno ime. Oče je proces, ki je pognal program, ki ste ga napisali. Če kot zgled v ukazni lupini `bash` poženemo nek program `preskus` s parametri

```
preskus -v test -o izhod.dat
```

in bi ta program hotel izpisati pot do izvršilne datoteke svojega očeta, bi moral izpisati pot do lupine `/bin/bash`. Če pa bi napisali svoj program za izpisovanje poti do izvršilne datoteke očeta in ga pognali iz programa

```
preskus -v test -o izhod.dat
```

bi moral naš program izpisati pot, kjer je na disku shranjena izvršilna datoteka programa `preskus`, denimo `/usr/local/bin/preskus`. V primeru

```
/usr/bin/preskus
```

bi to bil niz

```
/usr/bin/preskus
```

V primeru

```
./preskus
```

pa je to lahko nekaj takega:

```
/home/uporabnik/preskus
```

2003.U.4

Rešitev: str. 49

Nekateri programi so napisani tako, da delujejo kot filtri (berejo podatke s standardnega vhoda in pišejo na standardni izhod), drugim pa moramo podati vhodno in izhodno datoteko.

Program `sort`, denimo, zna delati celo na oba načina.

```
sort datoteka1 -o datoteka2
sort datoteka1 > datoteka2
```

Razliko opazimo takrat, ko sta `datoteka1` in `datoteka2` ista datoteka. Takrat druga oblika ukaza poreže datoteko, preden je urejanje končano in izgubimo njeno vsebino.

Napiši program `prepis`, ki bo prepisal vhodno datoteko na izhodno, podobno kot `cat datoteka1 > datoteka2`, na začetek datoteke pa bo dodal niz „`PREPIS`“.

Če podamo le en parameter, bo program podano datoteko izpisal na standardni izhod. Če mu podamo izhodno datoteko, bo izhod pisal nanjo. V tem primeru se bo tudi pametno odzval, kadar sta podani datoteki ista datoteka, kar pomeni, da bo vhodni datoteki na njen začetek dodal zahtevani niz.

REŠITVE NALOG ZA PRVO SKUPINO

R2003.1.1 Dva kupa števil

Naloga: str. 1

Možnih je več rešitev. Začnemo lahko z dvema praznima kupoma in nato pregledujemo števila od večjih proti manjšim ter vsako število odložimo na tisti kup, ki ima trenutno manjšo vsoto (če imata oba enako, pa na prvi kup). Spodnji podprogram naredi dva prehoda po vseh številih in v prvem izpisuje, kaj je odložil na prvi kup, v drugem prehodu pa, kaj je odložil na drugi kup.

```
procedure Razdeli(N: integer);
var i, Kup, Kam: integer; Vsota: array [1..2] of integer;
begin
  for Kup := 1 to 2 do begin
    Write(Kup, '. kup:');
    Vsota[1] := 0; Vsota[2] := 0;
    for i := N downto 1 do begin
```

```
      if Vsota[1] <= Vsota[2] then Kam := 1 else Kam := 2;
      Vsota[Kam] := Vsota[Kam] + i;
      if Kam = Kup then Write(' ', i);
    end; {for i}
    WriteLn(' Vsota: ', Vsota[Kup]);
  end; {for Kup}
end; {Razdeli}
```

Recimo, da imata v nekem trenutku oba kupa enako vsoto; naslednje število (recimo k) bomo torej odložili na prvi kup. Ta ima zdaj večjo vsoto kot drugi, zato bomo naslednje število, $k-1$, odložili na drugi kup. Prvi ima še vedno večjo vsoto, zato tudi $k-2$ odložimo na drugi kup. Zdaj ima večjo vsoto drugi in bomo naslednje število, $k-3$, odložili spet na prvi kup. Zdaj pa, ker je $k+(k-3) = (k-1)+(k-2)$, imata oba kupa spet enako vsoto.

Torej bomo po vsakih štirih pregledanih številih imeli na obeh kupih enako vsoto in tudi enako mnogo števil. Če je N večkratnik števila 4, bo veljalo to tudi na koncu, po pregledu vseh števil, iz česar vidimo, da je ta rešitev najboljša možna. Če N ni večkratnik števila 4, pa nam po pregledu vseh četveric števil (spomnimo se, da imamo takrat dva kupa z enako vsoto in enako mnogo števili) ostane še število 1 ali pa števili 2 in 1 ali pa števila 3, 2 in 1, odvisno od tega, kakšen ostanek ima N pri deljenju s 4. Če nam ostane le 1 ali pa 2 in 1, vidimo, da mora biti vsota vseh števil od 1 do N liha (ker če smo vsa dosedanja števila razdelili na dva kupa z enako vsoto, mora biti njihova skupna vsota soda, tu pa smo ji prišteli še 1 ali pa $2+1$, kar je liho), torej se jih sploh ne da razdeliti na dva kupa z enako vsoto; naš podprogram bi dobil na enem kupu za 1 večjo vsoto kot na drugem, kar je v tem primeru torej najboljša rešitev. Če pa nam na koncu ostanejo števila 3, 2 in 1, bi naš algoritem na koncu dobil na obeh kupih enako vsoto, pri čemer bi bilo na enem kupu eno število več kot na drugem; boljše rešitve ni, saj mora biti N lih (če smo doslej gledali po štiri števila skupaj in so nam na koncu ostala tri) in se torej ne da dobiti dveh kupov z enako mnogo števili.

Oglejmo si primer za $N = 13$. Če razdelimo števila na skupine po štiri, dobimo:

$$13\ 12\ 11\ 10 \qquad 9\ 8\ 7\ 6 \qquad 5\ 4\ 3\ 2 \qquad 1.$$

Prvi kup bi torej dobil števila $13, 10, 9, 6, 5, 2$ in na koncu še 1, drugi kup pa bi dobil $12, 11, 8, 7, 4$ in 3. Tako ima prvi kup sedem števil z vsoto 46, drugi pa šest števil z vsoto 45.

Takšno obravnavanje števil v skupinah po štiri lahko zapišemo tudi bolj eksplicitno:

```
procedure Razdeli2(N: integer);
var i, Vsota: integer;
begin
  Write('1. kup: ', N); Vsota := N; i := N - 4;
  while i >= 0 do begin
    Write(' ', i + 1); if i > 0 then Write(' ', i);
    Vsota := Vsota + i + (i + 1); i := i - 4;
  end; {while}
  WriteLn(' Vsota: ', Vsota);
  Write('2. kup:'); Vsota := 0; i := N - 2;
  while i >= 0 do begin
```

```
    Write(' ', i + 1); if i > 0 then Write(' ', i);
    Vsota := Vsota + i + (i + 1); i := i - 4;
  end; {while}
  WriteLn(' Vsota: ', Vsota);
end; {Razdeli2}
```

Do enako dobrih razbitij na dva kupa pa pridemo tudi z naslednjim razmislekom. Recimo, da bi šli po vrsti od 1 do N in dajali števila izmenično na prvi in na drugi kup. Pri $N = 13$ bi dobili:

prvi kup:	1	3	5	7	9	11	13
drugi kup:	2	4	6	8	10	12.	

Če drugo vrstico malo zamaknemo,

prvi kup:	1	3	5	7	9	11	13
drugi kup:		2	4	6	8	10	12,

vidimo, da dobi pri vsakem paru števil (za 1 si mislimo, da je v paru z 0) prvi kup za eno večje število kot drugi. Parov je sedem, torej ima prvi kup za sedem večjo vsoto kot drugi. Če zdaj v treh parih zamenjamo števili (tisto, ki je bilo prej v prvem kupu, pride v drugega in obratno), se vsota prvemu zmanjša za tri, drugemu pa poveča za tri, torej se vsoti zdaj razlikujeta le še za 1.

prvi kup:		2	4	7	9	11	13
drugi kup:	1	3	5	6	8	10	12,

Prvi kup ima zdaj vsoto 46, drugi pa 45.

Hitro se lahko prepričamo, da bi lahko podoben razmislek opravili tudi v primerih, ko je N sod (takrat nam druge vrstice ne bi bilo treba zamikati, bi pa imel zato drugi kup večjo vsoto kot prvi; vsoti bi spet zbližali s pomočjo zamenjav, na enak način kot prej).

Za izvedbo s programom je ta rešitev malo bolj nerodna, ker je preračunavanje, na kateri kup gre katero število, malo bolj zapleteno. Po tisti prvi delitvi (števila izmenično na en in na drugi kup) ostane $p := (N + 1) \operatorname{div} 2$ parov, torej ima en kup za p večjo vsoto kot drugi. Torej bo dovolj, če izvedemo zamenjavo v prvih $p \operatorname{div} 2$ parih.

```
procedure Razdeli2(N: integer);
var P, Vsota, i, j, Kup: integer;
begin
  P := (N + 1) div 2;
  for Kup := 1 to 2 do begin
    Write(Kup, '. kup:'); Vsota := 0;
    for i := 1 to P do begin
      { Prvi kup naj bo tisti, ki bi imel brez zamenjav
        večjo vsoto. Od vsakega para bi torej ta kup dobil
        j, drugi pa j - 1. Če je N lih, se moramo delati,
        da začnemo s parom (0, 1), sicer pa s parom (1, 2). }
      j := 2 * i - (N mod 2);
      { Število, ki je v paru z j, je j - 1. Tega moramo
        uporabiti, če smo na drugem kupu ali pa če smo
        izvedli pri tem paru zamenjavo (ne pa, če velja oboje!). }
```

```
      if (Kup = 1) = (i <= P div 2) then j := j – 1;
      { Izpišimo število in ga dodajmo v vsoto. }
      Vsota := Vsota + j; if j > 0 then Write(' ', j);
    end; {for i}
    WriteLn(' Vsota: ', Vsota);
  end; {for Kup}
end; {Razdeli3}
```

R2003.1.2 „Pet čevljev merim, palcev pet“

Naloga: str. 1

Za lažje računanje z merskimi enotami (seštevanje in odštevanje) lahko vse količine najprej preračunamo v najmanjšo mersko enoto, torej v palce. Da ne bomo pisali vsake reči za vseh sedem enot posebej, jih kar oštevilčimo od 1 (liga) do 7 (palec). Vrednost Enota[i] nam bo povedala, kolikokrat je enota $i-1$ daljša od enote i. Ta preračun lahko opravimo od večjih enot proti manjšim: število lig pomnožimo s tri in prištejemo število milj; to pomnožimo z osem in prištejemo število furlongov; itd. Po vsakem takem koraku smo vse enote, večje od trenutne, pretvorili v trenutno, tako da na koncu dobimo prvotno količino izraženo v palcih. Potem ni težko seštevati in odštevati, na koncu pa moramo rezultat pretvoriti nazaj, da ga bomo lahko izpisali; to je obratna operacija od pretvorbe v palce. Kjer smo prej množili in prištevali, moramo zdaj deliti, računati ostanke in odštevati. Količino 5 000 jardov bi na primer delili z 220 (ker je 220 jardov en furlong) in ker je $5\,000 = 22 \cdot 220 + 160$, vemo, da je 5 000 jardov enako 22 furlongom in 160 jardom. Furlonge bi potem na enak način pretvorili v milje in furlonge in tako naprej.

```
program MerskeEnote;

{ Enote so oštevilčene od 1 (lige) do 7 (palci).
  Enota[i] pove, kolikokrat je enota i – 1 daljša od enote i. }
const Enota: array [1..7] of integer= (1, 3, 8, 220, 3, 3, 4);

function Preberi: integer; { vrne prebrano dolžino v palcih }
var i, e, Dolzina: integer;
begin
  Dolzina := 0;
  for i := 1 to 7 do begin
    { Vrednost Dolzina je zdajle v enoti i – 1. Pretvorimo jo v enoto i. }
    Dolzina := Dolzina * Enota[i];
    { Preberimo količino enote i in jo prištejmo dolžini. }
    Read(e); Dolzina := Dolzina + e;
  end; {for}
  Preberi := Dolzina; ReadLn;
end; {Preberi}

{ Dolzina naj bo v palcih, Zapisi pa jo zapiše, kot zahteva naloga. }
procedure Zapisi(Dolzina: integer; S: string);
var i: integer; e: array [1..7] of integer;
begin
  for i := 7 downto 2 do begin
    { Vrednost Dolzina je zdajle v enoti i.
      Poglejmo, koliko se ne bo dalo izraziti z večjimi enotami. }
    e[i] := Dolzina mod Enota[i];
    { Preostanek pretvorimo v enoto i – 1. }
```

```
      Dolzina := Dolzina div Enota[i];
   end; {for}
   e[1] := Dolzina;
   for i := 1 to 7 do Write(e[i], ' ');
   WriteLn(S);
end; {Zapisi}

var Stranke, Tovarna, i: integer;
begin
   Stranke := 0; for i := 1 to 10 do Stranke := Stranke + Preberi;
   Tovarna := Preberi; Write('Naročil si ');
   if Stranke < Tovarna then Zapisi(Tovarna − Stranke, 'preveč blaga.')
   else if Stranke > Tovarna then Zapisi(Stranke − Tovarna, 'premalo blaga.')
   else WriteLn('ravno prav blaga.');
end. {MerskeEnote}
```

Druga možnost bi bila, da bi seštevali in odštevali kar v predstavitvi, razbiti na posamezne enote. Ta postopek bi bil tak kot pisno seštevanje ali odštevanje, le meja za prenos naprej je od mesta do mesta različna. Na primer, pri običajnem pisnem seštevanju pride do prenosa na naslednje mesto, če je bila vsota na prejšnjem večja ali enaka 10; tu pa mora priti do prenosa, če je bila vsota na prejšnjem mestu dovolj velika, da bi iz tega dobili že vsaj eno večjo enoto. Podobno bi razmišljali tudi pri odštevanju. Vendar pa bi imeli z vsem tem po vsej verjetnosti več dela kot z gornjo rešitvijo.

R2003.1.3 Glasovanje

Pomagali si bomo s tabelo (StGlasov v spodnjem podprogramu), v kateri bomo za vsakega kandidata hranili število volilcev, ki so glasovali zanj. Na začetku postavimo vse elemente te tabele na 0, nato pa se sprehodimo po vseh glasovih in pri vsakem povečajmo števec pri tistem kandidatu, na katerega se ta glas nanaša. Na koncu ta števila glasov uporabimo, da vemo, koliko zvezdic izpisati pri posameznem kandidatu. Naloga: str. 2

```
program Glasovanje;

const MaxStVolilcev = 10; MaxStKandidatov = 10;
type TabelaT = array [1..MaxStVolilcev] of integer;

procedure Histogram(StKandidatov, StVolilcev: integer; Glasovi: TabelaT);
var
   StGlasov: array [1..MaxStKandidatov] of integer;
   i, j: integer;
begin
   for i := 1 to StKandidatov do StGlasov[i] := 0;
   for i := 1 to StVolilcev do
      StGlasov[Glasovi[i]] := StGlasov[Glasovi[i]] + 1;
   for i := 1 to StKandidatov do begin
      Write(i, ':');
      for j := 1 to StGlasov[i] do Write('*');
      WriteLn;
   end; {for}
end; {Histogram}

const Glasovi: TabelaT = (1, 3, 2, 4, 1, 4, 7, 6, 1, 2);
```

```
begin
  Histogram(7, 10, Glasovi);
end. {Glasovanje}
```

Še primer enovrstične rešitve v jeziku perl:

```
perl −ne '$k[$_]++; END { printf("%d:%s\n", $_, "*" x $k[$_]) for (1..$#k) }'
```

Stikalo −ne pove, da je program naveden kot naslednji parameter v ukazni vrstici (e) in da naj ga interpreter izvede po enkrat za vsako vrstico vhodne datoteke (n). Program predpostavi, da je v vsaki vrstici naveden en glas; vsebino trenutne vrstice dobimo v spremenljivki $_ in jo uporabimo kot indeks v tabelo k, kjer ustreznemu elementu povečamo vrednost za 1. Preostanek programa, „END { . . . }“ je podprogram po imenu END; če obstaja tak podprogram, ga interpreter pokliče na koncu, po tistem, ko je že obdelal celo vhodno datoteko. Takrat v tabeli k že piše, koliko glasov je prejel posamezni kandidat, zato se lahko lotimo risanja histograma. Izraz $#k pomeni število elementov v tabeli k. „**for** (1..$#k)“ za stavkom, ki kliče printf, pomeni, da se bo ta stavek izvedel po enkrat za vsako število od 1 do $#k (trenutno število pa bomo videli v spremenljivki $_). Operator x pomeni ponavljanje niza: „"*" x $k[$_]“ je torej niz $k[$_] zvezdic, to pa je ravno toliko, kolikor glasov je dobil kandidat številka $_.

R2003.1.4 Radar

Naloga: str. 2

Naloga pravi, da je meritev preveč, da bi lahko vse shranili v pomnilnik; če bi jih bilo manj, bi lahko vse prebrali v pomnilnik, jih uredili in tako ugotovili, katere so največje. (Pravzaprav bi bilo to časovno potratno, tudi če bi imeli dovolj pomnilnika za vsa števila.) Ker je meritev veliko, tudi ne bi bilo pametno iti dvajsetkrat skozi celotno datoteko (da bi npr. pri prvem prehodu poiskali največje število, pri drugem drugo največje in tako naprej).

Raje si med branjem vzdržujmo tabelo dvajsetih največjih izmed doslej prebranih števil. Ko preberemo novo število (recimo x), ga primerjamo s tistim, ki je bilo doslej dvajseto največje (recimo mu y); če je x večje, lahko y pozabimo in si namesto njega zapomnimo x.

Dvajset največjih doslej znanih števil lahko hranimo urejena po velikosti ali pa tudi ne. Vsaka od teh dveh različic ima svoje dobre in slabe strani. Če jih hranimo urejena po velikosti, bomo imeli vedno pri roki dvajseto največje doslej znano, vendar pa bo zato vstavljanje novega števila v tabelo malo zahtevnejše (ker ga bo treba včasih vriniti nekam na sredo tabele in ostala števila zato zamakniti za eno mesto). Če jih hranimo neurejena, je vstavljanje enostavno (preprosto vpišemo x v tisto celico, kjer je bil prej y), vendar pa bi morali načeloma vsakič prečesati celo tabelo dvajsetih števil, da bi ugotovili, katero je najmanjše med njimi. Temu se lahko izognemo, če si v neki spremenljivki zapomnimo, katero je najmanjše; ta podatek je treba potem popraviti le, ko vpišemo v tabelo novo število.

```
program RadarZUrejenoTabelo;
const N = 20;
var Tabela: array [1..N + 1] of real; i: integer;
    T: text; x: real;
begin
```

```
  Assign(T, 'podatki.txt'); Reset(T);
  for i := 1 to N do Tabela[i] := 0;
  while not Eof(T) do begin
    ReadLn(T, x);
    { Radi bi imeli manjše elemente na koncu tabele. Torej, kolikor je na koncu
      tabele elementov, ki so manjši od x, jih premaknimo za eno mesto naprej.
      Prav zato je tabela nalašč za eno celico daljša (ima N + 1 namesto N celic). }
    i := N;
    while i > 0 do
      if Tabela[i] >= x then break
      else begin Tabela[i + 1] := Tabela[i]; i := i − 1 end;
    { Zdaj vemo, da je Tabela[i] >= x, tako da moramo
      x postaviti eno mesto za njim. }
    Tabela[i + 1] := x;
  end; {while}
  Close(T);
  for i := 1 to N do WriteLn(Tabela[i]:0:2);
end. {RadarZUrejenoTabelo}

program RadarZNeurejenoTabelo;
const N = 20;
var Tabela: array [1..N] of real; i, KjeNajmanjsi: integer;
    T: text; x: real;
begin
  Assign(T, 'podatki.txt'); Reset(T);
  KjeNajmanjsi := 1; for i := 1 to N do Tabela[i] := 0;
  while not Eof(T) do begin
    ReadLn(T, x);
    if x > Tabela[KjeNajmanjsi] then begin
      Tabela[KjeNajmanjsi] := x;
      for i := 1 to N do
        if Tabela[i] < Tabela[KjeNajmanjsi] then KjeNajmanjsi := i;
    end; {if}
  end; {while}
  Close(T);
  for i := 1 to N do WriteLn(Tabela[i]:0:2);
end. {RadarZNeurejenoTabelo}
```

Če bi naloga zahtevala največjih n meritev za nek večji n, ne pa le $n = 20$, bi bilo koristno namesto urejene tabele uporabiti kakšno od podatkovnih struktur za prioritetno vrsto, npr. dvojiško kopico. To bi bilo podobno rešitvi z neurejeno tabelo, le da bi imeli v primerih, ko med n največjih pride neka nova meritev, le $O(\lg n)$ dela, ne pa $O(n)$.

V vsakem primeru je za opisane postopke najhujši tak scenarij, pri katerem pride ob vsaki novi meritvi do spremembe v množici n največjih meritev (na primer: če so meritve v vhodni datoteki že urejene naraščajoče). (Kajti v primerih, ko preberemo novo meritev, pa vidimo, da je manjša od n-te doslej največje, ni treba z njo narediti ničesar več, tako da imamo s tako meritvijo le $O(1)$ dela, neodvisno od n.) Če je v vhodni datoteki m meritev, imata prikazani rešitvi v takem najslabšem primeru časovno zahtevnost $O(mn)$, rešitev s kopico pa $O(m \lg n)$. Če bi si lahko privoščili prebrati vse meritve naenkrat v pomnilnik, bi jih lahko tam uredili in tako videli, katere so največje, vendar pa bi urejanje zahtevalo $O(m \lg m)$ časa, kar torej ni nič bolje od rešitve s kopico (saj je m

večji od n, verjetno celo precej večji); pač pa bi lahko (če bi imeli vse meritve v pomnilniku) uporabili znani postopek z mediano median, ki bi znal izbrati n-ti največji element v času $O(m)$, ne glede na n (glej npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 10.3 v prvi izdaji, 9.3 v drugi).

Na srečo pa, če so meritve naključno premešane, do sprememb v množici n največjih doslej prebranih meritev prihaja precej bolj poredko: čim več meritev smo že prebrali, tem manjša je verjetnost, da bo naslednja meritev prišla med n največjih; zato prihaja do sprememb med n največjimi vse bolj poredko in pri večini meritev bomo porabili le $O(1)$ časa za vsako meritev. Na primer: recimo, da so naše meritve naključne spremenljivke, porazdeljene neodvisno in enakomerno na intervalu $[0, 1]$. Recimo, da smo prebrali že m meritev (in $m \geq n$); naj bo $X_{(n)}$ n-ta največja med njimi. Potem za vsak $x \in [0, 1]$ velja

$$P(X_{(n)} < x) = \textstyle\sum_{k=0}^{n-1} P(\text{točno } k \text{ meritev je } \geq x) = \sum_{k=0}^{n-1} \binom{m}{k}(1-x)^k x^{m-k}.$$

Označimo naslednjo prebrano meritev z X; ker je porazdeljena tako kot ostale, je njena gostota verjetnosti kar $f_X(x) = 1$ za $x \in [0, 1]$ in $f_X(x) = 0$ drugod. Zato je

$$\begin{aligned} P(X_{(n)} < X) &= \textstyle\int_0^1 dx P(X_{(n)} < x) f_X(x) \\ &= \textstyle\int_0^1 dx \sum_{k=0}^{n-1} \binom{m}{k}(1-x)^k x^{m-k} \\ &= \textstyle\sum_{k=0}^{n-1} \binom{m}{k} \int_0^1 (1-x)^k x^{m-k} dx. \end{aligned}$$

Zadnji integral je poseben primer funkcije beta; pokazati je mogoče, da je enak $k!(m-k)!/(m+1)!$. Tako dobimo

$$\begin{aligned} P(X_{(n)} < X) &= \textstyle\sum_{k=0}^{n-1} m!/(k!(m-k)!) \cdot k!(m-k)!/(m+1)! \\ &= \textstyle\sum_{k=0}^{n-1} 1/(m+1) = n/(m+1). \end{aligned}$$

Ta verjetnost je torej res vse manjša, čim več meritev smo prebrali (čim večji je m). Pričakovano število sprememb med n največjimi števili je zato do časa, ko bomo prebrali M števil, enako $\sum_{m=n}^{M-1} n/(m+1)$. Pri tej vsoti si lahko pomagamo s harmoničnimi števili: $H_k = \sum_{i=1}^{k} 1/i$, za katera je znano, da je $H_k = \ln k + \gamma + 1/(2k) + O(k^{-2})$, konstanta γ pa je približno 0,577. Naša vsota je zato

$$\textstyle\sum_{m=n}^{M-1} n/(m+1) = n \sum_{m=n+1}^{M} 1/m = n(H_M - H_{n+1}) \approx n \ln(M/n).$$

Torej lahko pričakujemo, da se pri branju M meritev spremembe med največjimi n meritvami zgodijo le v približno $n \ln(M/n)$ primerih. To pa ni le precej manjše od M, ampak tudi narašča precej počasneje.

V gornjem odstavku smo razmišljali o primeru, ko prihajajo meritve iz verjetnostne porazdelitve, porazdeljene enakomerno na $[0, 1]$. Vendar, če meritve pretransformiramo s poljubno strogo naraščajočo funkcijo, je jasno, da do spremembe med največjimi n meritvami zdaj prihaja v natanko istih primerih kot prej (saj če je bila ena meritev npr. manjša od druge pred transformacijo, bo po njej tudi, če je uporabljena funkcija res strogo naraščajoča). Torej, če prihajajo naše meritve iz neke porazdelitve X in je verjetnostna funkcija te porazdelitve, torej $F_X(x) := P(X < x)$, strogo naraščajoča, lahko meritve poženemo skozi funkcijo F_X in dobimo vrednosti, porazdeljene enakomerno po intervalu $[0, 1]$: res, kajti $P(F_X(X) < y) = P(X < F_X^{-1}(y)) = F_X(F_X^{-1}(y)) = y$ (prvi enačaj velja, ker je F_X strogo naraščajoča, drugi velja po definiciji funkcije F_X, tretji pa zaradi definicije inverza). Zato lahko razmislek iz prejšnjega odstavka uporabimo tudi v tem primeru. Dobljena posplošitev pride prav, če so merive na primer porazdeljene normalno (Gaussova porazdelitev), kar je v praksi najbrž bližje resnici kot pa začetna predpostavka, da so porazdeljene enakomerno po nekem intervalu.

REŠITVE NALOG ZA DRUGO SKUPINO

R2003.2.1 Križanka

Naloga: str. 3

Križanko bomo pregledovali po vrsticah od zgoraj navzdol, vsako vrstico od leve proti desni, in pri vsakem polju preverili, če se tu začenja nova beseda. Vodoravna beseda se začenja, če na trenutnem polju in na tistem desno ob njem ni zvezdice, na tistem levo ob trenutnem pa je zvezdica. Podobno je za navpične besede, le da gledamo poleg trenutnega polja še tisto nad in tisto pod njim. Primere, ko je trenutno polje na robu križanke, bi morali obravnavati posebej; vodoravna beseda se lahko začne na levem robu križanke, ne pa na desnem (ker potem ne bi bila dolga vsaj dve črki), podobno pa velja tudi na navpične besede. Opazimo lahko, da je učinek tega pravila tak, kot da bi bila zunaj križanke tudi polja, na njih pa same zvezdice. To upošteva spodnji podprogram Zvezdica in nam s tem malo poenostavi preverjanje, ali se na trenutnem polju začne nova beseda. Kakorkoli že, če moramo potem neko besedo tudi res izpisati, se moramo le premikati od trenutnega polja v pravi smeri (desno za vodoravne besede, dol za navpične) in izpisovati črke, dokler ne naletimo na zvezdico (ali na rob križanke, vendar za to poskrbi že podprogram Zvezdica).

```
const Visina = 5; Sirina = 10;
type KrizankaT = array [1..Visina, 1..Sirina] of char;

procedure IzpisiBesede(var Krizanka: KrizankaT);

  { Delali se bomo, kot da so zunaj križanke same zvezdice. }
  function Zvezdica(i, j: integer): boolean;
  begin
    if (i < 1) or (j < 1) or (i > Visina) or (j > Sirina)
    then Zvezdica := true else Zvezdica := Krizanka[i, j] = '*';
  end; {Zvezdica}

var i, j, k, Stevilka: integer; Vod, Nav: boolean;
begin
  Stevilka := 0;
  for i := 1 to Visina do for j := 1 to Sirina do
    if not Zvezdica(i, j) then begin

      { Preverimo, če se v tem polju začenja kakšna beseda. }
      Vod := Zvezdica(i, j − 1) and not Zvezdica(i, j + 1);
      Nav := Zvezdica(i − 1, j) and not Zvezdica(i + 1, j);
      if not (Vod or Nav) then continue;
      Stevilka := Stevilka + 1;

      { Izpišimo vodoravno besedo. }
      Write(Stevilka, ', vodoravno: '); k := j;
```

```
      if not Vod then Write('-')
      else while not Zvezdica(i, k) do
        begin Write(Krizanka[i, k]); k := k +1  end;

      { Izpišimo navpično besedo. }
      WriteLn; Write(Stevilka, ', navpično: '); k := i;
      if not Nav then Write('-')
      else while not Zvezdica(k, j) do
        begin Write(Krizanka[k, j]); k := k + 1 end;
      WriteLn;

    end; {if}
end; {IzpisiBesede}

const Krizanka: KrizankaT = (
  'LOREM*IP*S',
  'UM*DOL*ORS',
  'ITAMET*CON',
  'SETE*TUR*S',
  'ADI*PSCING');
begin
  IzpisiBesede(Krizanka);
end.
```

R2003.2.2 Števke

Naloga: str. 4

Naivna rešitev bi bila, da bi preprosto našteli vsa števila od M do N, pri vsakem pogledali, katera je zadnja neničelna števka (to lahko naredimo tako, da ga delimo z deset, dokler ni njegov ostanek po deljenju z deset različen od 0; ta ostanek je ravno zadnja neničelna števka), ter povečali ustrezno celico tabele Rezultat. Ta rešitev je spodaj v podprogramu Stevke2. Vendar pa, kot pravi že besedilo naloge, sta lahko M in N velika in bi takšna rešitev tekla predolgo.

Lahko pa bi razmišljali takole: če gledamo po deset zaporednih števil, $10k$, $10k + 1, 10k + 2, \ldots, 10k + 9$, je za vsako števko od 1 do 9 tu natanko eno število, ki ima to števko kot zadnjo števko. Če je torej med M in N veliko takih deseteric števil, ni nobene potrebe, da bi jih vse naštevali posebej; lahko preprosto izračunamo, koliko jih je, in ustrezno povečamo vse celice tabele Rezultat. S števili oblike $10k$ pa je tako, da je njihova zadnja števka 0, tako da se jim zadnja *neničelna* števka nič ne spremeni, če vsa ta števila delimo z deset. Tako lahko torej namesto števil $10k, 10k + 10, 10k + 20, \ldots$ gledamo kar števila $k, k + 1, k + 2, \ldots$.

Spodnji podprogram Stevke najprej poveča M in zmanjša N do prvega večkratnika števila 10. Števila, ki smo jih zaradi tega preskočili, imajo vsa zadnjo števko različno od 0 in jih torej lahko upoštevamo tako, da povečamo ustrezno celico tabele Rezultat za 1.

Ko sta enkrat M in N večkratnika števila 10, vemo, da je med njima $(N - M)/10$ skupin po deset števil (to so števila od M do vključno $N - 1$), za povrhu pa še število N. Kot smo ugotovili zgoraj, lahko izmed teh števil vsa, ki niso večkratniki 10, upoštevamo v rezultatih tako, da vse celice tabele Rezultat povečamo za $(N - M)/10$. Števila $M, M + 10, M + 20, \ldots, N$ pa lahko vsa delimo z 10 in jih obdelamo tako, da kličemo Rezultat s parametroma $M \operatorname{div} 10$ in $N \operatorname{div} 10$.

```
type RezultatT = array [1..9] of integer;

procedure Stevke(M, N: integer; var Rezultat: RezultatT);
var i, d: integer; R: RezultatT;
begin
  for i := 1 to 9 do Rezultat[i] := 0;
  while (M mod 10 <> 0) and (M <= N) do
    begin Rezultat[M mod 10] := Rezultat[M mod 10] + 1; M := M + 1 end;
  while (N mod 10 <> 0) and (M <= N) do
    begin Rezultat[N mod 10] := Rezultat[N mod 10] + 1; N := N − 1 end;
  { Zdaj sta M in N večkratnika 10. Naj bo d := (N − M)/10. Na intervalu M..N − 1
    se torej pojavi d števil z zadnjo števko i, in to za vsak i od 0 do 9. Za i = 1, . . . , 9
    jih lahko torej kar prištejemo v Rezultat, za i = 0 pa imajo tista števila,
    pa tudi N sam, isto zadnjo neničelno števko, kot če bi jih vsa delili z 10,
    to pa nam da interval (M div 10)..(N div 10). }
  if M <= N then begin
    Stevke(M div 10, N div 10, R);
    for i := 1 to 9 do Rezultat[i] := Rezultat[i] + R[i];
    d := (N − M) div 10;
    for i := 1 to 9 do Rezultat[i] := Rezultat[i] + d;
  end; {if}
end; {Stevke}

procedure Stevke2(M, N: integer; var Rezultat: RezultatT);
var i, j: integer;
begin
  for i := 1 to 9 do Rezultat[i] := 0;
  for i := M to N do begin
    j := i; while j mod 10 = 0 do j := j div 10;
    Rezultat[j mod 10] := Rezultat[j mod 10] + 1;
  end; {for}
end; {Stevke2}

var M, N, i: integer; R: RezultatT;
begin
  ReadLn(M, N); Stevke(M, N, R);
  for i := 1 to 9 do if R[i] > 0 then Write(i, ':', R[i], ' ');
  WriteLn; Stevke2(M, N, R);
  for i := 1 to 9 do if R[i] > 0 then Write(i, ':', R[i], ' ');
  WriteLn; { Upajmo, da se rezultata ujemata. }
end.
```

R2003.2.3 Različnost nizov

Naloga: str. 4

S premikanjem znakov, ki je zastonj, lahko poljubno spremenimo vrstni red znakov v nizu, ne moremo pa brisati odvečnih pojavitev neke črke ali dodajati primerkov črke, ki se je dotlej pojavljala v premalo izvodih. Seveda ne bi imelo smisla, če bi neko črko najprej dodali in kasneje zbrisali ali pa obratno. Najcenejše zaporedje operacij bo zato tisto, ki le doda manjkajoče ali zbriše odvečne črke in jih nato preuredi v pravi vrstni red.

Zato za vsako črko abecede poglejmo, kolikokrat se pojavlja v prvem in kolikokrat v drugem nizu. Če se pojavlja v prvem večkrat kot v drugem, bo treba nekaj pojavitev zbrisati, če v drugem večkrat kot v prvem, pa bo treba nekaj pojavitev dodati. V vsakem primeru je cena tega kar absolutna vrednost razlike števila pojavitev. Vsota teh cen nam zadostuje, da prvi niz predelamo v nekaj, kar ima enake črke kot drugi niz (in v enakem številu izvodov), nato pa jih moramo le še preurediti, kar pa je zastonj.

```
function Razdalja(S, T: string): integer;
var NS, NT: array ['a'..'z'] of integer; c: char; i: integer;
begin
  for c := 'a' to 'z' do begin NS[c] := 0; NT[c] := 0 end;
  for i := 1 to Length(S) do NS[S[i]] := NS[S[i]] + 1;
  for i := 1 to Length(T) do NT[T[i]] := NT[T[i]] + 1;
  i := 0; for c := 'a' to 'z' do i := i + Abs(NS[c] − NT[c]);
  Razdalja := i;
end; {Razdalja}
```

R2003.2.4 Pošiljanje sporočil

Naloga: str. 5

Ker so naslovi (številke IP) enolični, lahko računalnike uredimo po naslovih v urejen seznam. Seznam nato predelamo tako, da je računalnik, ki je prvi prejel sporočilo (od uporabnika), tudi prvi v seznamu (računalnike z nižjim IP-jem dajmo na konec seznama). Nato ta računalnik pošlje sporočilo do naslednjega v seznamu. Nato 1. in 2. računalnik pošljeta sporočilo do 3. in 4. Vsi skupaj pošljejo sporočilo do 5., 6., 7. in 8. Vsi računalniki pošiljajo sporočila, dokler ne dosežejo zadnjega v seznamu.

Primer za 9 računalnikov (oštevilčeni z 0–8):

1. sekunda:	$0 \to 1$
2. sekunda:	$0 \to 2$, $1 \to 3$
3. sekunda:	$0 \to 4$, $1 \to 5$, $2 \to 6$, $3 \to 7$
4. sekunda:	$0 \to 8$

Potrebovali smo torej 4 sekunde. Pomembno je, da uporabljajo vsi računalniki enak seznam naslovov, česar pa ni težko zagotoviti, saj vsi poznajo vse naslove, s prej opisanim urejanjem pa lahko tudi vsak razporedi naslove v enak vrstni red.

Kako bi to delovalo v splošnem? Če je v neki sekundi m računalnikov poslalo sporočilo, bodo v naslednji sekundi poslali sporočilo vsi ti računalniki, pa tudi vsi tisti, ki so v prejšnji sekundi šele prejeli obvestilo: to pa je ravno tistih m prejemnikov, ki so jim pošiljatelji v prejšnji sekundi poslali sporočila. V naslednji sekundi bo torej pošiljalo sporočila $2m$ računalnikov. Ta razmislek nam pove, da v k-ti sekundi pošilja sporočila 2^{k-1} računalnikov. Če računalnike oštevilčimo z indeksi od 0 naprej, vidimo, da v k-ti sekundi pošiljajo računalniki $0, \ldots, 2^{k-1} - 1$ sporočila računalnikom $2^{k-1}, \ldots, 2^k - 1$, in sicer tako, da vsak računalnik i pošlje obvestilo računalniku $i + 2^{k-1}$.

Ko torej naš računalnik prvič dobi obvestilo, mora vedeti le, v kateri sekundi pošiljanja se je to zgodilo, pa bo lahko ugotovil, koga mora začeti sam obveščati v naslednji sekundi. Dovolj je že, če računalnik ugotovi svoj indeks, kar lahko stori tako, da poišče svoj naslov v primerno urejeni tabeli naslovov. Ko najde svoj indeks i, lahko poišče tak k, za katerega je $2^{k-1} \leq i < 2^k$; za tega potem velja, da je naš računalnik dobil obvestilo v k-ti sekundi. (Lahko pa bi ta k prenašali tudi skupaj s sporočilom. Kasneje, ko bi naš računalnik pošiljal sporočila drugim, bi k pred vsakim pošiljanjem povečal za 1.) Zdaj torej vemo, s kakšnim k-jem moramo nadaljevati, ko bomo sami pošiljali sporočila. Svoje prvo sporočilo bomo poslali v okviru $(k+1)$-ve sekunde, tako da ga bomo morali poslati računalniku $i+2^k$. Sekundo zatem bomo obvestili računalnik $i+2^{k+1}$ in tako naprej. Ko nam ti indeksi padejo čez število računalnikov, moramo seveda nehati; takrat vemo, da bodo najkasneje do konca trenutne sekunde obveščeni že vsi računalniki.

```
type SporociloT = record
                    PrvoSporocilo: boolean;
                    { Naslov prvega računalnika (tistega, ki je prvi dobil
                      sporočilo od uporabnika). Tega potrebujemo, da lahko
                      vsak računalnik sestavi enak vrstni red naslovov in to
                      takega, v katerem je prvi računalnik na začetku tabele. }
                    NaslovPrvega: NaslovT;
                  end; {SporociloT}

procedure ObPrejemuSporocila(S: SporociloT; Posiljatelj: NaslovT);

  procedure PrerazporediNaslove(var Naslovi: NasloviT; NaslovPrvega: NaslovT);
  var Naslovi2: NasloviT; i, j: integer;
  begin
    { Naslove, ki so manjši kot NaslovPrvega, odložimo v pomožno tabelo. }
    i := 0; while Naslovi[i + 1] <> NaslovPrvega do
      begin i := i + 1; Naslovi2[i] := Naslovi[i] end;
    { Naslove, ki so večji ali enaki NaslovPrvega, premaknemo na začetek tabele. }
    for j := i + 1 do StRacunalnikov do Naslovi[j − i] := Naslovi[j];
    { Na konec tabele zapišemo naslove, manjše od NaslovPrvega. }
    for j := 1 to i do Naslovi[StRacunalnikov − i + j] := Naslovi2[j];
  end; {PrerazporediNaslove}

  function IndeksNaslovaVTabeli(Naslov: NaslovT; var Naslovi: NasloviT): integer;
  var i: integer;
  begin
    i := 1; while Naslovi[i] <> Naslov do i := i + 1;
    IndeksNaslovaVTabeli := i;
  end; {IndeksNaslovaVTabeli}

var
  Naslovi: NasloviT;
  Indeks, k: integer;
begin
  { Če je tole uporabnikovo sporočilo, vemo, da je
    naš računalnik prvi, ki je bil obveščen. }
  if S.PrvoSporocilo then begin
    S.PrvoSporocilo := false;
    S.NaslovPrvega := NasNaslov;
  end; {if}

  { Spomnimo se, da nam podprogram NasloviVsehRacunalnikov
    vrne naslove, urejene naraščajoče. Torej bo vsak računalnik
    po klicu te funkcije videl enak vrstni red. }
  NasloviVsehRacunalnikov(Naslovi);

  { PrerazporediNaslove premakne naslove, manjše od S.NaslovPrvega, na konec
    tabele, drugače pa njihovega medsebojnega vrstnega reda ne spreminja. }
  PrerazporediNaslove(Naslovi, S.NaslovPrvega);
```

```
  { Kateri po vrsti (0..StRacunalnikov − 1) v tabeli naslovov je naš naslov? }
  Indeks := IndeksNaslovaVTabeli(NasNaslov, Naslovi) − 1;

  { Izračunajmo k, torej v kateri sekundi smo mi prejeli tole sporočilo. }
  k := 0; while not (Indeks < 1 shl k) do k := k + 1;

  { V naslednji, torej (k + 1)-vi sekundi, bomo morali
    obvestiti računalnik Indeks + 1 shl k. }
  while Indeks + 1 shl k < StRacunalnikov do begin
    k := k + 1; { Zdaj smo v novi sekundi. }
    { V k-ti sekundi obvestimo računalnik Indeks + 1 shl (k − 1).
      Vendar pa ne pozabimo, da mi štejemo indekse od 0 naprej,
      v tabeli Naslovi pa so od 1 naprej. }
    PosljiSporocilo(S, Naslovi[Indeks + 1 shl (k − 1) + 1]);
  end; {while}

end; {ObPrejemuSporocila}
```

REŠITVE NALOG ZA TRETJO SKUPINO

R2003.3.1 Napadalne kraljice

Naloga: str. 8

Za začetek si oglejmo nekaj rezultatov, dobljenih z metodo „razveji in omeji“ (*branch and bound*) in s simuliranim ohlajanjem.

Najmanjše število kraljic, potrebnih, da napademo vsa polja šahovnice $n \times n$:[1]

n	1	2	3	4	5	6	7	8	9	10
št. kraljic	1	1	1	2	3	3	4	5	5	5

n	11	12	13	14	15	16	17	18	19	20	21
št. kraljic	5	6	7	≤ 8	≤ 9	≤ 9	≤ 9	≤ 9	≤ 10	≤ 11	≤ 11

Na šahovnici $n \times n$ se da s k kraljicami napasti vsaj toliko polj:

[1] Glej tudi: *The On-Line Encyclopedia of Integer Sequences*, A075458 (minimalno potrebno število kraljic, ki napadejo vsa polja), A002563 (z dodatnim pogojem, da ne smejo napadati druga druge); Matthew D. Kearse, Peter B. Gibbons: *Computational methods and new results for chessboard problems*, Australasian Journal of Combinatorics, 23:253–284, March 2001 (tudi: Tech. Rept. CDMTCS-133, Centre for Disc. Math. and Theoretical Comp. Sci., CS Dept., Univ. of Auckland, NZ, May 2000); Patric R. J. Östergård, William Douglas Weakley: *Values of domination numbers in the queen's graph*, The Electronic Journal of Combinatorics, 8(1):R29, 2001. Naj bo a_n minimalno število kraljic, potrebnih, da napademo vsa polja šahovnice n×n. Za vse šahovnice $n \times n$ do $n = 122$ (in še za nekaj večjih n) je znano, da je $a_n \in \{\lceil n/2 \rceil, \lceil n/2 \rceil + 1\}$ (edini izjemi sta $n = 3$ in $n = 11$, kjer je dovolj že $\lfloor n/2 \rfloor$ kraljic); za 53 izmed teh n-jev je znano že tudi, kakšna je res prava vrednost a_n (največkrat je to $\lceil n/2 \rceil$, med drugim tudi za vse n-je, ki so oblike $4t + 1$, vse do vključno $n = 129$). Za vsak n pa velja $a_n \geq \lfloor n/2 \rfloor$.

Za šahovnice do vključno $n = 13$ smo se s pregledom vseh možnih razporedov prepričali, da z manj kot v gornji tabeli navedenim številom kraljic ne gre; za nadaljnje n pa takega preizkusa nismo naredili, saj bi trajal predolgo (zato so v gornji tabeli znaki $\leq$). Vendar pa iz članka Östergårda in Weakleya sledi, da so vse tudi vse nadaljnje meje, navedene v gornji tabeli, dejansko tesne (z manj kraljicami ne moremo napasti vseh polj), le pri $n = 20$ še ni znano, če ni morebiti dovolj že tudi samo deset kraljic. Vendar pa, če razpored z desetimi kraljicami za šahovnico 20×20 obstaja, ga je presneto težko najti, saj ga tudi po večdnevnem iskanju s simuliranim ohlajanjem nismo našli. Ker pri drugih tu preizkušenih n večinoma ni tako težko priti do razporeda z minimalnim številom kraljic, se nagibamo k mnenju, da je pri $n = 20$ najbrž vendarle potrebnih enajst kraljic.

n	$k=$	5	6	7	8	9	10	11
12		134	144					
13		153	165	169				
14		172	186	194	196			
15		193	209	221	224	225		
16		212	231	242	252	256		
17		233	255	269	282	289		
18		252	277	294	310	324		
19		273	301	321	341	357	361	
20		292	323	348	370	385	398	400
21		313	347	377	401	419	435	441

Rezultati, ki jih je bilo malo težje najti[2] (ostale rezultate gornje tabele najde simulirano ohlajanje zelo hitro): (14, 7, 194); (15, 7, 221); (17, 8, 282); (17, 9, 289); (18, 9, 324); (19, 9, 357); (19, 10, 361); (20, 10, 398); (20, 11, 400); (21, 10, 435); (21, 11, 441) (že (21, 11, 439) se ne najde zelo hitro; pač pa ni težko dobiti (21, 12, 441)).

Ker učinkovitega algoritma, ki bi pri danem številu kraljic in velikosti šahovnice zagotovljeno poiskal najboljši razpored, najbrž sploh ni, nam preostane le to, da poskušamo s kakšnimi hevristikami preizkusiti veliko razporedov, se osredotočati na čim bolj obetavne razporede in končno odkriti nek čim boljši razpored.

Preprost postopek bi bil kar ta, da kraljice na šahovnico **razporedimo naključno**; pazimo le na to, da jih ne bi po več pristalo na istem polju. Tak naključni razpored sicer običajno najbrž ne bo pretirano dober, ker pa lahko tak razpored sestavimo in ocenimo zelo hitro, si lahko privoščimo preizkusiti veliko naključnih razporedov. Na koncu izpišimo najboljšega od vseh preizkušenih razporedov; z malo sreče ta vendarle ne bo tako zelo slab. Seveda pa je z naključnim razporejanjem težko priti do res vrhunskih rezultatov; na primer, pri 6 kraljicah in šahovnici 13×13 je mogoče napasti največ 165 polj, vendar to doseže le 72 razporedov, vseh pa je $N = \binom{13 \cdot 13}{6} = 29\,581\,203\,652$. Verjetnost, da je nek naključni razpored slabši, je torej $1 - 72/N$; verjetnost, da je slabši vsak od c preizkušenih razporedov, je zato $(1 - 72/N)^c$; verjetnost, da je vsaj eden najboljši, je torej $1 - (1 - 72/N)^c$. Pri $c = 10^6$ poskusih nam ta formula pove, da imamo le 0,24 % možnosti, da bi odkrili enega od najboljših razporedov (v povprečju pa lahko, kot se izkaže, pričakujemo, da bo imel najboljši izmed milijona preizkušenih razporedov napadenih okoli 157 polj). Če bi hoteli naše možnosti dvigniti na 90 %, bi morali preizkusiti skoraj milijardo razporedov.

Po tistem, ko smo s preizkušanjem naključnih razporedov našli nek ne-tako-zelo-slab razpored, ga lahko poskušamo še malo izboljšati. Lahko na primer poskusimo na vse možne načine premakniti dve kraljici (ostalih $k-2$ pa pustimo pri miru). (Tovrstni lokalni optimizaciji včasih pravijo **2-opt**.) Dve kraljici izmed k si lahko izberemo na $\binom{k}{2} = k(k-1)/2$ načinov, njuna položaja (pri čemer nočemo, da bi bili na istih poljih kot ostale kraljice) pa na $\binom{n^2-(k-2)}{2}$ načinov. Vsega skupaj moramo torej preizkusiti približno $k^2 n^4/4$ novih razporedov; pri vrednostih k in n, s kakršnimi imamo opravka mi, je to še sprejemljivo, čeprav ne več bliskovito hitro. Če je kakšen od teh novih razporedov boljši od začetnega, vzemimo najboljšega izmed novih razporedov in na njem

[2] Npr. ker je moralo simulirano ohlajanje teči nekaj deset sekund, preden je našlo tak razpored. Lahko pa praktično vedno pri istih n in k hitro najdemo razpored z le malo manj napadenimi polji.

isti postopek ponovimo: mogoče se da dobiti še kaj boljšega, če premikamo zdaj še kakšni drugi kraljici. Pri naših poskusih tega postopka običajno ni bilo treba izvesti več kot trikrat ali štirikrat. Izkaže se, da s tem pridemo do že kar precej dobrih rezultatov, sploh če je kraljic bolj malo (npr. le pet ali šest).

	Naključno razporejanje								Naključno razporejanje + 2-opt							
n $k=$	5	6	7	8	9	10	11	12	5	6	7	8	9	10	11	12
12	2	6							0	2						
13	0	8	7						0	0	4					
14	7	9	9	6					0	1	8	0				
15	8	13	14	10	6				0	3	5	2	2			
16	13	18	14	16	12				0	0	2	2	2			
17	15	15	18	17	17				0	0	1	11	11			
18	17	18	17	20	25				0	9	0	0	13			
19	19	24	20	25	29	24			0	3	10	3	4	6		
20	19	32	21	29	30	30	23		0	1	2	1	17	16	8	
21	24	24	31	35	36	33	30	23	0	3	2	6	6	7	6	5

Za vsak par (n, k) smo preizkusili milijon naključnih razporedov, kar je za vsak (n, k) vzelo povprečno po deset sekund (na računalniku z 800-megaherčnim procesorjem). Levi del tabele kaže, za koliko je bil najboljši med temi naključnimi razporedi slabši od najboljšega znanega razporeda. Potem smo na najboljšem naključnem razporedu poskusili izvajati še 2-opt, dokler se je s tem razpored kaj izboljševal; desni del tabele kaže, koliko je bil končni razpored slabši od najboljšega znanega. (Ti rezultati so seveda odvisni od vrednosti, ki jih je vračal generator naključnih števil. Če bi pognali program še enkrat z drugačnim semenom, bi mogoče odkrili še kak boljši razpored.)

Sestavljanja razporeda pa se lahko lotimo tudi drugače. Poskusimo postavljati kraljice na šahovnico eno za drugo; ker si želimo, da bi bilo napadenih čim več polj, lahko poskusimo vsako naslednjo postaviti na tako mesto, da bo napadla čim več takih polj, ki jih prejšnje kraljice niso napadale. Tak **požrešen postopek** je zelo hiter in tudi ne daje slabih rezultatov.

Požrešni postopek lahko dopolnimo s **sestopanjem**. Po tistem, ko razpostavimo vse kraljice in ocenimo dobljeni razpored, lahko poskusimo zadnjo kraljico vzeti s šahovnice in nato predzadnjo kraljico postaviti na kakšno drugo polje; potem bi spet vzeli najboljši položaj zadnje kraljice. Ko preizkusimo več položajev predzadnje kraljice, poskusimo spremeniti tudi položaj predpredzadnje kraljice in nato spet preizkusimo več položajev predzadnje kraljice; itd. Ta postopek je pravzaprav rekurzija, ki poskusi dodati trenutno kraljico na razna mesta na šahovnici in pri vsakem izvede še en rekurzivni klic, da bi razmestila še preostale kraljice.

Če hočemo za vsako kraljico preizkusiti p položajev, kraljic pa je k, bomo vsega skupaj preizkusili p^k razporedov. Če imamo na primer deset kraljic, je 3^{10} še čisto sprejemljivo, 10^{10} pa najbrž že ne več. Paziti moramo torej, da ne vzamemo prevelikega p. Vsako kraljico poskusimo postaviti le na nekaj najobetavnejših položajev. To, kako obetaven je nek položaj, lahko ocenjujemo enako kot pri požrešnem postopku: položaj nove kraljice je tem bolj obetaven, čim več doslej nenapadenih polj lahko tja postavljena kraljica napade.

Če sta šahovnica in število kraljic dovolj majhni, lahko pri tej rekurzivni rešitvi preizkusimo tudi vse možne položaje naslednje kraljice in s tem tudi vse razporede cele skupine k kraljic. S tem se lahko prepričamo, da pri šahovnici 8×8 z manj kot petimi kraljicami ne moremo napasti vseh polj, pa da pri 12×12 s petimi kraljicami ne moremo napasti več kot 134 od 144 polj ipd.

		Požrešni algoritem ($p = 1$)								Rekurzija ($p = 3$ in $p = \lceil (10^6)^{1/k} \rceil$)							
n	$k =$	5	6	7	8	9	10	11	12	5	6	7	8	9	10	11	12
12		1	4							–	2						
13		2	5	4						–	–	–					
14		–	1	2	–					–	–	2	–				
15		2	4	2	1	–				–	–	3	–	–			
16		–	1	–	1	1				–	–	–	–	–			
17		2	4	2	3	3				–	–	–	1	–			
18		–	1	–	1	7				–	–	–	$1^\star$	4			
19		2	4	2	4	8	4			–	–	–	–	4	2		
20		–	1	4	8	8	9	5		–	–	$2^\star$	$3^\star$	$1^\star$	4	–	
21		2	3	6	7	6	9	7	2	–	–	–	–	–	4	2	–

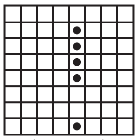
Ta tabela kaže, za koliko se razlikujejo od najboljših znanih rezultatov rešitve, ki smo jih našli s požrešnim algoritmom (levi del tabele) in z rekurzijo (desni del). (Požrešni algoritem je pravzaprav poseben primer rekurzivnega: kot da bi vzeli $p = 1$.) Pri rekurziji smo p izbirali na dva načina: vedno smo poskusili vzeti $p = 3$, kar pomeni, da se rekurzija izteče že po nekaj sekundah; kot drugo možnost pa smo p prilagodili številu kraljic po formuli $p = \lceil (10^6)^{1/k} \rceil$, tako da je število preizkušenih razporedov vedno nekaj čez milijon. Ti dve možnosti sta dali skoraj vedno enako dobre rezultate in zato desna tabela velja za obe; razlikujeta se le v nekaj primerih, ko daje večji p rezultate, enakovredne najboljšim znanim, $p = 3$ pa nekaj slabše. Ti primeri so v tabeli označeni z zvezdico in številka poleg nje se nanaša na rezultat za $p = 3$. Ker so rezultati rekurzije zelo pogosto enakovredni najboljšim znanim razporedom, smo namesto ničel pisali znak $-$, da je neničelne razlike lažje opaziti.

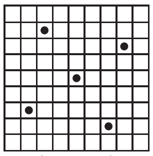
Večjih problemov od tega slednjega pa na ta način najbrž že ne bi več mogli rešiti v sprejemljivem času.[3] Mogoče pa bi rekurzija hitro našla neko precej dobro rešitev in bi jo lahko po nekaj časa preprosto prekinili in izpisali najboljšo dotlej najdeno rešitev.

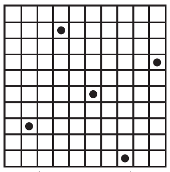
Še en optimizacijski postopek, ki se pri tej nalogi kar dobro odreže, je **simulirano ohlajanje**. Pri tem postopku poskušamo trenutni razpored naključno spremeniti (na primer tako, da spremenimo položaj ene od kraljic). Če je novi razpored boljši, to spremembo obdržimo; če pa bi bil novi razpored slabši, ga z neko verjetnostjo vendarle sprejmemo, z neko verjetnostjo pa ga zavrnemo. To verjetnost običajno izberemo tako, da pada eksponentno z razliko med razporedoma: če napade novi razpored le m' polj, prvotni pa m polj, ga obržimo z verjetnostjo $c^{m-m'}$ za nek $c < 1$, na primer $c = 1/2$. (Če smo blizu kakšne zelo dobre rešitve (npr. napadamo že skoraj vsa polja), lahko c še zmanjšamo, da ne bi program prehitro obupoval in si poslabšal razporeda.) Namen tega pravila za sprejemanje sprememb na slabše je, da bi programu, če se zapleza v nek lokalni optimum, omogočili tudi, da se izvleče iz njega; obenem pa mu hočemo preprečiti, da bi po nemarnosti sprejemal neumne spremembe (zato, če je novi

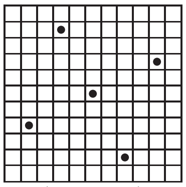
[3]Preizkušanje vseh razporedov 6 kraljic na šahovnici 13×13 (ki jih je dobrih 29,5 milijard) je trajalo na računalniku z dvema 2400-MHz procesorjema približno 13 ur in pol (program je bil dvoniten, tako da sta bila ves čas zasedena oba procesorja). No, s tem smo se vsaj prepričali, da s šestimi kraljicami na tolikšni šahovnici ne moremo napasti več kot 165 polj. Pri tej hitrosti (približno 600 000 razporedov na sekundo) bi trajal pregled vseh razporedov sedmih kraljic na šahovnico 14×14 (ki jih je slaba dva bilijona) skoraj 38 dni.

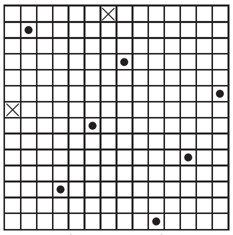
Res pa je, da lahko takšen rekurzivni postopek še precej izboljšamo z raznimi hevristikami, ki poskušajo čim prej odkriti, če je trenutni razpored neobetaven (ne bo vodil do rešitve), tako da se z njim ni treba ukvarjati še naprej. Več takih tehnik opisujeta Kearse in Gibbons (razdelek 3), ki sta z njimi pregledala še nekaj naslednjih šahovnic (do 18×18, za katero sta se na ta način prepričala, da je ni mogoče v celoti napasti z 8 ali manj kraljicami).

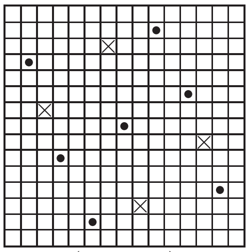

$(8, 5, 64)$

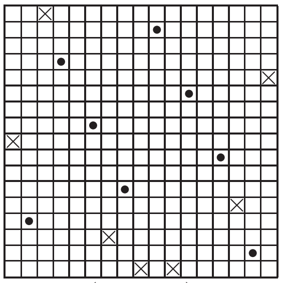

$(9, 5, 81)$

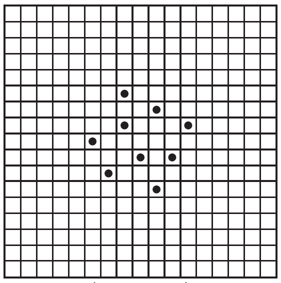

$(10, 5, 100)$

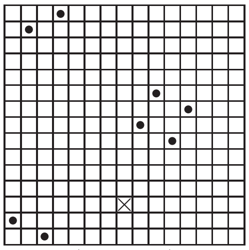

$(11, 5, 121)$

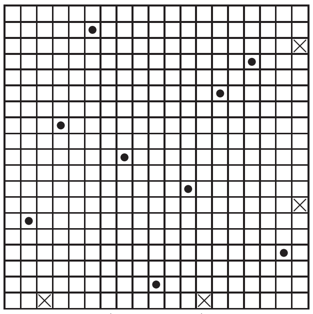

$(14, 7, 194)$

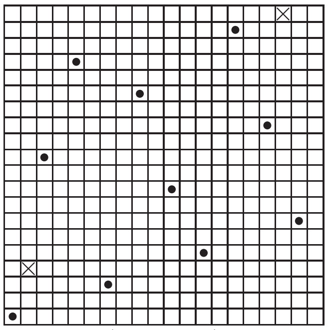

$(15, 7, 221)$

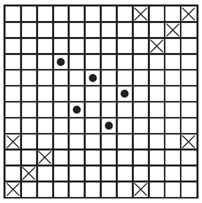

$(17, 8, 282)$

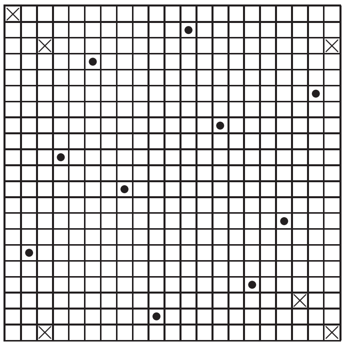

$(17, 9, 289)$

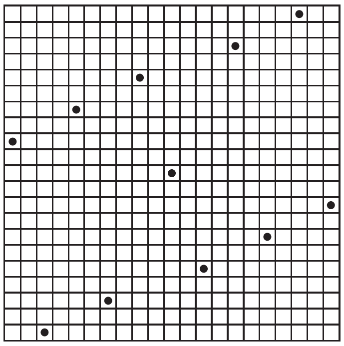

$(15, 8, 224)$


$(19, 9, 357)$


$(20, 10, 398)$


$(12, 5, 134)$


$(21, 10, 435)$


$(21, 11, 441)$

Nekaj dobrih razporedov kraljic na šahovnice. Številke pod vsako šahovnico pomenijo velikost šahovnice (n), število kraljic (k) in število napadenih polj. Črne pike so kraljice, znak $\times$ pa pomeni nenapadeno polje. Še nekaj zanimivih razporedov lahko dobimo, če tu prikazanim dodamo ali odvzamemo kakšno kraljico: $(13, 7, 169)$ in $(12, 6, 144)$ iz $(11, 5, 121)$; $(14, 8, 196)$ iz $(14, 7, 194)$; $(15, 9, 225)$ in $(16, 9, 256)$ iz $(17, 9, 289)$; $(18, 9, 324)$ in $(19, 10, 361)$ iz $(19, 9, 357)$; $(20, 11, 400)$ iz $(20, 10, 398)$.

razpored res precej slabši od prvotnega, bo verjetnost sprejema tako majhna, da ga bomo skoraj gotovo zavrnili). Naključno spreminjanje razporeda pa programu omogoča, da raziskuje prostor možnih razporedov in, upajmo, sčasoma najde predele z obetavnimi razporedi. (Da se program ne bi takoj po premiku na slabše povrnil nazaj v isti razpored, iz katerega je malo prej prišel (saj bi bil to zdaj zanj premik na bolje in se mu načeloma ne bi mogel upreti), si je koristno zadnjih nekaj (npr. zadnjih trideset) razporedov zapomniti in se vanje ne premikati. Temu običajno pravijo, da smo jih razglasili za tabu.) Simulirano ohlajanje je koristno na vsake toliko časa pognati spet od začetka iz nekega naključno sestavljenega razporeda. Dlje ko ga pustimo teči, več je možnosti, da se bo našla kakšna dobra rešitev. Od tu opisanih postopkov je dajalo simulirano ohlajanje še najboljše razporede. Praktično vedno je že po nekaj sekundah našlo kakšen zelo dober razpored; v nekaj primerih je po minuti ali dveh našlo še kaj boljšega, kasneje pa skoraj nikoli nič (razen pri 10 in 11 kraljicah na plošči 21×21), tudi če smo ga pustili teči po pol ure.

Nekaj pa lahko naredimo tudi ročno. Pri naših poskusih smo morali pustiti simulirano ohlajanje teči približno sedem minut, preden je našlo razpored, ki napade vsa polja šahovnice 19×19 z desetimi kraljicami; pač pa je v manj kot minuti našlo razpored, ki z devetimi kraljicami napade 357 od $19^2 = 361$ polj. Izkazalo se je, da so pri tem razporedu nenapadena polja ležala tako, da bi jih lahko vsa štiri napadli z eno samo dodatno kraljico; tako pridemo do razporeda, ki napade vsa polja šahovnice z desetimi kraljicami. Če šahovnico povečamo na 20×20 in dodamo še eno kraljico v kot, pa dobimo tudi razpored, ki z enajstimi kraljicami napade vsa polja šahovnice 20×20.

R2003.3.2 **Smučarji**

Do najboljše uvrstitve nekega tekmovalca v skupnem seštevku ni težko priti. To uvrstitev doseže, če zmaga, obenem pa vsi ostali odstopijo. Po novem ima torej namesto a_i točk v skupnem seštevku $a_i + t_1$ točk, ostali pa toliko, kot so jih imeli prej zadnjo tekmo. Treba je le še pogledati, koliko bi jih s tem prehitel. Ker bomo hoteli to izračunati za vse smučarje, si lahko pomagamo s tem, da najboljša možna končna uvrstitev smučarja, ki je bil prej $(s + 1)$-vi v skupnem seštevku, prav gotovo ni boljša od najboljše možne končne uvrstitve smučarja, ki je bil prej s-ti. Zato se ta uvrstitev le povečuje, ko gledamo vse slabše smučarje; vsega skupaj je torej s tem le $O(n)$ dela. Naloga: str. 10

Če bi dovolili, da se več tekmovalcev uvrsti na isto mesto, bi bilo tudi do najslabše uvrstitve lahko priti: naš tekmovalec naj odstopi, vsi ostali pa naj si delijo prvo mesto. Vendar pa naloga takšnega izida tekme ne dopušča.

Mislimo si tisti izid zadnje tekme, po katerem je naš tekmovalec na najslabšem možnem mestu v skupnem seštevku. V njem lahko po vrsti izvedemo naslednje spremembe, pa se položaj našega tekmovalca v skupnem seštevku gotovo ne bo nič izboljšal:

- Naš tekmovalec lahko odstopi. Zaradi tega dobijo drugi vsaj toliko točk kot prej, naš pa največ toliko kot prej, tako da ga vsi, ki so ga prej v skupnem seštevku prehiteli, prehitijo tudi zdaj. Zato njegova uvrstitev ni nič boljša.

- Odstopijo lahko vsi, ki so bili pred zadnjo tekmo v skupnem seštevku pred našim. Ker naš v zadnji tekmi ne dobi nič točk, bodo vsi ti še vedno pred njim; vsi ostali pa dobijo zaradi te spremembe vsaj toliko točk kot prej in torej tisti, ki bi našega sicer prehiteli, to storijo tudi zdaj.

- Odstopijo lahko vsi, ki našega tekmovalca po tej zadnji tekmi v skupnem seštevku ne prehitijo. Vsi ostali, namreč tisti, ki ga prehitijo, dobijo zaradi tega vsaj toliko točk kot prej in ga zato še vedno prehitijo.
- Označimo zdaj s S_i smučarja, ki je bil v skupnem seštevku pred zadnjo tekmo i mest za našim. Po vseh spremembah, ki smo jih doslej izvedli v izidu zadnje tekme, je ta zdaj tak, da pride do cilja le še nekaj smučarjev S_i za $i > 0$ in vsi ti tudi prehitijo našega smučarja v skupnem seštevku. Recimo, da pridejo na cilj $S_{i_1}, \ldots, S_{i_k}$ (ne nujno v tem vrstnem redu) za neke $1 \leq i_1 < i_2 < \cdots < i_k$. Potem očitno velja $i_1 \geq 1$, $i_2 \geq 2$, itd., $i_k \geq k$. Če bi zdaj smučarja S_{i_j} zamenjali s S_j (za vsak $j = 1, \ldots, k$), bi tudi ta uspel prehiteti našega (saj je imel zaradi $j \leq i_j$ smučar S_j pred zadnjo tekmo v skupnem seštevku vsaj toliko točk kot S_{i_j}).
- Zdaj smo torej prišli do takega izida, pri katerem pridejo na cilj le smučarji $S_1, \ldots, S_k$ in vsi prehitijo našega v skupnem seštevku. Recimo, da za nek i doseže i-to mesto nek smučar S_j, $(i+1)$-vo pa nek smučar S_r za $r > j$. Če bi se zdaj vrstni red teh dveh ravno zamenjal, bi S_r dobil vsaj toliko točk, kot jih je prej (saj je zdaj uvrščen eno mesto više), tako da bi gotovo še vedno prehitel našega; po drugi strani pa bi S_j dobil toliko točk, kot jih je prej S_r, in ker je $r > j$, je imel S_r od prej v skupnem seštevku kvečjemu toliko točk kot S_j (mogoče pa še manj), tako da, če je S_r-ju uvrstitev na $(i+1)$-vo mesto zadostovala za to, da je prehitel našega, bo S_j-ju še toliko lažje.
- S ponavljanjem prejšnje točke lahko preuredimo izid zadnje tekme tako, da pride na prvo mesto smučar S_k, na drugo S_{k-1}, itd., na predzadnje smučar S_2 in na zadnje smučar S_1.

Videli smo: vsak izid zadnje tekme lahko predelamo v izid take oblike, kot ga opisuje zadnja točka, pa se pri tem uvrstitev, kot jo naš tekmovalec doseže po zadnji tekmi, ne bo nič poslabšala. Torej je dovolj, če se pri iskanju najslabše možne uvrstitve v skupnem seštevku omejimo na izide tega tipa.

Za vsakega smučarja S_i lahko ugotovimo, katera je najslabša uvrstitev, pri kateri še dobi dovolj točk, da bo našega prehitel v skupnem seštevku; recimo, da je to uvrstitev u_i. Po drugi strani pri izidu gornje oblike, če pride v cilj k smučarjev, smučar S_i doseže $(k + 1 - i)$-to mesto. Če hočemo, da res prehiti našega, mora torej veljati $k+1-i \leq u_i$ oziroma $k \leq u_i+i-1$. To mora veljati za vse smučarje, torej za vse i od 1 do k. Veljati mora torej $k \leq \min\{u_i+i-1 : i = 1, \ldots, k\}$. Jasno je, da, če pri nekem k to ne velja več, tudi pri nobenem večjem ne bo (ker se leva stran neenačbe le povečuje, desna pa lahko ostane enaka ali pa se celo zmanjša). Čim naletimo na tak k, lahko nehamo z iskanjem. Drugi možni ustavitveni pogoj je to, da nam zmanjka smučarjev; če je bil naš smučar prej v skupnem seštevku s-ti, ga lahko pač na novo prehiti le $n - s$ smučarjev.

Prav nam bo prišlo tudi dejstvo, da tekmovalec, ki od prej v skupnem seštevku za našim zaostaja bolj kot nek drug, potrebuje v zadnji tekmi vsaj tako dobro uvrstitev kot ta, če naj prehiti našega. Z drugimi besedami, $u_i \geq u_{i+1}$.

Časovna zahtevnost tega postopka za posameznega smučarja je $O(n+\lg m)$. $O(\lg m)$ časa potrebujemo, da z bisekcijo poiščemo u_1; če je ta večji od $n - s$, ga lahko postavimo na $n - s$, saj k nikoli ne bo večji od $n - s$ in zato tudi ne bo potrebe po tem, da bi bil kdo na zadnji tekmi uvrščen na slabše mesto od tega. Za vsakega naslednjega smučarja potem ugotovimo u_{i+1} tako, da začnemo pri u_i in ga po potrebi zmanjšujemo, dokler ne dosežemo uvrstitve, s katero lahko S_{i+1} prehiti našega smučarja. Zato je vseh zmanjševanj pri računanju vrednosti u_i lahko največ u_1 (potem pademo na 0, kar pomeni, da smo prišli do smučarjev, ki ne morejo niti z zmago prehiteti našega v skupnem seštevku), ta pa je $\leq n - s = O(n)$. Zato imamo tu tako z iskanjem nadaljnjih vrednosti u_i kot s preverjanjem, če še velja zgoraj omenjena neenačba, le $O(n)$ dela. Ker moramo to ponoviti za vsakega smučarja (s gre od 1 do n), je skupna časovna zahtevnost $O(n^2 + n \lg m)$.

```
program Smucarji;
const MaxM = 10000; MaxN = 10000; MaxT = 100000;
var f: text;
   s, ui, i, r, n, m, Najboljsa: integer;
   t: array [0..MaxM] of integer;
   a: array [1..MaxN + 1] of integer;
begin
  Assign(f, 'smucarji.in'); Reset(f);
  { Na konec tabele a dajmo kot stražarja enega čisto brezupnega smučarja, ki ne more
    prehiteti nikogar. Na začetek tabele t dajmo kot stražarja odlično uvrstitev,
    s katero nas lahko prehiti vsakdo, celo tisti brezupni smučar. }
  ReadLn(f, m); t[0] := 2 * MaxT + 2; for i := 1 to m do ReadLn(f, t[i]);
  ReadLn(f, n); a[n] := – MaxT – 1; for i := 1 to n do ReadLn(f, a[i]);
  Close(f); Assign(f, 'smucarji.out'); Rewrite(f);

  Najboljsa := 1;
  for s := 1 to n do begin

    { Poiščimo najboljši možni končni položaj s-tega smučarja. }
    while a[s] + t[1] < a[Najboljsa] do Najboljsa := Najboljsa + 1;

    { V nadaljevanju se ukvarjamo z najslabšim končnim položajem. }
    { Katero mesto mora smučar s + 1 doseči, da nas bo prehitel? }
    ui := 0; if n – s > m then r := m + 1 else r := n – s + 1;
    while ui + 1 < r do begin
      { Invarianta: če doseže tekmovalec s + 1 ui-to mesto,
        bo tekmovalca s v skupnem seštevku prehitel, če
        doseže r-to mesto, pa ne. }
      i := (ui + r) div 2;
      if t[i] + a[s + 1] > a[s] then ui := i else r := i;
    end; {while}

    { Tekmovalec s + 1 mora doseči ui-to ali boljše mesto.
      Če nas prehiti k tekmovalcev, bo dosegel k-to.
      Torej mora biti k ≤ ui. }
    i := 1; r := ui;
    while i <= r do begin
      { Invarianta: r = min(u_j + j – 1 : j = 1, . . . , i).
        Zaradi i ≤ r že vemo, da nas lahko prehiti i smučarjev. }
      i := i + 1; { Nas lahko prehiti i + 1 smučarjev? }
      while ui > 0 do
        if t[ui] + a[s + i] <= a[s] then ui := ui – 1 else break;
      { Tekmovalec s + i potrebuje ui-to mesto, da nas bo prehitel.
```

```
        Če nas prehiti k tekmovalcev, bo dosegel (k + 1 − i)-to.
        Torej mora biti k + 1 − i ≤ ui oz. k ≤ ui + i − 1. }
      if r > ui + i − 1 then r := ui + i − 1;
    end; {while i <= r}

    { Lahko nas prehiti i − 1 smučarjev, ne pa tudi i smučarjev. }
    WriteLn(f, Najboljsa, ' ', s + i − 1);
  end; {for s}
  Close(f);
end. {Smucarji}
```

R2003.3.3 Vplivi

Naloga: str. 11

Odnose med inkvizitorji lahko predstavimo z grafom G, v katerem je za vsakega inkvizitorja po ena točka in če inkvizitor u vpliva na inkvizitorja v, naj bo v grafu povezava „$u \to v$“ od točke u do točke v z vplivom $p(u, v)$. Množico vseh točk označimo z $V = \{1, \ldots, n\}$, množico vseh povezav pa z E. Pot od u_0 do u_k je vsako tako zaporedje točk $\pi = \langle u_0, u_1, \ldots, u_k \rangle$, za katerega so $(u_{i-1} \to u_i) \in E$ za vse $i = 1, \ldots, k$. Vpliv poti je $p(\pi) = \min\{p(u_{i-1}, u_i) : i = 1, \ldots, k\}$. Označimo začetno točko s s (pri tej nalogi je $s = 1$, ker ima veliki inkvizitor številko 1); naj bo $p_M(u)$ vpliv najvplivnejše poti od s do u. Za prazno pot $\pi = \langle u_0 \rangle$ je smiselno definirati $p(\pi) = \infty$ (to sledi iz želje, naj vedno velja $\min(A \cup \{a\}) = \min\{\min A, a\}$ in zato $\min \emptyset = \infty$); zato je $p_M(s) = \infty$.

Ko za neko točko u odkrijemo neko novo najboljšo pot od s do nje, je vsekakor pametno pregledati še njene naslednice, torej tiste v, za katere obstaja povezava $u \to v$. Možne poti od s do v so namreč tudi tiste, ki vodijo skozi u, tako da, če smo odkrili do u neko novo najboljšo pot, se jo bo mogoče dalo podaljšati s korakom $u \to v$ v novo najboljšo pot do v. Naš program bo vzdrževal neko množico Q vseh točk, ki jih bo treba na ta način še pregledati. V vsakem koraku vzame neko točko u iz Q, pregleda njene naslednice in če za katero od njih pri tem odkrije novo najboljšo pot, doda to naslednico v Q. Postopek se ustavi, ko se množica Q izprazni. V tabeli $p_M[\cdot]$ bomo hranili vpliv najboljše doslej znane poti do u; na koncu hočemo seveda priti do tega, da bo za vsak u veljalo $p_M[u] = p_M(u)$ (torej: da bomo poznali res prave najboljše poti). Na začetku, ko za noben u (razen $u = s$) ne poznamo še nobene poti od s do u, postavimo $p_M[u]$ na $-\infty$, saj je vpliv vsake poti od s do u enak vplivu neke povezave, ta pa je $> -\infty$, tako da nam ni treba skrbeti, da bi zaradi prevelike začetne vrednosti $p_M[u]$ kakšno pot spregledali.

1 za vsako točko $u \in V$ naj bo $p_M[u] := -\infty$;
2 $Q := \{s\}$; $p_M[s] := \infty$;
3 ponavljaj, dokler množica Q ni prazna:
4   naj bo u nek element Q; vzemi u iz Q;
5   za vsako u-jevo neposredno naslednico v (torej: za vsako $(u \to v) \in E$):
6     $c := \min\{p_M[u], p(u, v)\}$;
7   če je $c > p_M[v]$,
8       $p_M[v] := c$;
9   če $v \notin Q$, dodaj v v Q;

Kaj lahko povemo o **časovni zahtevnosti** tega postopka? Glavna zanka (vrstice 3–9) vsakič vzame neko točko iz Q, tako da se lahko izvede le tolikokrat, kolikorkrat se v Q kaj doda. Neko točko pa dodamo v Q le takrat, kadar se ji poveča $p_M[u]$ (in še to le v primeru, če u tisti hip še ni bil v vrsti). Kot smo videli, je $p_M[u]$ vedno enak vplivu neke povezave našega grafa; če označimo število povezav v grafu z m, vidimo, da ima $p_M[u]$ le m možnih vrednosti (pa še začetno vrednost $-\infty$), torej se lahko poveča le m-krat. Tako torej vidimo, da se lahko vsaka točka znajde v množici Q le m-krat. Notranja zanka (vrstice 5–9) pregleda vse naslednice točke u in če mora pregledati po enkrat vsako $u \in V$, pregleda s tem ravno vse povezave v grafu; torej se pri tem izvede skupaj m-krat. Če se vsaka u znajde v množici Q največ m-krat, bo morala notranja zanka pregledati vse povezave največ m-krat, torej se bo skupaj izvedla največ m^2-krat. Ta razmislek nam torej pove, da se zunanja zanka izvede največ nm-krat, notranja pa največ m^2-krat, tako da je časovna zahtevnost našega algoritma $O(m(n+m))$. Ker je rečeno, da obstaja do vsake točke u vsaj ena pot od s, mora v vsako u (razen mogoče v s) kazati vsaj ena povezava, tako da mora biti povezav vsaj $n-1$; zato lahko $O(m(n+m))$ poenostavimo kar v $O(m^2)$. Po drugi strani je število povezav, m, navzgor omejeno s tem, koliko je vseh parov točk (pri tem se še spomnimo, da nobena točka ne kaže sama nase, saj pravi naloga, da je $p(u,u) = 0$); torej je $m \leq n(n-1)$. Zato je časovna zahtevnost našega postopka v najslabšem primeru $O(n^4)$ (če je graf dovolj gost — se pravi, če ima veliko povezav), znala pa bi biti tudi le $O(n^2)$, če je graf dovolj redek (torej če ima dovolj malo povezav).

Dokaz pravilnosti. Prepričajmo se še, da naš postopek res vedno poišče prave rešitve, torej da na koncu izvajanja za vsak u velja $p_M[u] = p_M(u)$. Definirajmo v mislih nov graf G' z istimi točkami V kot doslej, le množica povezav E' bo malo manjša:

$$(u \to v) \in E' \Leftrightarrow (u \to v) \in E \wedge p_M(v) = \min\{p_M(u), p(u,v)\}.$$

Za prvotni graf G nam že opis naloge zagotavlja, da so v njem vse točke dosegljive iz s. Prepričajmo se zdaj, da velja to tudi za G'. Pa recimo, da neka v v grafu G' ni dosegljiva iz s. Naj bo W množica vseh točk, do katerih lahko v G' pridemo, če začnemo iz v in gremo v nasprotni smeri povezav. Očitno $s \notin W$ (sicer bi bila v dosegljiva iz s tudi v G'). Naj bo W' množica vseh tistih točk iz W, ki imajo maksimalno $p_M(\cdot)$ (torej: vsaj tolikšno kot katerakoli druga točka iz W). Naj bo $(u \to w)$ poljubna povezava (v G) med neko $u \notin W'$ in $w \in W'$. (Neka taka povezava gotovo obstaja, saj W ni prazna, torej tudi W' ni prazna in ker je v G vsaka $w \in W'$ dosegljiva iz s, s pa ni v W', mora biti na vsaki poti od s do w prej ali slej neka povezava iz $V - W'$ v W'.) V E' te povezave ni, saj za $u \notin W$ to ne gre že po definiciji W, za $u \in W - W'$ pa ne gre zato, ker bi bilo sicer (po definiciji E') $p_M(u) \geq p_M(w)$ in bi bil tedaj po definiciji W' tudi u v W'. Ker povezave $(u \to w)$ ni v E', je pa v E, mora biti (po definiciji E') $p_M(w)$ strogo večji od $\min\{p_M(u), p(u,w)\}$. (Po definiciji E' sledi pravzaprav, da mora biti različen, toda manjši ne more biti, kar je jasno že iz definicije funkcije p_M.) — Naj bo π najboljša pot od s do neke $w' \in W'$. Ker se začne v $s \notin W'$, mora vsaj enkrat prestopiti iz $V - W'$ v W', torej mora vsebovati neko povezavo $(u \to w)$ za nek $u \notin W'$ in nek $w \in W'$. Pot π lahko v mislih predelamo tako, da tisti del poti do u zamenjamo z najboljšo potjo do u; nastali poti, ki gotovo ni slabša od prvotne, recimo π'. Torej je $p_M(w') = p(\pi) \leq p(\pi') \leq \min\{p_M(u), p(u,w)\}$; kot pa smo ugotovili malo prej, je to naprej $< p_M(w)$, kar je po definiciji W' enako $p_M(w')$. Zdaj smo prišli v protislovje, češ da je $p_M(w') < p_M(w')$. Tako vidimo, da je nemogoče, da kakšna v v grafu G' ne bi bila dosegljiva iz s.

Vrednosti, ki jih program vpisuje v $p_M[u]$, so vedno enake $p(\pi)$ za neko pot π od s do u. Zmotimo se torej lahko le tako, da je za nek u vrednost $p_M[u]$ tudi na koncu izvajanja programa manjša od prave vrednosti $p_M(u)$, ne more pa biti $p_M[u]$ večja od prave vrednosti. Recimo, da za nek u naš program ne najde prave rešitve. Naj bo π najkrajša pot (najkrajša po številu povezav) od s do u v grafu G' (ki gotovo obstaja, saj smo se malo prej prepričali, da so tudi v G' vse točke dosegljive iz s). Naj bo v prva točka na tej poti, za katero naš program ne najde prave rešitve; naj bo w njena neposredna predhodnica na tej poti (kajti v gotovo ima predhodnico, saj je na tej poti le s brez predhodnice, za s pa poznamo pravo rešitev že na začetku izvajanja programa, tako da $v \neq s$). Ker je w dosegljiva iz s, je $p_M(w) > -\infty$, in ker naš program odkrije pravilno rešitev za w, mora med njegovim izvajanjem $p_M[w]$ vsaj enkrat narasti, da od svoje začetne vrednosti $-\infty$ pride do $p_M(w)$. Torej dodamo w vsaj enkrat v Q; torej ga tudi vsaj enkrat vzamemo iz Q. Ko ga zadnjič vzamemo iz Q, je $p_M[w]$ že enako $p_M(w)$. Takrat pogledamo vse w-jeve naslednike, torej tudi v; pri slednjem izračunamo $c := \min\{p_M[w], p(w, v)\}$. Po predpostavki, da imamo že pravi rezultat za w, je c enak $\min\{p_M(w), p(w, v)\}$; in ker smo w in v izbrali tako, da je $(w \to v) \in E'$, je to naprej enako $p_M(v)$. Torej, tudi če je bil $p_M[v]$ doslej še manjši od $p_M(v)$, bi ga zdaj postavili na to vrednost. Ker se vrednosti tabele $p_M[\cdot]$ nikoli ne zmanjšujejo, bo tudi na koncu $p_M[v] = p_M(v)$, torej bo naš program našel pravilno rešitev za v. Prišli smo v protislovje, saj smo na začetku rekli, da se za v naš program zmoti.

Implementacija množice Q. Postopek torej načeloma deluje in daje pravilne rezultate ne glede na to, kako jemljemo točke iz množice Q. Vendar pa se izkaže, da vrstni red jemanja pomembno vpliva na časovno zahtevnost postopka. Če organiziramo Q kot **sklad**, torej vzamemo iz nje vedno tisto točko, ki smo jo nazadnje dodali vanjo, in če pregledujemo naslednike (v notranji zanki) v nekem dovolj nesrečno izbranem vrstnem redu, se pri nekaterih grafih res lahko zgodi, da ima naš postopek zahtevnost $O(n^4)$.

Če organiziramo Q kot **vrsto**, torej vzamemo iz Q vedno tisto točko, ki je že najdlje v njej, pa se da dokazati, da ne bo nobena točka prišla v vrsto več kot n-krat.[4] Zato mora notranja zanka pregledati vse povezave iz E največ n-krat in časovna zahtevnost našega postopka je tedaj le še $O(nm)$, kar je v najslabšem primeru $O(n^3)$, nikoli pa ne $O(n^4)$. Tako dobljen postopek je pravzaprav različica Bellman-Fordovega algoritma za iskanje najkrajših poti v grafih (le da namesto najkrajših išče najvplivnejše poti).

Še bolje je, če vzamemo iz Q vedno tisto točko u, ki ima med vsemi v Q največjo vrednost $p_M[u]$. Temu pravimo, da smo Q organizirali v **prioritetno vrsto**; naš postopek postane različica znanega Dijkstrovega algoritma za iskanje

[4]Načrt dokaza: mislimo si, da za vsako točko u vzdržujemo še neko vrednost $h[u]$, ki je na začetku 0; vsakič, ko pri pregledovanju naslednikov nekega u dodamo nek v v vrsto, pa postavimo $h[v] := h[u] + 1$. Potem lahko z indukcijo pokažemo, da na začetku vsake iteracije zunanje zanke velja: obstaja nek k, tako da za prvih nekaj točk v vrsti velja $h[u] = k$, za preostale (nič ali več točk) pa $h[u] = k + 1$. Iz te ugotovitve sledi, da ko točke jemljemo iz vrste, dobivamo take z nepadajočimi vrednostmi h. Naj bo $d(v)$ dolžina najkrajše poti od s do v v grafu G'. Potem lahko z indukcijo po $d(v)$ pokažemo, da po zadnji postavitvi točke v v vrsto prav gotovo velja $h[v] \leq d(v)$. Na koncu še pokažimo, da se, če neko v večkrat dodamo v vrsto, po vsakem dodajanju njena $h[v]$ strogo poveča. Iz vsega tega že sledi, da nobene v ne moremo dodati v vrsto več kot $d(v)$-krat, $d(v)$ pa je seveda vedno $< n$.

najkrajših poti. Zdaj se da pokazati, da vsako točko vzamemo iz vrste največ enkrat[5] (pravzaprav natanko enkrat, saj je vsaka dosegljiva iz s), tako da se notranja zanka izvede le $O(m)$-krat, zunanja pa $O(n)$-krat. Prioritetno vrsto se običajno implementira z dvojiško kopico, pri kateri dodajanje in brisanje točk zahteva $O(\lg n)$ časa; zahtevnost našega algoritma bi bila v tem primeru $O(m \lg n)$, kar je v najslabšem primeru $O(n^2 \lg n)$.[6]

Ko smo merili časovno zahtevnost na konkretnih testnih primerih pri tej nalogi, se ni različica s prioritetno vrsto odrezala čisto nič hitreje kot tista z navadno vrsto, razen pri tistem testnem primeru, ki je bil nalašč sestavljen tako, da ima vrsta pri njem res zahtevnost $O(n^3)$ (notranja zanka se izvede približno $\frac{1}{2}n^3$-krat[7]). Eden od ostalih testnih primerov je sestavljen tako, da je zelo neugoden za različico s skladom,[8] ostali testni primeri pa so bili pripravljeni naključno. Na vseh teh sta porabili različici s prioritetno in z navadno vrsto praktično enako veliko časa. Zato objavljamo kar rešitev z navadno vrsto, saj je krajša in preprostejša.

---

[5]Pri dokazu si je koristno pomagati z naslednjo invarianto, ki velja na koncu vsake iteracije zunanje zanke. Naj bo A množica vseh točk, ki smo jih že kdaj vzeli iz Q. Naj bo B množica vseh tistih neposrednih naslednic točk iz A, ki same niso v A. Naj bo C množica vseh ostalih točk (torej $V - A - B$). (1*a*) Za vsako $u \in A$ je $p_M[u]$ zdajle že enak vplivu najvplivnejše poti od s do u; (1*b*) nadalje je med potmi od s do u s tem vplivom tudi vsaj ena taka, ki gre ves čas le po točkah iz A. (2*a*) Za vsako $v \in B$ je $p_M[v]$ zdajle enak vplivu najvplivnejše take poti od s do v, ki gre ves čas le po točkah iz A, razen v zadnjem koraku, ko stopi v v. (2*b*) Množica Q vsebuje vse točke iz B in nobene točke iz A ali C. (3) Za vsako $w \in C$ je $p_M[w] = -\infty$.

[6]S kakšno drugo vrsto kopic, npr. Fibonaccijevimi, bi šlo asimptotično tudi hitreje, ker bi dodajanje v Q in povečevanje vrednosti p_M vzelo v povprečju le $O(1)$ časa, tako da bi bila časovna zahtevnost celega algoritma le še $O(m + n \lg n)$. Vendar so te kopice bolj zapletene in se jih v praksi najbrž ne uporablja kaj dosti. — Še ena alternativa kopici bi bila, da bi vsakič pregledali kar vse $p_M[u]$ za vse $u \in Q$ in na ta način poiskali največjo. To bi nam vzelo vsakič $O(n)$ časa, torej skupno $O(n^2)$; zato pa je vsak popravek kakšne vrednosti $p_M[u]$ le še $O(1)$, tako da je zahtevnost celega algoritma zdaj $O(n^2 + m)$. To je lahko boljše ali pa tudi slabše od $O(m \lg n)$, odvisno od tega, ali je naš graf gost ($m = O(n^2)$) ali redek ($m = O(n)$).

[7]Primer takega grafa: n točk, oštevilčenih z $1, \ldots, n$; od vsake točke u obstajajo povezave do vseh ostalih točk s težo u, razen povezave do $u + 1$, ki ima težo $2n - u$. Ta povezava (torej $u \to u+1$) naj bo navedena kot zadnja med vsemi povezavami, ki kažejo iz u. Naš algoritem bi na začetku vzel iz vrste točko 1 (s $p_M[1] = \infty$) in nato dodal v vrsto točke $3, \ldots, n$ s $p_M[u] = 1$ ter za njimi še točko 2 s $p_M[2] = 2n - 1$. Ko bi nato jemal točke $3, \ldots, n$ iz vrste, se ne bi spremenilo nič; nato pa bi vzel iz vrste točko 2 in dodal v vrsto točke $4, \ldots, n$ s $p_M[u] = 2$ ter še točko 3 s $p_M[3] = 2n - 2$. Tako bi šlo naprej; točko 1 dodamo v vrsto enkrat, ostale točke u pa po $(u - 1)$-krat. Tako se izvede $1 + \sum_{u=2}^{n}(u - 1) = 1 + n(n - 1)/2$ jemanj iz vrste, ob vsakem od njih pa je treba pregledati vseh $n - 1$ naslednic trenutne točke, tako da je število izvajanj notranje zanke kar $(1 + n(n - 1)/2)(n - 1) = (n^3 - 2n^2 + 3n - 2)/2$, skratka, približno $n^3/2$.

[8]Primer takega grafa: imejmo točke $u, v_1, \ldots, v_a, w_1, \ldots, w_b, x_1, \ldots, x_c$; u vpliva na vse v_i s težo ib; vsaka v_i vpliva na vsako w_j s težo $(i - 1)b + j$; vsaka w_j vpliva na vsako x_k s težo ab; vsaka x_k vpliva na vse ostale točke s težo 1. Povezave, ki izhajajo iz posamezne točke, naj bodo navedene po padajoči teži. Naš algoritem bi najprej vzel s sklada točko u in naložil nanj $v_a, v_{a-1}, \ldots, v_2, v_1$ z utežmi $p_M[v_i] = ib$. Nato bi vzel s sklada v_1 in naložil nanj $w_b, w_{b-1}, \ldots, w_2, w_1$ z utežmi $p_M[w_j] = j$. Nato bi vsako w_j vzel s sklada, naložil nanj vsakič vse x_k (s $p_M[x_k] = p_M[w_j]$) ter jih takoj spet pobral s sklada. Po vsem tem bi vzel s sklada v_2 in naložil nanj spet $w_b, w_{b-1}, \ldots, w_2, w_1$, vendar zdaj z večjimi utežmi: $p_M[w_j] = j + b$. Zato bi pri jemanju vsake w_j s sklada zdaj na novo dodal na sklad tudi vse x_k (spet s $p_M[x_k] = p_M[w_j]$, le da je $p_M[w_j]$ zdaj večja kot prej). Podobno bi se potem zgodilo pri v_3, v_4 in tako naprej. To pomeni, da se u znajde na skladu enkrat, vsaka v_i tudi enkrat, vsaka w_j se pojavi na skladu a-krat, vsaka x_k pa ab-krat. Če upoštevamo še izhodne stopnje teh točk, vidimo, da se mora notranja zanka izvesti $((1 + a)b + abc + abcd)$-krat, če je $d = a + b + c$ izhodna stopnja točk x_k. Če vzamemo $a = b = c = (n - 1)/3$, imamo kar $(n^4 - 3n^3 + 6n^2 + 2n - 6)/27 \approx n^4/27$ izvajanj notranje zanke.

```
program Vplivi;
const MaxN = 1000; MaxM = 200000; MaxVpliv = 1000000000;
type TabelaN = array [1..MaxN] of integer;
     TabelaM = array [1..MaxM] of integer;
     TabelaB = array [1..MaxN] of boolean;
var T: text;
    i, u, v, m, n, Glava, Dolz, p, Kand: integer;
    Kdo, NaKoga1, Vpliv1, NaKoga, Vpliv, Vrsta: ↑TabelaM;
    NaKoliko, Prvi, NajVpliv: ↑TabelaN; VVrsti: ↑TabelaB;
begin
  Assign(T, 'vplivi.in'); Reset(T); ReadLn(T, n); ReadLn(T, m);

  { Preberimo vse povezave (u, v) v tabeli Kdo in NaKoga1.
    V Vpliv1 shranimo vplive povezav, v NaKoliko pa za
    vsakega inkvizitorja zapišemo, na koliko drugih vpliva. }
  New(Kdo); New(NaKoga1); New(Vpliv1); New(NaKoliko);
  for u := 1 to n do NaKoliko↑[u] := 0;
  for i := 1 to m do begin
    ReadLn(T, u, v, p); NaKoliko↑[u] := NaKoliko↑[u] + 1;
    Kdo↑[i] := u; NaKoga1↑[i] := v; Vpliv1↑[i] := p;
  end;
  Close(T); New(Prvi); New(NaKoga); New(Vpliv);

  { Povezave uredimo tako, da bomo imeli za vsakega inkvizitorja
    pri roki seznam vseh, na katere vpliva: za inkvizitorja u
    so to inkvizitorji NaKoga↑[Prvi↑[u]..Prvi↑[u] + NaKoliko↑[u] − 1]. }
  Prvi↑[1] := 1; for u := 2 to n do Prvi↑[u] := Prvi↑[u − 1] + NaKoliko↑[u − 1];
  for u := 1 to n do NaKoliko↑[u] := 0;
  for i := 1 to m do begin
    u := Kdo↑[i];
    NaKoga↑[Prvi↑[u] + NaKoliko↑[u]] := NaKoga1↑[i];
    Vpliv↑[Prvi↑[u] + NaKoliko↑[u]] := Vpliv1↑[i];
    NaKoliko↑[u] := NaKoliko↑[u] + 1;
  end; {for}
  Dispose(Kdo); Dispose(NaKoga1); Dispose(Vpliv1); New(NajVpliv);

  { Glavni del našega postopka. V vrsti (tabela Vrsta) je Dolz
    točk, prva je na indeksu Glava. (Če padejo indeksi čez rob,
    nadaljujemo pri indeksu 1.) VVrsti↑[u] pove, če je u v vrsti. }
  NajVpliv↑[1] := MaxVpliv + 1; for u := 2 to n do NajVpliv↑[u] := −1;
  Glava := 1; Dolz := 1; New(Vrsta); New(VVrsti); Vrsta↑[Glava] := 1;
  VVrsti↑[1] := true; for u := 2 to n do VVrsti↑[u] := false;
  while Dolz > 0 do begin
    u := Vrsta↑[Glava]; { Vzamemo nek u iz vrste. }
    VVrsti↑[u] := false; Dolz := Dolz − 1;
    for i := 1 to NaKoliko↑[u] do begin { Na koga vse lahko u vpliva? }
      v := NaKoga↑[Prvi↑[u] + i − 1]; p := Vpliv↑[Prvi↑[u] + i − 1];
      if p < NajVpliv↑[u] then Kand := p else Kand := NajVpliv↑[u];
      if Kand > NajVpliv↑[v] then begin { Našli smo novo najboljšo pot do v. }
        NajVpliv↑[v] := Kand;
        if not VVrsti↑[v] then begin { Dodajmo v v vrsto. }
          Vrsta↑[(Glava + Dolz) mod n + 1] := v;
          Dolz := Dolz + 1; VVrsti↑[v] := true;
        end; {if}
      end; {if}
```

```
      end; {for}
      Glava := Glava + 1; if Glava > n then Glava := 1;
   end; {while}
   Dispose(Vrsta); Dispose(VVrsti); Dispose(Vpliv);
   Dispose(NaKoga); Dispose(NaKoliko); Dispose(Prvi);

   { Izpis rezultatov. }
   Assign(T, 'vplivi.out'); Rewrite(T);
   for u := 2 to n do WriteLn(T, NajVpliv↑[u]);
   Close(T); Dispose(NajVpliv);
end. {Vplivi}
```

R2003.3.4 Vodenje projektov

Naloga: str. 12

Podobno kot pri prejšnji nalogi si lahko tudi tu pomagamo z grafom. Vsaka točka ustreza eni od aktivnosti, povezava od u do v (označimo jo „$u \to v$“) pa naj obstaja natanko v primeru, ko je v vhodnih podatkih v navedena kot odvisna od u. To, da je neka aktivnost v posredno ali neposredno odvisna od u, pa pomeni, da je v grafu neka pot (zaporedje povezav) od u do v.

Naloga zdaj pravzaprav zahteva, naj zbrišemo iz grafa čim več povezav (to ustreza brisanju odvisnosti), ne da bi pri tem spremenili relacijo dosegljivosti v njem. (V teoriji grafov pravijo temu problemu *tranzitivna redukcija* grafa.) Z drugimi besedami, če je neka točka dosegljiva iz neke druge v prvotnem grafu, mora ostati dosegljiva iz nje tudi po brisanju. Preden zbrišemo neko povezavo $u \to v$, moramo torej preveriti, da je v dosegljiva iz u tudi po kakšni drugačni (daljši) poti. To lahko učinkovito preverimo takole: naj bo $R(u)$ množica vseh točk, dosegljivih iz u (v enem ali več korakih). Naj bo $P(v)$ množica vseh neposrednih predhodnic točke v (torej takih, iz katerih kaže v v neka povezava). Potem, če je presek $R(u) \cap P(v)$ neprazen, pomeni, da je neka v-jeva predhodnica (recimo ji w) tudi dosegljiva iz u (v enem ali več korakih), zato pa lahko povezavo $u \to v$ zbrišemo, pa se bo od u do v še vedno dalo priti skozi w. (Poti od u do w pa z brisanjem povezave $u \to v$ prav gotovo ne pretrgamo, saj če bi tista pot vsebovala tudi to povezavo, bi pomenilo, da se da iz v iti naprej po tej poti do w in nato (ker je w predhodnica točke v) v v, torej obstaja v grafu cikel — nam pa naloga zagotavlja, da cikličnih odvisnosti ne bo.)

```
ponovi za vsako točko u ∈ V:
   naj bo R(u) množica vseh točk, dosegljivih iz u v enem ali več korakih;
   za vsako u-jevo neposredno naslednico v (torej: za vsako (u → v) ∈ E):
      naj bo P(v) množica v-jevih neposrednih predhodnic;
      če je R(u) ∩ P(v) ≠ ∅,
         izpiši, da se lahko pobriše povezava u → v;
```

Dokaz pravilnosti. Naš program torej predlaga brisanje neke povezave $u \to v$ natanko tedaj, ko obstaja v prvotnem grafu neka daljša pot $u \rightsquigarrow w \to v$. Prepričajmo se, da množica tistih povezav iz E, za katere naš postopek ne predlaga brisanja (recimo tej množici F) res ohrani celotno relacijo dosegljivosti prvotne množice E.

Recimo, da bi obstajala v E neka pot

$$u_1 \rightsquigarrow u_2 \rightsquigarrow \ldots \rightsquigarrow u_k \rightsquigarrow v_k \rightsquigarrow \ldots \rightsquigarrow v_2 \rightsquigarrow v_1,$$

kjer za vsak i velja $(u_i \neq u_{i-1}) \vee (v_i \neq v_{i-1})$ in v_i v F ni dosegljiv iz u_i. (Iz slednjega tudi sledi $u_i \neq v_i$.) Recimo še, da je del med u_k in v_k dolg vsaj dva koraka (dve povezavi). (Opazimo, da je dolžina cele poti vsaj $k-1$ korakov.)

Naj bo π del gornje poti med u_k in v_k. Če bi za vsako povezavo $u \to v$ iz π veljalo, da je v v F dosegljiv iz u, bi bil tudi v_k v F dosegljiv iz u_k. Torej obstaja med temi povezavami neka $u_{k+1} \to v_{k+1}$, da v_{k+1} v F ni dosegljiv iz u_{k+1}. Ker je π po predpostavki dolga vsaj dva koraka (in ker v našem grafu ni ciklov), to seveda pomeni, da ne more biti obenem $u_{k+1} = u_k$ in $v_{k+1} = v_k$ — vsaj nekaj od tega dvojega ni res. Poleg tega, ker v F točka v_{k+1} ni dosegljiva iz u_{k+1}, je moral naš program med drugim pobrisati tudi povezavo $u_{k+1} \to v_{k+1}$, torej obstaja v E neka daljša (vsaj dva koraka dolga) pot π' od u_{k+1} do v_{k+1}. Vidimo torej: če velja predpostavka iz prejšnjega odstavka (torej da obstaja pot z navedenimi lastnostmi) za k, potem velja tudi za $k+1$.

Čim torej obstaja v E neka pot $u_1 \rightsquigarrow v_1$, dolga vsaj dva koraka, in v_1 v F ni dosegljiv iz u_1, bi lahko n-krat uporabili gornji razmislek in videli, da mora v E obstajati neka pot od u_1 do v_1, dolga vsaj n korakov; toda tako dolge poti v grafu na samo n točkah ne more biti, ne da bi vsebovala cikel, ciklov pa naš graf ne vsebuje.

Recimo torej, da je nek v_1 dosegljiv iz nekega u_1 v E, ne pa v F. Prejšnji odstavek je pokazal, da v E ne obstaja nobena pot od u_1 do v_1, dolga vsaj dva koraka. Ker pa je v E točka v_1 dosegljiva iz u_1, pomeni, da v E obstaja povezava $u_1 \to v_1$; in ker v F točka v_1 ni dosegljiva iz u_1, pomeni, da je naš postopek tisto povezavo $u_1 \to v_1$ pobrisal; to pa pomeni, da je videl v E še neko daljšo (vsaj dva koraka dolgo) pot od u_1 do v_1, mi pa smo rekli, da take ni. Prišli smo v protislovje, torej takega neugodnega para u_1 in v_1 ni.

Dosedanji razmislek je pokazal, da nismo z brisanjem izgubili nobene dosegljivosti; obenem pa z brisanjem seveda tudi ni mogoče nobene dosegljivosti pridobiti, tako da zdaj vemo, da v množici povezav, ki jih po brisanju pusti naš postopek, res veljajo natanko iste dosegljivosti kot v prvotni množici povezav E. Torej rešitev, ki jo vrne naš program, ustreza zahtevam naloge; prepričajmo se še o tem, da je tudi najboljša možna.

Recimo, da imamo dve množici povezav, F_1 in F_2, obe dobljeni iz E z brisanjem nekaj povezav; in recimo, da je tako pri F_1 kot pri F_2 dosegljivost v grafu enaka kot pri celi množici E. Recimo, da obstaja v F_1 neka povezava $u \to v$, ki je v F_2 ni. Ker F_1 ima to povezavo, je v v prvotnem grafu dosegljiv iz u, torej mora biti v F_2 tudi, na primer po neki poti $u_0 \to u_1 \to \ldots \to u_{k-1} \to u_k$ za $u_0 = u$, $u_k = v$. Ker so vse te povezave $u_{i-1} \to u_i$ v F_2, so morale biti že v E, torej mora biti vsaka u_i tudi v F_1 dosegljiva iz u_{i-1}. Če se v F_1 od u_{i-1} do u_i ne bi dalo priti drugače kot tako, da obiščemo tudi povezavo $u \to v$, bi to pomenilo, da obstaja cikel $u = u_0 \rightsquigarrow u_1 \rightsquigarrow \ldots \rightsquigarrow u_i \rightsquigarrow u$ ali pa cikel $v \rightsquigarrow u_{i+1} \rightsquigarrow u_{i+2} \rightsquigarrow \ldots \rightsquigarrow u_{k-1} \rightsquigarrow u_k = v$;[9] torej bi bil tak cikel že v prvotnem grafu, ta pa je, kot zagotavlja naloga, acikličen, tako da se to ne more zgoditi. Torej se da v F_1 od vsake u_{i-1} priti do u_i brez obiska povezave $u \to v$; torej se da tudi od u_0 $(= u)$ priti do u_k $(= v)$, ne da bi obiskali povezavo $u \to v$; torej lahko to povezavo brez škode pobrišemo iz F_1, pa se dosegljivost v grafu ne bo nič spremenila.

[9] Oba tadva sprehoda sta možna; stvar je le v tem, da nista nujno oba cikla — če je $i = 0$ in $u = u_i = u_0 = u$, potem prvi od njiju sploh ni cikel; podobno pa, če je $i = k-1$ in $v = u_{i+1} = u_k = v$, drugi od njiju ni cikel; oboje naenkrat pa ne more biti res, ker bi to pomenilo $k = 1$ in F_2 bi vseboval kar povezavo $u \to v$, to pa smo predpostavili, da je ne.

Prejšnji odstavek nam med drugim pove, da ne moreta obstajati dve različni najboljši rešitvi (iz gornjega razmisleka bi takoj sledilo, da lahko eno od njiju še zmanjšamo in da prej torej ni bila najboljša). Pove pa nam tudi, da je rešitev, ki jo predlaga naš program, res najboljša. Kajti če ni najboljša, je moralo ostati po brisanju več povezav kot pri najboljši, to pa zagotovo pomeni, da ima naša rešitev kakšno tako povezavo $u \to v$, ki je najboljša nima. Ker obenem ohranja dosegljivost, lahko uporabimo razmislek iz prejšnjega odstavka in vidimo, da bi lahko iz naše rešitve F kakšno povezavo $u \to v$ še pobrisali, ker obstaja v F namesto nje še neka daljša pot od u do v. Toda ta daljša pot obstaja potemtakem tudi v E in bi jo zato naš postopek moral opaziti in $u \to v$ pobrisati in je potem sploh ne bi bilo v F.

Pri **implementaciji** postopka je koristno, če si za vsako točko pripravimo seznam predhodnic in seznam naslednic. Seznami naslednic bodo prišli prav pri ugotavljanju, kaj vse je dosegljivo iz neke u. Ko za neko točko na novo izvemo, da je dosegljiva iz u, jo dodajmo v neko vrsto, da bomo kasneje pregledali še njene naslednice (če jih kaj ima), saj so one potemtakem tudi dosegljive iz u. Takemu načinu pregledovanja grafa pravimo tudi *iskanje v širino*. To, ali je neka točka v dosegljiva iz u, si spodnji program zapomni v celici Dosegljiva[v]. Da ni treba za vsak u posebej postavljati cele tabele Dosegljiva na false, si pomagamo tako, da namesto vrednosti true uporabimo kar vrednost u, katerakoli manjša vrednost pa velja kot false; na začetku postavimo vse celice te tabele na 0. Kakorkoli že, tabelo Dosegljiva lahko potem uporabimo ob pregledovanju v-jevih predhodnic, da vidimo, če je katera od njih tudi dosegljiva iz u.

```
program VodenjeProjektov;
const MaxN = 1000; MaxM = 10000;
type TabelaN = array [1..MaxN] of integer;
     TabelaM = array [1..MaxM] of integer;
var T: text;
    m, n, i, j, u, v, w, Glava, Rep: integer;
    InDeg, OutDeg, PrviPred, PrviNasl, Vrsta, Dosegljiva: ↑TabelaN;
    Zac, Kon, Pred, Nasl: ↑TabelaM;
    Zbrisi: boolean;
begin
  Assign(T, 'projekti.in'); Reset(T); ReadLn(T, n); ReadLn(T, m);

  { Preberimo seznam povezav. }
  New(InDeg); New(OutDeg); New(Zac); New(Kon);
  for u := 1 to n do begin InDeg↑[u] := 0; OutDeg↑[u] := 0 end;
  for i := 1 to m do begin
    ReadLn(T, u, v); Zac↑[i] := u; Kon↑[i] := v;
    OutDeg↑[u] := OutDeg↑[u] + 1; InDeg↑[v] := InDeg↑[v] + 1;
  end; {for}
  Close(T); New(PrviPred); New(PrviNasl); New(Pred); New(Nasl);

  { Pripravimo sezname predhodnic in naslednic.
    u-jeve predhodnice bodo Pred↑[PrviPred↑[u]..PrviPred↑[u] + InDeg↑[u] − 1],
    naslednice pa Nasl↑[PrviNasl↑[u]..PrviNasl↑[u] + OutDeg↑[u] − 1]. }
  PrviPred↑[1] := 1; PrviNasl↑[1] := 1;
  for u := 2 to n do PrviPred↑[u] := PrviPred↑[u − 1] + InDeg↑[u − 1];
  for u := 2 to n do PrviNasl↑[u] := PrviNasl↑[u − 1] + OutDeg↑[u − 1];
  for u := 1 to n do begin InDeg↑[u] := 0; OutDeg↑[u] := 0 end;
  for i := 1 to m do begin
```

```
      u := Zac↑[i]; v := Kon↑[i];
      Nasl↑[PrviNasl↑[u] + OutDeg↑[u]] := v; OutDeg↑[u] := OutDeg↑[u] + 1;
      Pred↑[PrviPred↑[v] + InDeg↑[v]] := u; InDeg↑[v] := InDeg↑[v] + 1;
   end; {for}
   Dispose(Zac); Dispose(Kon); New(Vrsta); New(Dosegljiva);

   { Preglejmo vsako točko. }
   Assign(T, 'projekti.out'); Rewrite(T);
   for u := 1 to n do Dosegljiva↑[u] := 0;
   for u := 1 to n do begin

      { Kaj vse je dosegljivo iz u? }
      Glava := 1; Rep := 1; Vrsta↑[1] := u; Dosegljiva↑[u] := u;
      while Glava <= Rep do begin
         v := Vrsta↑[Glava]; Glava := Glava + 1;
         for i := 1 to OutDeg↑[v] do begin { Preglejmo v-jeve naslednice. }
            w := Nasl↑[PrviNasl↑[v] + i − 1];
            if Dosegljiva↑[w] <> u then { Dodajmo w v vrsto. }
               begin Rep := Rep + 1; Vrsta↑[Rep] := w; Dosegljiva↑[w] := u end;
         end; {for}
      end; {while}

      { Preglejmo vse povezave u → v, ki izhajajo iz u. }
      for i := 1 to OutDeg↑[u] do begin
         v := Nasl↑[PrviNasl↑[u] + i − 1]; Zbrisi := false;
         { Ali je kakšna v-jeva predhodnica tudi dosegljiva iz u? }
         for j := 1 to InDeg↑[v] do begin
            w := Pred↑[PrviPred↑[v] + j − 1];
            if (Dosegljiva↑[w] = u) and (w <> u) then
                  begin Zbrisi := true; break end;
         end; {for j}
         if Zbrisi then WriteLn(T, u, ' ', v);
      end; {for i}
   end; {for u}

   Close(T);
   Dispose(Vrsta); Dispose(Dosegljiva); Dispose(Nasl); Dispose(Pred);
   Dispose(PrviNasl); Dispose(PrviPred);
end. {VodenjeProjektov}
```

Naj bo n število točk in m število povezav našega grafa. Ugotavljanje, kaj je dosegljivo iz posamezne točke, nam vzame le $O(m)$ časa — bilo bi $O(m+n)$, če bi morali vsakič sproti inicializirati celo tabelo Dosegljiva, vendar se temu, kot smo videli, lahko izognemo. Ker je treba ta postopek pognati po enkrat za vsako točko grafa, je skupna zahtevnost $O(mn)$ in poleg tega še $O(n)$ za začetno inicializacijo tabele Dosegljiva. Notranja zanka, ki pregleduje naslednice trenutnega u, se izvede vsega skupaj m-krat (po enkrat za vsako povezavo) in ima vsakič v najslabšem primeru $O(n)$ dela (kolikor ima pač trenutni v neposrednih predhodnic). Branje vhodnih podatkov in priprava seznamov predhodnic in naslednic vzameta še $O(n+m)$ časa. Časovna zahtevnost celotnega postopka je tako $O(mn+n+m) = O(mn)$.

R2003.3.5 Knjižnica

Naloga: str. 13

Naloga je podobna znanemu problemu polnjenja nahrbtnika, le da imamo tu več polic (kot da bi imeli več nahrbtnikov); iskanje rešitve nam olajša zahteva, da moramo pri razporejanju knjig na police spoštovati prvotni vrstni red (po datumu izida). Nalogo bomo rešili z dinamičnim programiranjem, s podobnim algoritmom kot pri nahrbtniku. Tam razmišljamo o tem, kaj storiti, če pride nov predmet (ali ga vzeti ali ne), tu pa poleg tega razmišljamo še o možnosti, če pride nova polica.

Koristno si je zastaviti podprobleme takšne oblike: $f(n, m, g)$ naj bo vrednost najboljše take rešitve za prvih n knjig in prvih m polic, ki ima na zadnji polici zasedenega g ali manj prostora. Potem je $f(0, m, g) = 0$ (za vsak m in g), za $n \geq 1$ pa je

$$
\begin{aligned}
f(n, m, g) = \max\{ \quad & f(n-1, m, g), \\
& (c_n + f(n-1, m, g-d_n)) \text{ [če } g \geq d_n], \\
& (c_n + f(n-1, m-1, d)) \text{ [če } m > 1 \text{ in } g \geq d_n] \quad \}.
\end{aligned}
$$

Pri tem moramo vzeti $c_n = 1$, če nas zanima le največ knjig, ali pa $c_n = d_n$, če nas zanima največja skupna debelina knjig. Prva od treh vrednosti, katerih max iščemo, predstavlja možnost, da n-te knjige sploh ne damo na nobeno polico; druga predstavlja možnost, da n-to knjigo damo na m-to polico, pri čemer je bila na tej polici od prej mogoče že kakšna druga knjiga; tretja vrednost pa predstavlja možnost, da damo n-to knjigo kot prvo na m-to polico: vse knjige pred njo so torej šle na prvih $m-1$ polic (ali pa jih sploh nismo dali na nobeno polico). Rešitev našega prvotnega problema, torej rezultat, ki ga pravzaprav iščemo, pa je $f(N, M, d)$ — kar ustreza podproblemu, ki ima na razpolago vse knjige, vse police in nobene posebne omejitve glede tega, koliko prostora sme biti največ zasedenega na zadnji polici.

Gornja formula deluje tudi v primeru, ko je polic več kot knjig ($m > n$); ne more sicer najti razporedov, pri katerih je zadnjih nekaj polic praznih, ampak saj pri takih bi lahko knjige vedno premestili tako, da bi bilo praznih le prvih nekaj polic, vse od tam naprej pa bi vsebovale vsaj eno knjigo: do takih rešitev lahko gornja formula pride, če začne pri $f(0, m, g)$ za nek $m \geq 1$.

Za vsak konkreten par (n, m) je funkcija $f_{n,m}(g) := f(n, m, g)$ „stopničasta“, torej nepadajoča (če dovolimo bolj zasedeno zadnjo polico, lahko spravimo nanjo vsaj toliko knjig kot prej) in odsekoma konstantna (ko se spremeni, se spremeni zato, ker je mogoče spraviti v regal vsaj eno knjigo več kot prej, torej se vrednost f poveča vsaj za 1; dokler pa števila knjig ni mogoče povečati, ostaja f nespremenjena); torej jo lahko predstavimo tako, da navedemo g-je, pri katerih se njena vrednost poveča, in za vsak g navedemo še novo vrednost funkcije. Operacijo max med funkcijami lahko izvedemo z zlivanjem teh seznamov. Pri naši nalogi pa je stvar še toliko lažja, ker so možne vrednosti g le cela števila $\{0, \ldots, d\}$ (in $d \leq 100$), tako da lahko predstavimo vsako tako funkcijo kar s tabelo sto elementov.

Kot ponavadi pri dinamičnem programiranju je tudi tu zelo pomembno, da si rešitve podproblemov, torej funkcije $f_{n,m}(g)$ za razne pare (n, m), ko jih enkrat izračunamo, nekje zapomnimo, ker bodo kasneje prišle še prav in bi bilo škoda, če bi jih morali takrat računati ponovno. Iz gornje formule lahko opazimo, da pri izračunu $f_{n,m}$ potrebujemo le $f_{n-1,m}$ in $f_{n-1,m-1}$; zato je koristno računati te funkcije po naraščajočih n in pri vsakem n-ju po naraščajočih m. Tako imamo vedno pri roki rešitve podproblemov, ki jih potrebujemo, rezultate za $n - 2$ in manj knjig pa lahko sproti pozabljamo.

```
program Knjige;

  procedure Max(var a: integer; b: integer);
    begin if b > a then a := b end;

const MaxN = 1000; MaxM = 100; MaxD = 100;
var T: text;
    n, m, d, ni, mi, g, b, bb: integer;
    di: array [1..MaxN] of integer;
    f: array [1..MaxM, 0..MaxD] of integer;
begin
  Assign(T, 'knjige.in'); Reset(T); ReadLn(T, n, m, d);
  for ni := 1 to n do Read(T, di[ni]);
  Close(T);

  { 0 knjig. }
  for mi := 1 to m do for g := 0 to d do f[mi, g] := 0;

  { Dodajajmo knjige. }
  for ni := 1 to n do begin
    b := 0; { Najboljši rezultat za ni − 1 knjig in 0 polic. }
    for mi := 1 to m do begin

      { ni knjig in mi polic. Ko računamo za nek g, bomo potrebovali rezultat
        za ni − 1 knjig, mi polic in razne manjše g, tako da je koristno
        iti od večjih g proti manjšim, da si ne bomo sproti kvarili starih
        rezultatov, ki jih bomo še potrebovali. Zapomnimo si tudi najboljši
        rezultat za ni − 1 knjig in mi polic. }

      bb := f[mi, d];
      for g := d downto di[ni] do begin
        { Ena možnost je, da knjige ni sploh ne vzamemo.
          Potem je rezultat za ni knjig in mi polic enak kot za ni − 1 knjig
          in mi polic, ta pa je že v f[mi, g]. }
        { Lahko dodamo knjigo ni na polico mi, pri čemer je prej na njej
          že kakšna knjiga. Rešitve za ni − 1 knjig in mi polic
          in majhne g so trenutno še v f[mi]. }
        Max(f[mi, g], 1 + f[mi, g − di[ni]]);
        { Lahko dodamo knjigo ni na polico mi kot prvo.
          Najboljši rezultat za ni − 1 knjig in mi − 1 polic je trenutno v b. }
        Max(f[mi, g], 1 + b);
      end; {for g}
      b := bb; { Najboljši rezultat za ni − 1 knjig in mi polic. }

    end; {for mi}
  end; {for ni}

  Assign(T, 'knjige.out'); Rewrite(T); WriteLn(T, f[mi, d]); Close(T);
end. {Knjige}
```

REŠITVE NALOG PETEGA TEKMOVANJA IZ UNIXA

R2003.U.1

Naloga: str. 14

Pri tej nalogi nam bo prišel prav program `egrep`, ki prebere neko datoteko in izpiše le tiste njene vrstice, ki ustrezajo določenemu regularnemu izrazu. Pognali ga bomo dvakrat, najprej zato, da bo obdržal vrstice, ki ustrezajo prvemu izrazu, nato pa bomo te vrstice še enkrat pognali skozi `egrep` in obdržali le tiste izmed njih, ki *ne* ustrezajo drugemu izrazu (stikalo −v). Programu `egrep` lahko s stikalom −f povemo, naj regularni izraz prebere iz določene datoteke.

Naša skripta za `bash` lahko do parametrov, ki jih je dobila iz ukazne vrstice, dostopa prek spremenljivk $1, $2 in $3. To, če nek niz res predstavlja ime neke datoteke, lahko preverimo z operatorjem −f; vse tri pogoje združimo z operatorjem −a, ki pomeni logični in.

```
#!/bin/bash
if [ -f "$1" -a -f "$2" -a -f "$3" ]; then
  egrep -f "$2" "$1" | egrep -v -f "$3"
else
  echo "NAPAKA"
fi
```

R2003.U.2

Naloga: str. 14

Spodnja rešitev (v perlu) prebere kar celo datoteko v pomnilnik (njeno ime je prvi parameter iz ukazne vrstice in do njega pridemo z $ARGV[0]) in potem z regularnim izrazom preveri, če je v njej prisotna glava. Operator s/$vzorec_1$/$vzorec_2$/*zastavice* zamenja pojavitev vzorca 1 z vzorcem 2; v našem primeru pokrije vzorec 1 celo glavo, vzorec 2 pa je prazen in tako se glave znebimo. Zastavica s na koncu pa zahteva, naj obravnava interpreter cel niz kot eno samo dolgo vrstico; to potrebujemo, saj bi se utegnili znotraj glave pojavljati tudi znaki za konec vrstice. Pozorni moramo biti tudi na naslednje: glava se konča že pri prvi pojavitvi niza `data`; znak `*` v regularnem izrazu pa načeloma poskuša pokriti čim več besedila („požrešno ujemanje“, *greedy matching*), torej bi šel tu do zadnje pojavitve niza `data` v opazovanem nizu. Če hočemo, da pokrije čim manj besedila (torej le do prve pojavitve niza `data`), moramo za zvezdico postaviti še ?.

```
#!/usr/bin/perl
use strict;
use warnings;

open FH, $ARGV[0];
$_= join('', (<FH>));           # Preberemo celo datoteko.
close FH;

if (s/^RIFF.*?data//s) {        # Zbrišimo glavo, če je prisotna.
  open FH, '>', $ARGV[0];       # Če je bila glava prisotna, shranimo
  print FH;                     # preostanek podatkov nazaj v datoteko.
  close FH;
}
```

Naloga pravi, da če se datoteka ne začne na `RIFF`, naj program ne reže ničesar; ne pove pa, kaj storiti v primeru, če se začne na `RIFF`, vendar kasneje ne vsebuje niza `data`. Gornji program bi jo pustil pri miru; verjetno je to vendarle bolje, kot pa če bi pobrisali celo datoteko.

Morebitna slabost gornje rešitve je, da prebere celo datoteko v pomnilnik. To utegne biti nerodno, če je datoteka velika (kar ni pri multimedijskih datotekah nič neobičajnega). Za take primere bi bilo bolje, če bi datoteko brali po koščkih in vsebino, ki sledi nizu `data`, sproti prepisovali na začetek datoteke (podobno kot v rešitvi naloge 2003.U.4), na koncu pa bi datoteko skrajšali s funkcijo `truncate`.

Naloga: str. 14

R2003.U.3

Za vsak proces obstaja navidezni imenik `/proc/`*pid*, pri čemer je *pid* številka procesa. V tem imeniku je med drugim datoteka z imenom `exe`, ki je simbolna povezava na izvršilno datoteko tistega procesa. V lupini `bash` lahko prek spremenljivke `$PPID` dobimo številko procesa-očeta. Ime prave izvršilne datoteke, kamor kaže naša simbolna povezava, lahko izvemo od programa `ls`, če zahtevamo izčrpnejši izpis (stikalo `-l`). Vrstico, ki jo `ls` izpiše, razbijmo pri vseh presledkih (`cut -d ' '`); izkaže se, da je ime datoteke, kamor kaže simbolna povezava, potem ravno enajsta komponenta vrstice. Ker pa lahko ime vsebuje tudi presledke, je bolje izpisati vse komponente od enajste naprej (stikalo `-f 11-`). Težava je le ta, da `cut` prereže pri vsakem presledku, v izpisu programa `ls` pa je včasih po več presledkov skupaj in bi zato `cut` tam vmes ustvaril še neko število praznih komponent; to število je nepredvidljivo, ker ne vemo, koliko presledkov je `ls` vrinil zaradi poravnavanja stolpcev pri izpisu (odvisno je npr. od tega, koliko števk je porabil za izpis dolžine datoteke). Dobro bi bilo torej spremeniti vsako zaporedje presledkov v en sam presledek. To lahko naredimo s programom `tr`, če uporabimo stikalo `-s`. Druga možnost je, da si izpis programa `ls` shranimo v neko spremenljivko in jo podamo programu `echo`: iz posameznih komponent bodo nastali posamezni argumenti programu `echo`, interpreterju lupine pa je čisto vseeno, s koliko presledki so ločeni argumenti; `echo` bo med dvema argumentoma vedno izpisal en presledek.

```
#!/bin/bash
povezava=`ls -l /proc/$PPID/exe`
echo $povezava | cut -d ' ' -f 11-
```

ali pa

```
#!/bin/bash
ls -l /proc/$PPID/exe | tr -s ' ' | cut -d ' ' -f 11-
```

Gornja rešitev ima še majhno slabost: če se v imenu datoteke, na katero kaže opazovana simbolna povezava, kdaj pojavlja po več zaporednih presledkov, bo naš program tam izpisal en sam presledek, ker pač v `ls`-jevem izpisu nadomesti vsa zaporedja presledkov s po enim samim. Na srečo pa se v imenih datotek le redko pojavi več zaporednih presledkov.

Lahko bi si pomagali z dejstvom, da v izpisu programa `ls` pred imenom datoteke, na katero kaže simbolna povezava, stoji niz `"-> "`. S `sed`om lahko pobrišemo vse do vključno te puščice in presledka (stikalo `-n` je zato, da ne bo `sed` izpisal še prvotnega niza, kakršen je bil pred brisanjem):

```
#!/bin/bash
ls -l /proc/$PPID/exe | sed -n "s/^.*-> //p"
```

Slabost te rešitve je, da `sed` ujemanje z regularnimi izrazi preverja „požrešno“ (*greedy matching*) — zvezdica poskuša pobrati čim daljši kos niza. Če bi se torej v imenu očetovskega procesa pojavil niz `"-> "`, bi `sed` pobrisal še del tega imena, vse do zadnje pojavitve niza `"-> "`.

Še ena možnost je, da namesto `seda` uporabimo `awk`; na začetku nastavimo njegovo spremenljivko FS in mu s tem naročimo, naj vrstico, ki jo je izpisal `ls`, razreže pri vseh puščicah. Vrstica tako razpade na NF delov (i-tega dobimo v spremenljivki $i), mi pa moramo izpisati vse razen prvega.

```
#!/bin/bash
ls -l /proc/$PPID/exe | awk '
  BEGIN { FS = "-> " }
  {
    for (i = 2; i < NF; i++)
      printf "%s-> ", $i;
    print $NF;
  }'
```

Še lažje in bolj elegantno gre na primer v perlu, saj imamo funkcijo readlink, ki nam pove, kam kaže simbolna povezava. Očetovo številko dobimo s funkcijo getppid, nize pa stikamo z operatorjem . (pika).

```
#!/usr/bin/perl
print readlink("/proc/" . getppid . "/exe") . "\n";
```

R2003.U.4

Naloga: str. 15

Za stikanje datotek lahko uporabimo program `cat`. S programom `echo` mu pošljemo niz „`PREPIS`“ (brez znaka za konec vrstice, zato stikalo −n) in nato programu `cat` naročimo, naj stakne to, kar je prišlo s standardnega vhoda (−), z vsebino vhodne datoteke. Rezultat bi lahko zapisovali kar v izhodno datoteko, vendar pa bi to v primerih, ko sta vhodna in izhodna datoteka ena in ista, pomenilo, da bomo vhodne podatke najbrž izgubili, še preden bomo vse sploh prebrali. Zato raje uporabimo pomožno datoteko in jo potem preimenujmo v ime, ki smo ga dobili kot ime izhodne datoteke. Ime pomožne datoteke pripravimo s programom `mktemp`, ki znake `X` na koncu danega argumenta zamenja z naključnimi števkami in pazi na to, da datoteka s takšnim imenom še ne obstaja; dobljeno ime potem izpiše na svoj standardni izhod.

```
#!/bin/bash
pomozna=`mktemp "$2.XXXXXX"`
echo -n PREPIS | cat - "$1" > "$pomozna"
mv "$pomozna" "$2"
```

Slabost te rešitve je, da je včasih lahko potratna s prostorom. Če sta vhodna in izhodna datoteka različni, bi lahko pisali kar naravnost v izhodno datoteko, tako pa so tik pred klicem mv prisotne na disku vse tri: vhodna, pomožna (ki je dolga približno toliko kot vhodna, pravzaprav šest znakov daljša) in še stara izhodna. V primerih, ko sta vhodna in izhodna datoteka ena in ista, pa uporaba pomožne datoteke pomeni, da bosta pred klicem mv prisotni na disku dve kopiji vhodne (prvotna in tista pomožna z nizom „`PREPIS`“).

Varčnejša rešitev bi najprej preverila, če sta vhodna in izhodna datoteka ena in ista; če je res tako, naj odpre to datoteko za branje in pisanje obenem, nato pa prebira iz nje podatke po koščkih in se po vsakem branju pomakne po datoteki nazaj ter povozi ravnokar prebrane podatke s tistim, kar bo moralo biti na tem mestu zapisano v izhodni datoteki. Ker je vsebina izhodne datoteke v primerjavi z vsebino vhodne „zamaknjena“ za šest znakov (ker je v izhodni na začetku še niz „`PREPIS`“), nam vedno ostane šest znakov, ki smo jih že prebrali iz vhodne datoteke ter jih pri zadnjem pisanju tudi že povozili z drugimi podatki; te obdrži spodnji program v nizu `buf` in bodo prišli kot prvi na vrsto pri naslednjem pisanju v datoteko (takoj za naslednjim branjem).

```
import sys, os, os.path

def IstaDatoteka(ime1, ime2):
   if ime1 == ime2: return True
   # Mogoče pa sta to trdi povezavi na isto datoteko.
   st1 = os.stat(ime1); st2 = os.stat(ime2)
   return st1.st_dev == st2.st_dev and st1.st_ino == st2.st_ino

vhodIme = os.path.realpath(sys.argv[1])
pazi = False
if len(sys.argv) <= 2:
   # Izpisovali bomo na standardni izhod.
   izhod = sys.stdout
else:
   # Preverimo, če je izhodna datoteka ista kot vhodna.
   izhodIme = os.path.realpath(sys.argv[2])
   pazi = IstaDatoteka(vhodIme, izhodIme)
   if pazi:
      # Je — odprimo jo le enkrat, za branje in pisanje.
      # „vhod“ in „izhod“ bosta le dve referenci na isti objekt.
      vhod = file(vhodIme, "r+b"); izhod = vhod
   else:
      # Izhodna datoteka ni ista kot vhodna; odprimo izhodno za pisanje
      # (in uničimo morebitno obstoječo datoteko s tem imenom).
      izhod = file(izhodIme, "wb")
if not pazi:
   # Če sta vhodna in izhodna datoteka različni,
   # odprimo zdaj še vhodno, vendar le za branje.
   vhod = file(vhodIme, "rb")

buf = "PREPIS"
bufLen = 1024 * 1024
while len(buf) > 0:
   # Zapomimo si trenutni položaj v datoteki.
   if pazi: pos = izhod.tell()
   # Preberimo nekaj novih podatkov.
   buf = buf + vhod.read(bufLen)
   # Če je vhodna datoteka ista kot izhodna, se postavimo nazaj na položaj
   # „pos“, da bomo pri pisanju povozili pravkar prebrane podatke.
   if pazi: izhod.seek(pos)
   # Zapišimo nekaj podatkov.
   izhod.write(buf[:bufLen])
   # Če mešamo branja in pisanja nad isto datoteko, lahko pride včasih do težav,
   # npr. da vhod.read() vrne tisto, kar smo ravnokar zapisali na stari položaj,
   # namesto da bi prebral nove podatke. Rešitev je, da med pisanjem in branjem
   # pokličemo izhod.flush() ali pa vhod.seek(vhod.tell()).
```

```
if pazi: vhod.seek(vhod.tell())
# Obdržimo podatke, ki jih še nismo zapisali.
buf = buf[bufLen:]
```

Za ugotavljanje, če se dani imeni nanašata na eno in isto datoteko, smo uporabili najprej funkcijo realpath, ki sledi simbolnim povezavam; nato pa, če sta imeni tudi po tem različni, pogledamo za vsako ime par (st.st_dev, st.st_ino), ki enolično identificira posamezno datoteko (glej rešitev naloge 2001.U.3). Tako odkrijemo še primere, ko dobimo dve „trdi povezavi" na isto datoteko.

Viri nalog za leto 2003: pošiljanje sporočil — Andraž Bežek; vplivi — Uroš Jovanovič; napadalne kraljice — Mitja Lasič; dva kupa števil — Jure Leskovec; radar — Mark Martinec; „pet čevljev merim, palcev pet" — Mojca Miklavec; vodenje projektov — Blaž Novak; glasovanje, knjižnica — Anže Žagar; križanka, števke — Klemen Žagar; različnost nizov, smučarji — Janez Brank.

Hvala Juretu Leskovcu za implementacijo rešitve naloge s smučarji in Andreju Bauerju za pripombe k besedilu nalog. Citat pri tretji nalogi za tretjo skupino je iz 5. poglavja pete knjige *Bratov Karamazovih*.

Tekmovanje v poznavanju Unixa 2003 so pripravili: Aleš Košir, Saša Divjak, Gašper Fele-Žorž, Boris Gašperin, Jure Koren, Rok Papež, Primož Peterlin, Marko Samastur, Andraž Tori in Miha Tomšič.