# 21. državno tekmovanje v znanju računalništva (1997)

## NALOGE ZA PRVO SKUPINO

## 1997.1.1

Člani komisije računalniškega tekmovanja srednješolcev so pripravili podprogram Uredi za urejanje doseženih rezultatov in trdijo, da deluje. R: 12

```
const n = ...;
var a: array [0..n] of integer;

procedure Uredi;
var i, j: integer;
begin
  for i := 2 to n do begin
    j := i;
    a[0] := a[j];
    while (j > 1) and (a[j − 1] > a[0]) do begin
      a[j] := a[j − 1];
      j := j − 1;
    end;
    a[j] := a[0];
  end;
end; {Uredi}
```

**Opiši osnovno idejo** urejanja, ki je uporabljena v podprogramu Uredi. **Ugotovi**, ali je podpogoj (j > 1) v zanki **while** zares potreben, se pravi, ali so bili člani komisije pri sestavljanju podprograma Uredi preveč prizadevni in smejo prepisati pogoj v zanki

```
while (j > 1) and (a[j − 1] > a[0]) do begin
  a[j] := a[j − 1];
  j := j − 1;
end;
```

v pogoj

```
while a[j − 1] > a[0] do begin
  a[j] := a[j − 1];
  j := j − 1;
end;
```

R: 12

## 1997.1.2

V podani tabeli Tabela je Dolz števil ($0 \leq \text{Dolz} \leq 100$), preostanek tabele pa je neuporabljen.

```
var Tabela: array [1..100] of integer;
    Dolz: integer;
```

**Napiši program**, ki tabelo spremeni tako, da v primeru zaporednih enakih števil odstrani iz tabele vse zaporedne kopije. Na koncu mora biti v spremenljivki Dolz shranjena nova dolžina tabele.

Primer:

| | | |
|---|---|---|
| pred brisanjem: | $3, 5, 9, 9, 1, 1, 1, 2, 3, 9, 9, 9$ | $\text{Dolz} = 12$ |
| po brisanju kopij: | $3, 5, 9, 1, 2, 3, 9$ | $\text{Dolz} = 7$ |

R: 12

## 1997.1.3

Pri zapisovanju slovarja slovenskega jezika na CD-ROM (speštanka) je dr. Ostropišič opazil, da ima veliko število besed enake končnice. Odločil se je, da bo izkoristil to lastnost in s primernim kodiranjem prihranil nekaj prostora.

Kot jezikoslovec sam tega seveda ne zmore, zato mu pomagaj. **Opiši postopek**, ki bo v seznamu besed poiskal takšno končnico besed, pri kateri bo zmnožek dolžine končnice in števila ponovitev besed, katerim je ta končnica skupna, največji. Če je takih končnic več, poišči katerokoli izmed njih.

Primer:

| | | |
|---|---|---|
| `BOLAN` | | |
| `BONBON` | Končnica | `N` se ponovi 5-krat, zmnožek je 5. |
| `SALON` | Končnica | `ON` se ponovi 4-krat, zmnožek je 8. |
| `ZAKLON` | Končnica | `LON` se ponovi 3-krat, zmnožek je 9. |
| `PRIKLON` | Končnica | `KLON` se ponovi 2-krat, zmnožek je 8. |
| `SKLON` | | |

Pozor: končnica `ON` se *pojavi* petkrat, a se *ponovi* le štirikrat. V zgornjem primeru je pravilni odgovor `LON`.

Seznam besed je zapisan v tabeli tako, da je v vsakem elementu po ena beseda. V tabeli je veliko število ($> 100$) kratkih besed ($< 10$ znakov). Predpostavi, da lahko vse besede shraniš v pomnilnik. Besede so že urejene po abecednem redu, vendar od konca besede proti začetku, kot to kaže tudi zgornji primer.

R: 15

## 1997.1.4

Ponudniki dostopa do Interneta (npr. slovenski ARNES in ameriški America On-Line), se vedno znova srečujejo s pomanjkanjem modemskih vstopnih linij. Izkušnje ameriških ponudnikov kažejo, da razmerje med številom uporabnikov in številom modemov ne sme presegati

deset uporabnikov na en modem. Kadar je to razmerje preseženo, so modemi neprestano zasedeni.

Ponudnik dostopa do Interneta BUTALE BBS ima na voljo $N$ modemov. Razmerje med številom uporabnikov in številom modemov je približno $42 : 1$.

**Napiši program**, ki nadzoruje uporabo modemov tako, da vedno obdrži en modem prost. V primeru, ko bi moral zasesti zadnji prosti modem, program sprosti tisti modem, ki je bil do takrat najdlje zaseden. Modemi so oštevilčeni od 1 do $N$. Na razpolago imaš podprogram Zaseden(i), ki pove, koliko sekund je modem i že zaseden. Če je modem prost, podprogram vrne 0. Modem i sprostimo s klicem podprograma Sprosti(i).

Ali je ta rešitev primerna? Kakšne težave lahko predvidiš? Odgovor utemelji in predlagaj boljšo rešitev.

## NALOGE ZA DRUGO SKUPINO

### 1997.2.1

Programer Jaša je našel program, ki ga je napisal pred davnimi leti. Žal se ne spomni, kaj je hotel z njim reševati. Spomni pa se, da z njim nekaj ni bilo v redu. Seveda mu boš pomagal ti. R: 17

(a) Razloži, **kaj dela ta program**.

(b) Skrajšaj program.

(c) Za katere vhodne podatke program deluje nepravilno?

(d) Navedi primer vhodnih podatkov, za katere program izpiše dvakrat 42.

Naj samo še spomnimo, da operator $\sim$ v C-ju oz. **not** v pascalu obrne vse bite v dvojiški predstavitvi števil.

```
#include <stdio.h>
int main()
{
  int n1, n2, c, d, e, f;
  printf("Vnesi 1. število:");
  scanf("%d", &n1);
  printf("Vnesi 2. število:");
  scanf("%d", &n2);
  c = d = 0;
  e = f = 1;
  if (n1 < 0) {
    n1 = -n1;
    e = f = -1;
  }
  if (n2 < 0) {
    n2 = -n2;
```

```
program Main(Input, Output);
var
  n1, n2, c, d, e, f: integer;
begin
  Write('Vnesi 1. število: ');
  ReadLn(n1);
  Write('Vnesi 2. število: ');
  ReadLn(n2);
  c := 0; d := 0;
  e := 1; f := 1;
  if n1 < 0 then begin
    n1 := -n1; e := -1; f := -1;
  end;
  if n2 < 0 then begin
```

```
    e = -e;
  }
  do {
    if (c % 2)
      n1 = n1 - n2;
    else
      n1 = n1 + ~n2 + 1;
    c += 1;
  } while (n1 >= n2);
  if (n1 < 0) d = 1;
  printf("%d %d\n",
    (c - d) * e, (n1 + n2 * d) * f);
}
```

```
    n2 := -n2; e := -e;
  end;
  repeat
    if Odd(c) then
      n1 := n1 - n2
    else
      n1 := n1 + not n2 + 1;
    c := c + 1;
  until n1 < n2;
  if n1 < 0 then d := 1;
  WriteLn((c - d) * e, ' ', (n1 + n2 * d) * f);
end.
```

R: 19

## 1997.2.2

S skrivanjem sporočil pred nepooblaščenimi pogledi se ukvarjata dve vedi: kriptografija in steganografija. Kriptografija poskuša zagotoviti takšno šifriranje dokumentov, da jih nasprotnik ne more prebrati, tudi če prestreže šifrirani dokument. Steganografija poskuša skriti vsebino dokumenta tako, da se nasprotnik sploh ne zaveda, da ima v rokah skriti dokument. V praksi se večkrat uporabljata obe metodi hkrati: šifrirani dokument še skrijemo.

Ena od možnosti za skrivanje dokumenta je, da ga skrijemo v nek drugi večji dokument, na primer v sliko ali v zvočni zapis. Pri tem večji dokument nekoliko pokvarimo, vendar na čim manj opazen način.

Vzemimo za večji dokument črno-belo sliko velikosti $256 \times 256$ točk (pikslov). Svetlost vsake točke je predstavljena s številom med 0 in 255. Če spremenimo najnižji bit, se svetlost točke spremeni kvečjemu za $1/256$, kar je manj kot 0,4 %. Tega s prostim očesom ne opazimo ali kvečjemu zaznamo kot povečan šum v sliki. V sliko lahko tako skrijemo $(256 \times 256)/8 = 8192$ osem-bitnih znakov.

```
const w = 256; h = 256;
var Slika: array [1..h, 1..w] of 0..255;
```

**Napiši podprogram** SkrijBesedilo, ki bo v sliko v tabelo Slika vpisal (oz. skril) besedilo, dolgo 8192 znakov, ter podprogram RazkrijBesedilo, ki bo iz slike izluščil skrito besedilo in ga izpisal.

Da ne zapletamo programa s kodiranjem koncev vrstic in se izognemo preverjanju konca vhodne datoteke, predpostavimo, da je v vhodni datoteki le ena vrstica besedila, dolga 8192 znakov.

R: 20

## 1997.2.3

**Napiši funkcijo** PrestejPodnize, ki mora vrniti število pojavitev niza PodNiz v nizu Niz. Štejejo tudi nestrnjene pojavitve, torej je lahko med črkami PodNiz-a v Niz-u tudi poljubno število drugih črk. Funkcija PrestejPodnize naj ima obliko

```
function PrestejPodnize(Niz, PodNiz: string): integer;
```

ali

```
int PrestejPodnize(char* Niz, char* PodNiz);
```

Primeri: v nizu `deafgahibjkclm` nastopajo podnizi: `abc` dvakrat (zaradi dveh `a`-jev), `bc` enkrat, `afg` enkrat in `ba` nobenkrat. V `deafgabhibjkclm` nastopa `abc` štirikrat, v `deafbgahibjkclm` pa trikrat. Niz `abcdefghij` nastopa v nizu `aabbccddeeffgghhiijj` 1024-krat.

## 1997.2.4

Celota je razdeljena na $N$ deležev, ki so realna števila med 0 in 1, njihova vsota pa je 1. Deleži so podani v tabeli: R: 25

```
const N = 20;    { N je poljuben, a ne zelo velik }
var Delez: array [1..N] of real;
```

Radi bi izpisali deleže v odstotkih, zaokrožene na cela števila. Pri tem nastane lepotni problem, saj se lahko zgodi, da vsota zaokroženih odstotkov ni točno 100.

Problemu se izognemo z „goljufanjem“ pri zaokrožanju. Odstotke zaokrožimo tako, da je skupna vsota natanko 100, skupna absolutna napaka pa čim manjša. Goljufajmo torej pri tistih deležih, ki se jim goljufija manj pozna.

Primer za $N = 5$:

| delež · 100 | zaokrožen | | po popravku |
|---|---|---|---|
| 30,9 | 31 | | 31 |
| 10,4 | 10 | $\longrightarrow$ | 11 |
| 20,3 | 20 | | 20 |
| 20,2 | 20 | | 20 |
| 18,2 | 18 | | 18 |
| 100,0 | 99 | | 100 |

Na primer, 10,4 lahko zaokrožimo na 10 ali na 11, kakor nam pač bolj ustreza. V prvem primeru je absolutna napaka 0,4, v drugem pa 0,6.

**Napiši del programa**, ki v tabeli podane deleže izpiše kot zaokrožene odstotke. Program naj poskrbi, da je vsota zaokroženih odstotkov 100, skupna absolutna napaka pa najmanjša.

## NALOGE ZA TRETJO SKUPINO

R: 26

# 1997.3.1

Ker se člani komisije računalniškega tekmovanja srednješolcev raje ukvarjajo s trapastimi podatkovnimi strukturami kot s sestavljanjem pametnih nalog, so skovali funkcijo DvojnaHitrost, sedaj pa ne vedo, kaj sploh dela. Ugotovi, **kaj vrne funkcija** DvojnaHitrost, in oceni, **kolikokrat** se v najslabšem primeru izvede telo zanke **repeat**, če seznam s vsebuje $n$ elementov. Člani komisije ti bodo za pravilno rešitev naloge zares hvaležni.

```
type
  Seznam = ↑Vozel;
  Vozel = record
            Naslednji: Seznam;
            ... podatki ...
          end;

function DvojnaHitrost(S: Seznam): boolean;
var p1, p2: Seznam;
begin
  p1 := S;
  p2 := S;
  repeat
    if p1 <> nil then p1 := p1↑.Naslednji;
    if p2 <> nil then p2 := p2↑.Naslednji;
    if p2 <> nil then p2 := p2↑.Naslednji;
  until (p1 = nil) or (p2 = nil) or (p1 = p2);
  DvojnaHitrost := (p1 <> nil) and (p2 <> nil);
end; {DvojnaHitrost}
```

R: 27

# 1997.3.2

Novoizvoljeni župan Gropel kraja Kraljana je želel izvedeti, kateri ljudje v njegovem mestu so najpomembnejši. Nekako mu je v roke prišel seznam meščanov (v datoteki `ligenj.doc`), ki so med seboj prijatelji.

Župan je sklepal takole: moč meščana je enaka številu njegovih prijateljev. Začetna pomembnost vsakega meščana je ena. Pomembnost meščanov se prenaša z enega na drugega tako, da vsak meščan, ki ima močnejše prijatelje, svojo pomembnost enakomerno razdeli mednje.[1] Njegova pomembnost pri tem postane nič. Najpomembnejše meščane dobimo, ko prenašanje pomembnosti med meščani ni več možno.

[1] Pozor: „moč“ in „pomembnost“ sta tu dva popolnoma ločena pojma. Pomembnost se prenaša med meščani, moč pa ostaja ves čas nespremenjena. Moč samo vpliva na to, kako se prenaša pomembnost.

**Napiši algoritem**, ki čim učinkoviteje določi pomembnosti meščanov in izpiše številke meščanov z neničelno pomembnostjo. Pri tem imaš na voljo naslednje funkcije in podprograme:

- **function** PreberiMescane: integer;
  Prebere podatke o meščanih in vrne njihovo število.
- **procedure** Tekoci(m: integer);
  Meščan m postane tekoči meščan.
- **function** Naslednji: integer;
  Vrne naslednjega prijatelja tekočega meščana ali 0, če so vsi pregledani.

Kaj veš povedati o rešitvah naloge?

## 1997.3.3

R: 29

Program BIOS (Basic Input/Output System) nekega računalnika za shranjevanje sistemskih nastavitev uporablja pomnilniški čip tipa *Flash* ROM. Tovrstni čipi obdržijo vpisano informacijo tudi po izklopu napajanja. Sprva je čip prazen in na njem so zapisane same ničle. V čip lahko vpisujemo le enice. Brisanje (vpisovanje ničel) je sicer možno, a dolgotrajno in skrajša življenjsko dobo čipa, zato ga ne bomo uporabljali.

BIOSu je čip predstavljen kot niz podatkovnih besed:

```
const MaxFlashROM = ...;
var FlashROM: array [1..MaxFlashROM] of integer;
```

BIOS vpisuje cela števila v čip s podprogramom Vpisi, pri čemer se vpišejo samo enice:

```
procedure Vpisi(Naslov, Podatek: integer);
begin
  FlashROM[Naslov] := FlashROM[Naslov] or Podatek;
end; {Vpisi}
```

Ob vsakem vpisovanju zapiše BIOS na čip najprej velikost zapisa ($N$), temu pa sledi $N-1$ besed (integerjev) podatkov. Zapisi se nizajo eden za drugim, zadnjemu zapisu pa sledi nepopisano področje samih ničel.

Ob vsakem zagonu sistema se zagonski program sprehodi prek vseh zapisov do zadnjega in ga prebere. Število zapisov se med delovanjem sistema veča, zato program za iskanje zadnjega zapisa porabi vse več časa.

Vaša naloga je, da **izdelate učinkovit algoritem**, ki poskuša podatke v čipu Flash ROM spremeniti tako, da bo zagonski program čim hitreje našel zadnji zapis. Pri tem sme spreminjati samo informacijo o velikosti posameznih zapisov.

R: 30

## 1997.3.4

Podjetje Domača pamet—Tuje ideje je ugotovilo, da njihovi zaposleni pogosto zahtevajo ene in iste spletne strani, ki se vedno znova prenašajo prek njihove povezave.

Zato so se odločili, da bodo naredili strežnik, ki bo zahteve prestregel, pogledal, če je stran že bila zahtevana, in jo v tem primeru podal kar iz vmesnega pomnilnika na disku. Najeli so te, da **napišeš dva podprograma**:

Zahtevek(Naslov: string; Tok: integer), ki bo poklican vsakič, ko bo kateri od uporabnikov zahteval stran, in

Prispelo(Podatki: string; Tok: integer), ki bo poklican vsakič, ko preko linije iz Interneta pridejo podatki za zahtevo, povezano s tem tokom. Če je podatkov konec (sprejeti so bili vsi podatki), bo Dolzina(Podatki) enako 0, sicer bo Dolzina(Podatki) $> 0$.

Vsi prenosi podatkov potekajo v tokovih. Vsak tok je označen s pozitivnim celim številom. Iz vmesnega pomnilnika bodo na tok vedno prišli vsi podatki iz zahteve (ali pa nič), na druge tokove pa lahko podatki prihajajo „po kapljah“, vsakič po nekaj zlogov (byteov). Tip string je zagotovljeno dovolj velik za vsako zahtevo.

Na voljo imaš naslednje funkcije:

Zahtevaj(Naslov: string): integer — Sproži zahtevo in vrne številko toka.

Poslji(Podatki: string; Tok: integer) — Na tok Tok pošlje podatke.

Shranjen(Naslov: string; **var** Podatki: string): integer — Če so podatki v vmesnem pomnilniku, vrne 0 in podatke v parametru Podatki. Če podatkov v vmesnem pomnilniku ni, vrne številko toka, na katerega naj pošljemo podatke, da se bodo shranili v vmesni pomnilnik.

Vedeti je treba, da bo pri polni obremenitvi strežnik moral ustreči nekaj tisoč zahtevam na uro.

## TRETJE ZAKLJUČNO TEKMOVANJE V ZNANJU RAČUNALNIŠTVA

### Navodila za obnašanje med tekmovanjem

Naloga tekmovalcev je napisati tri programe, ki iz danih vhodnih podatkov, zapisanih v vhodni datoteki `input.txt`, za dano nalogo izračunajo pravilen rezultat in ga zapišejo v datoteko `output.txt`. Čas reševanja bo 5 ur, pred začetkom reševanja pa bo imel vsak tekmovalec vsaj 15 minut časa za prilagajanje delovnega okolja za računalnikom.

Na vsakem računalniku se nahaja direktorij `c:\rtk\`, ki vsebuje poddirektorija `borlandc` (Borland C/C++ 3.1) in `bp` (Borland Pascal 7.0) in ukazno

datoteko `rtk.bat`, ki nastavi poti za zagon prevajalnikov. Po zagonu `rtk.bat` se Borland C požene z ukazom `bc`, Borland Pascal pa z `bp`.

Tekmovalci smejo delati le na direktoriju `c:\rtk\` in nižje, nikakor pa ne smejo brati in pisati datotek drugje, ker gre za računalnike, na katerih baje teče izobraževalni proces.

Med tekmovanjem niso dovoljeni pogovori med tekmovalci. Tekmovalci ne smejo imeti knjig ali disket. Ni dovoljeno goljufati na vse mogoče standarne in nestandardne načine. Tekmovalcem ni dovoljeno uporabljati drugih funkcij računalnika (mreže, programov, ipd.) razen tistih, ki so predvidene (DOS okno na direktoriju `c:\rtk\` in programa `bc` in `bp`). Glasnost tipkanja naj ne presega 40 dB.

Vsako nepravilnost bodo člani komisije kaznovali s takojšnjo ali naknadno diskvalifikacijo tekmovalca.

**Navodila za oddajo rezultatov**

Tekmovalci morajo rezultate zapisati na dve disketi, ki ju pred uradnim koncem tekmovanja oddajo komisiji. Po uradnem koncu komisija ne bo več sprejemala disket. Na vsaki disketi mora biti napisano ime tekmovalca.

Na disketi za oddajo komisiji se morajo nahajati 3 datoteke:

- `ceste.exe` — rešitev prve naloge,
- `tocke.exe` — rešitev druge naloge,
- `km.exe` — rešitev tretje naloge

in izvorna koda programov!

Naj še enkrat spomnimo, da morajo vsi trije programi brati podatke iz datoteke `input.txt`, rezultate pa morajo napisati v datoteko `output.txt`.

**Ocenjevanje rešitev**

Tekmovanju bo sledilo javno ocenjevanje, med katerim bo komisija vsako rešitev preverila z 10 različnimi vhodnimi datotekami (`input.txt`). Uspešna rešitev za dano vhodno datoteko bo tista, ki bo v 10 sekundah vrnila pravilen rezultat.

## Naloge

### 1997.Z.1

R: 32

V deželi *Transalpeniji* gradijo cestno omrežje, ki naj bi povezovalo vsa deželna mesta. **Napiši program**, ki ugotovi, koliko medsebojno še nepovezanih skupin mest je v deželi, in rezultat izpiše.

Vsakemu mestu pripišemo število med vključno 1 in $M$. Neposredne povezave med njimi so podane v obliki parov $(i, j)$, kar pomeni, da je mesto $i$ povezano z mestom $j$. Povezave so dvosmerne: če je mesto $i$ povezano z mestom $j$, je tudi mesto $j$ povezano z mestom $i$.

Dve mesti sta povezani, če obstaja med njima zaporedje povezav, ki vodi iz enega v drugo mesto. Skupina mest ni povezana z drugo skupino mest, če nobeno mesto iz prve skupine ni povezano z nobenim mestom iz druge skupine.

Podatki o povezavah med mesti se nahajajo v datoteki `input.txt`, kjer se v prvi vrstici nahaja število $M$, v drugi vrstici število $N$, ki predstavlja število vseh neposrednih povezav med mesti, v nadaljnjih $N$ vrsticah pa se nahajajo pari števil, ki predstavljajo neposredne povezave med mesti. Povezave se lahko ponovijo. Povezava lahko povezuje mesto s samim seboj.[2]

Program mora v datoteko `output.txt` izpisati število medsebojno nepovezanih skupin mest. V skupini mest mora biti vedno vsaj eno mesto.

Primer povezane skupine treh mest v datoteki `input.txt`:

```
3
3
1 2
2 3
3 1
```

Pravilni odgovor v datoteki `output.txt` je `1`.

Primer nepovezane skupine štirih mest v datoteki `input.txt`:

```
4
2
1 2
3 4
```

Pravilni odgovor v datoteki `output.txt` je `2`.

R: 34

## 1997.Z.2

Na ravnini je danih $n$ točk. Vsaka točka je podana s parom koordinat $(x, y)$. **Napišite program**, ki izračuna največje število točk, ki ležijo v nekem pravokotniku velikosti $a \times b$, pri čemer rotiranje pravokotnika ni dovoljeno (stranici pravokotnika morata torej biti vzporedni koordinatnima osema).

Program naj iz datoteke `input.txt` prebere vhodne podatke po naslednjem zaporedju:

$$
\begin{array}{l}
a\ b \\
n \\
x_1\ y_1 \\
x_2\ y_2 \\
\ldots\ldots \\
x_n\ y_n
\end{array}
$$

Predpostavite, da število točk, torej $n$, ne bo večje od 10 000. Kot rezultat naj program izpiše v datoteko `output.txt` največje število točk, ki ležijo v pravokotniku velikosti $a \times b$.

[2] V prvotnem besedilu naloge ni bilo podane nobene omejitve glede velikosti števil $M$ in $N$, kar ni ravno lepo. Pogled na testne primere, na katerih so preizkušali rešitve tekmovalcev, pokaže, da sta bila $M$ in $N$ vedno med vključno 1 in 30 000. Poleg tega se v dveh od desetih testnih primerov pojavlja kot številka mesta tudi 0; tedaj so torej mesta oštevilčena od 0 do $M$, drugače pa od 1 do $M$.

Primer vhodne datoteke:

```
3.0 2.0
6
0.0 0.0
1.0 0.0
1.0 1.0
4.0 1.0
0.0 2.0
1.0 3.0
```

Pripadajoča izhodna datoteka:

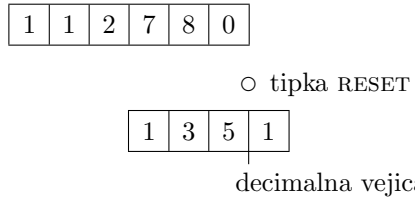
```
4
```

## 1997.Z.3

R: 45

Borut se pogosto vozi z avtomobilom. Čeprav je vzoren voznik, mu pogled vendarle pogosto zaide na števec kilometrov. Pri tem si je zamislil svojevrsten problem, ki pa ga ni znal rešiti, zato vas prosi za pomoč.

Števec je sestavljen iz dveh delov (velikega in malega števca). Veliki števec je sestavljen iz šestih števk in kaže skupne prevožene kilometre avtomobila. Mali števec ima 4 števke, pri čemer zadnja kaže desetine kilometra. Mali števec lahko v poljubnem trenutku postavimo na 000,0 s pritiskom na tipko RESET.




Naloga je **narediti program**, ki zna pri poljubnem stanju obeh števcev doseči stanje, kjer je vseh 10 števk obeh števcev med seboj različnih, tako da prevozimo čim manj kilometrov. Pri tem lahko poljubnokrat pritisnemo tipko RESET.

Vhodna podatka sta začetno stanje velikega in malega števca in sta zapisana v datoteki `input.txt` vsak v svoji vrstici. Primer:

```
000000
000.0
```

Privzemimo, da sta oba števca ravnokar dosegla svojo vrednost, tako da moramo prevoziti natanko 1 km, da se obrne veliki števec, oziroma 0,1 km, da se obrne mali.

Rezultat naj bo število kilometrov na velikem števcu, ko bo prvič mogoče doseči (lahko tudi takoj), da so vse števke različne. Izpiše naj se v datoteko `output.txt`. Rezultat za gornji primer je:

```
012345
```

## REŠITVE NALOG ZA PRVO SKUPINO

N: 1

# R1997.1.1

Podprogram Uredi uredi n elementov tabele a na intervalu 1..n z algoritmom navadnega vstavljanja s čuvajem. To pomeni, da v vsaki ponovitvi zanke **for** vstavi vrednost, ki je na začetku izvajanja procedure v elementu a[i], na pravo mesto med vrednosti a[1], a[2], ..., a[i − 1] in s tem poveča urejeni del tabele na interval a[1], a[2], ..., a[i]. Postopek se konča, ko vstavi vrednost elementa a[n] in s tem doseže urejenost cele tabele a.

Podpogoj j > 1 prvič ni izpolnjen v trenutku, ko velja j = 1. A ker vrednost elementa a[i], ki jo vstavljamo v že urejeni del tabele, med vstavljanjem hranimo v elementu a[0], tudi podpogoj a[j − 1] > a[0] pri j = 1 ni izpolnjen, zanka **while** pa se zato pri j = 1 gotovo ustavi. To pomeni, da lahko podpogoj j > 1 odstranimo.

N: 2

# R1997.1.2

S števcem f se sprehajamo po tabeli, novo stanje tabele pa pri tem postopoma nastaja v prvih celicah tabele (Tabela[1..t]). Vsak element (razen prvega) primerjamo s prejšnjim elementom; če je enak, se zanj ne zmenimo, drugače pa ga dodamo na konec nove tabele, torej na mesto Tabela[t + 1]. Ker se f poveča v vsakem koraku, t pa le občasno, je f vedno večji ali enak t, tako da nam ni treba skrbeti, da bi si povozili kakšne podatke, ki jih še nismo prebrali.

```
var Tabela: array [1..100] of integer;
    Dolz: integer;

procedure OdstranjevanjeDuplikatov;
var f, t: integer;
begin
  if Dolz > 1 then begin
    t := 1;
    for f := 2 to Dolz do
      if Tabela[t] <> Tabela[f] then
        begin t := t + 1; Tabela[t] := Tabela[f] end;
    Dolz := t;
  end; {if}
end; {OdstranjevanjeDuplikatov}
```

N: 2

# R1997.1.3

Problem lahko rešimo z dvema gnezdenima zankama. V zunanji zanki se premikamo po besedah, v notranji pa za besede od trenutne besede naprej gledamo, v koliko zadnjih črkah se ujemajo s trenutno. Ko pridemo do take, ki se s trenutno ne ujema niti v eni zadnji črki, lahko nehamo, saj so besede urejene po končnicah in tudi v bodoče ne bi našli nobene besede, ki bi se s trenutno ujemala v zadnjih nekaj črkah.

Še ena izboljšava je, da ne gledamo besed od trenutne naprej, ampak upoštevamo še, koliko besedam mora biti skupna neka na novo odkrita končnica, če hoče biti boljša od doslej najboljše znane. Na primer, če ima najboljša doslej znana končnica zmnožek 55, naše besede pa so dolge po največ deset znakov, bo vsaka končnica, ki je skupna vsaj dvema besedama, dolga največ devet znakov, torej mora biti skupna v resnici vsaj $\lceil 56/9 \rceil + 1 = 8$ besedam, če naj ima sploh kaj možnosti, da postane boljša od najboljše možne. Torej ni potrebno primerjati trenutne besede kar takoj z naslednjo, temveč jih lahko šest preskočimo in šele sedmo naslednjo primerjamo s trenutno. (Nobena končnica, ki bi bila skupna manj kot toliko besedam, namreč nima možnosti dobiti zmnožka nad 55.) Če imata tidve skupno neko končnico, imajo tudi vse med njima to končnico, saj so besede urejene po končnicah.

```
const n = ...; MaxDolz = 10;
type Tabela = array [1..n] of string;

function Koncnica(s: Tabela): string;
var i, j, k, Zmnozek, NajZmnozek: integer;
begin
  NajZmnozek := 0; Koncnica := s[1];
  for i := 1 to n do begin
    j := i + ((NajZmnozek + 1) + (MaxDolz - 1) - 1) div (MaxDolz - 1);
    if j > n then break; { odslej bi vedno imeli j > n }
    while j <= n do begin
      k := 0;
      while (k < Length(s[i])) and (k < Length(s[j])) do
        if s[i, Length(s[i]) - k] = s[j, Length(s[j]) - k] then k := k + 1
        else break;
      if k = 0 then { Besede od s[j] naprej nimajo s s[i] skupne }
        break;      { nikakršne končnice več. Takoj se lahko lotimo naslednjega i. }
      { Besedam s[i..j] je skupna končnica dolžine k. }
      Zmnozek := k * (j - i);
      if Zmnozek > NajZmnozek then begin
        { To je najboljša doslej znana končnica — zapomnimo si jo. }
        Koncnica := Copy(s[i], Length(s[i]) - k + 1, k);
        NajZmnozek := Zmnozek;
      end; {if}
      j := j + 1;
    end; {while}
  end; {for}
end; {Koncnica}
```

Slabost tega postopka je, da je lahko število parov besed (torej število parov $(i, j)$), ki jih moramo pregledati, če imamo smolo, sorazmerno s kvadratom števila vseh besed. Zato je časovna zahtevnost tega postopka $O(n^2)$, kar je lahko neugodno, če je besed veliko. Primer neugodnega zaporedja besed je na

primer zaporedje vseh $n = 2^m$ besed, dolgih po $m$ znakov in sestavljenih iz samih črk $a$ in $b$. (Za $m = 3$ bi dobili *aaa, baa, aba, bba, aab, bab, abb, bbb.*) Končnica dolžine $k$ je skupna $2^{m-k}$ besedam in zato dobi zmnožek $(2^{m-k}-1)\cdot k$. Najboljša je zato kar končnica dolžine 1,[3] ki bi jo naš program opazil že pri $i = 1$. Odtlej je torej NajZmnozek enak $2^{m-1} - 1$, zato pri vsakem naslednjem $i$ preizkušamo vrednosti $j$ od $i + \lceil 2^{m-1}/(m-1) \rceil \le i + 1 + 2^{m-1}/(m-1)$ naprej. Dokler je $i$ v prvi polovici tabele, torej $i \le 2^{m-1}$, bo šla notranja zanka do $j = 2^{m-1} + 1$ (do prve besede, ki se konča na $b$ namesto na $a$). Za dovolj velike $m$, na primer $m \ge 5$, je $2^{m-1}/(m-1) \le 2^{m-1}/4 = 2^m/8$. Dokler je $i$ še v prvi osmini tabele ($i \le 2^m/8$), mora $j$ torej obiskati vsaj vse nize od $i + 1 + 2^m/8 \le 2^m/4 + 1$ do konca prve polovice zaporedja, torej do vključno $2^m/2$. To pa je vsaj $2^m/4$ nizov in to se nam zgodi pri vsaj $2^m/8$ vrednostih $i$ (tudi pri $i = 1$, saj tam obiščemo celo vseh $2^m$ nizov). Že zaradi tega je število izvedb prireditve k := 0 vsaj $(2^m/4)(2^m/8) = n^2/32$. Časovna zahtevnost tega programa je torej vsaj $O(n^2)$.

Reševanja tega problema se lahko lotimo tudi drugače in bolj učinkovito. Recimo, da imamo besede že urejene po abecedi (od konca besede naprej) v neki tabeli $s[1..n]$ in da za besedo $s[i]$ vemo, da ima zadnjih $j$ črk skupnih z besedami $s[i-1], \ldots, s[i-g[i,j]+1]$, ne pa tudi z besedo $s[i-g[i,j]]$. Potem so za vsak $j$ (od 1 do dolžine besede $s[i]$) zmnožki $j \cdot (g[i,j]-1)$ vsekakor kandidati za najboljši zmnožek, ki ga moramo pri naši nalogi poiskati. Če torej izračunamo $g[i,j]$ za vse $i$ in $j$, ne bo težko poiskati najboljšega zmnožka. Lepo pa je to, da lahko $g[i,j]$ učinkovito računamo, če že poznamo $g[i-1,j]$: če se besedi $s[i]$ in $s[i-1]$ ujemata v zadnjih $j$ črkah, je $g[i,j] = g[i-1,j]+1$, drugače pa je $g[i,j] = 1$ (ta možnost obvelja tudi v primeru, če je $s[i-1]$ krajša od $j$ črk, in v primeru, ko je $i = 1$). Zato tudi ni treba hraniti tabele $g[i,j]$ za vse vrednosti $i$, pač pa le za trenutni $i$ in za $i-1$ (medtem ko jo za $i$ še računamo), prejšnje pa lahko sproti pozabljamo. Najboljši doslej najdeni zmnožek si zapomnimo kot b, indeks niza, kjer se ta končnica pojavlja, v bi, njeno dolžino pa v bj.

```
const n = ...; MaxDolz = 10;
type Tabela = array [1..n] of string;

function Koncnica2(s: Tabela): string;
var
  g: array [1..MaxDolz] of integer;
  i, j, b, bi, bj, c, Dolz, PrejDolz: integer;
begin
  b := 0; bi := 1; bj := 0; Dolz := Length(s[1]);
  for j := 1 to Dolz do g[j] := 1;
```

[3] Zakaj je najboljša prav ta končnica? Če se $k$ poveča za 1, se v zmnožku $(2^{m-k}-1)\cdot k$ prvi faktor zmanjša na manj kot polovico, drugi pa se največ podvoji, verjetno pa se poveča manj kot za toliko. Zato se celoten zmnožek zmanjšuje, če večamo $k$.

```
  for i := 2 to n do begin
    PrejDolz := Dolz; Dolz := Length(s[i]); j := 0;
    while (j < PrejDolz) and (j < Dolz) do
      if s[i, Dolz − j] = s[i − 1, PrejDolz − j] then begin
        j := j + 1; g[j] := g[j] + 1;
        c := j * (g[j] − 1);
        if c > b then { nova najboljša končnica }
          begin b := c; bi := i; bj := j end;
      end
      else break; { konec ujemanja }
    while j < Dolz do
      begin j := j + 1; g[j] := 1 end;
  end; {for}
  Koncnica2 := Copy(s[bi], Length(s[bi]) − bj + 1, bj);
end; {Koncnica2}
```

Lepo pri tej rešitvi je, da je količina dela, ki ga imamo, v najslabšem primeru sorazmerna s skupno dolžino vseh nizov, s katerimi moramo delati. Bolje kot to že skoraj ne bi moglo biti, saj je tudi čas, potreben za branje nizov, sorazmeren s skupno dolžino vseh nizov.

## R1997.1.4

N: 2

Problem lahko rešimo z naslednjim podprogramom, ki bi ga moral sistem poklicati vsakič, ko se nek modem na novo zasede. Podprogram gre po vseh modemih in ugotavlja, kateri je že najdlje zaseden; na koncu tistega pač sprosti.

```
const N = ...; { Število vseh modemov. }

procedure KlicanObZasedbi;
var VsiZasedeni: boolean; i, T, KateriNajdlje, KolikoNajdlje: integer;
begin
  VsiZasedeni := true; i := 1;
  while VsiZasedeni and (i <= N) do begin
    T := Zaseden(i);
    if T = 0 then
      VsiZasedeni := false
    else if (i = 1) or (T > KolikoNajdlje) then
      begin KateriNajdlje := i; KolikoNajdlje := T end;
    i := i + 1;
  end; {while}
  if VsiZasedeni then Sprosti(KateriNajdlje);
end; {KlicanObZasedbi}
```

Če pa bi nas sistem obveščal tudi o tem, kdaj se nek modem iz kakršnih koli razlogov sprosti, bi lahko vzdrževali seznam zasedenih modemov, urejenih recimo padajoče po trajanju zasedenosti. Potem bi lahko zelo hitro ugotovili,

kateri je že najdlje zaseden: ni nam treba z zanko iti po vseh modemih, ampak vemo, da je najdlje zaseden kar tisti na začetku seznama.

```
const N = ...; { Število vseh modemov. }
var
  { Spremenljivka Prvi pove, kateri modem je že najdlje zaseden,
    Zadnji pa, kateri je zaseden najmanj časa. Če ni zaseden
    noben modem, sta Prvi in Zadnji enaka 0. }
  Prvi, Zadnji, StZasedenih: integer;
  { Prej[i] je številka modema, ki je v seznamu zasedenih neposredno pred
    modemom i; podobno je Nasl[i] številka tistega neposredno za i.
    Če i ni zaseden, sta v Prej[i] in Nasl[i] pač neki nesmiselni vrednosti.
    Na koncih seznama velja: Prej[Prvi] = 0 in Nasl[Zadnji] = 0. }
  Prej, Nasl: array [1..N] of integer;

procedure Inicializacija;
begin
  { Seznam zasedenih modemov je na začetku prazen. }
  Prvi := 0; Zadnji := 0; StZasedenih := 0;
end; {Inicializacija}

procedure KlicanObZasedbi(Modem: integer);
begin
  { Dodajmo novi modem na konec seznama. }
  Prej[Modem] := Zadnji; Nasl[Modem] := 0;
  if Zadnji = 0 then Prvi := Modem { seznam je bil prej prazen }
  else Nasl[Zadnji] := Modem;      { novi modem je naslednik bivšega zadnjega }
  Zadnji := Modem;                 { novi modem je zdaj po novem pač zadnji }
  { Po potrebi sprostimo najdlje zasedeni modem. }
  StZasedenih := StZasedenih + 1;
  if StZasedenih = N then Sprosti(Prvi);
end; {KlicanObZasedbi}

procedure KlicanObSprostitvi(Modem: integer);
begin
  StZasedenih := StZasedenih - 1;
  { Zbrišimo ta modem iz seznama. Predhodnik in naslednik, ki sta prej kazala
    na ta modem, morata po novem kazati drug na drugega. Posebej obravnavamo
    primere, ko je zbrisani modem prvi in/ali zadnji v seznamu. }
  if Prej[Modem] = 0 then Prvi := Nasl[Modem]
                     else Nasl[Prej[Modem]] := Nasl[Modem];
  if Nasl[Modem] = 0 then Zadnji := Prej[Modem]
                     else Prej[Nasl[Modem]] := Prej[Modem];
end; {KlicanObSprostitvi}
```

Slabost predlaganega pristopa k reševanju prezasedenosti modemov se pokaže v primerih, ko uporabniki na veliko oblegajo modeme. Ko pokličemo, sicer

takoj dobimo zvezo, vendar vsak naslednji klicatelj prekine zvezo enemu izmed tistih uporabnikov, ki so povezani že dlje kot mi, tako da po $N$ klicih pridemo na vrsto mi in nam sistem prekine zvezo. Tako je vsak uporabnik mogoče povezan le nekaj sekund in v tem času najbrž še ne more narediti ničesar uporabnega. Če se uporabniki v takem primeru poskušajo povezati znova in znova, se stanje le še poslabša.

Boljša rešitev bi verjetno bila, če ne bi vztrajali na tem, da mora biti en modem vedno prost. Lahko bi se na primer odločili, da uporabniku ne bomo prekinili zveze, če ni povezan že vsaj nekaj časa (recimo $x$ minut). Tako se bo sicer lahko zgodilo, da kak nov klicatelj preprosto ne bo mogel takoj dobiti zveze, vendar po drugi strani lahko vsakdo, ko zvezo enkrat dobi, v miru dela vsaj $x$ minut. Pri izboru števila $x$ bi morali upoštevati gostoto klicev (koliko klicev na uro), številom modemskih linij, povprečni čas, ki ga uporabniki prebijejo na zvezi, pa tudi želeni delež uspešnih klicev (v vsaj koliko odstotkih primerov naj uporabnik, ki pokliče naše modeme, tudi res dobi zvezo z enim od njih).

Pravzaprav s stališča ponudnika dostopa do Interneta metanje uporabnikov z zveze niti ni tako zelo koristno, saj jih lahko s tem prekine sredi kakšnega pomembnega dela (in si s tem nakoplje njihovo nezadovoljstvo), od tega, da se bo hip zatem povezal s tako sproščenim modemom nek drug uporabnik (namesto da bi tisti prejšnji nadaljeval z delom), pa nima ponudnik nobene posebne koristi. To, da uporabniki preprosto ne morejo do Interneta, če so vsi modemi zasedeni, je za ponudnika škodljivo šele, če postanejo zaradi tega tako nezadovoljni, da začnejo uporabljati storitve kakšnega drugega ponudnika. Če uporabnikov nočemo izgubljati, moramo torej sproti dokupovati modeme in skrbeti, da razmerje med številom uporabnikov in številom modemov ne postane previsoko.

## REŠITVE NALOG ZA DRUGO SKUPINO

### R1997.2.1

N: 3

(a) Program deli prvo število z drugim in izpiše celi del ter ostanek. V glavni zanki odšteva `n2` od `n1` in pri tem počasi povečuje celi cel `c`.[4] Poseben primer je, če je `n1` manjši od `n2` — v tem primeru odšteje enkrat, čeprav ne bi smel nobenkrat, kar rešimo s tem, da `d` postavimo na 1 (sicer pa na 0). Glavna zanka dela pravzaprav z absolutnima

[4]Namen prireditve `n1 := n1 + not n2 + 1` je zmanjšati `n1` za `n2` (enako kot pri prireditvi `n1 := n1 − n2`). Pri tem se program zanaša na to, da so cela števila v našem računalniku predstavljena z dvojiškim komplementom (kar v praksi ponavadi tudi drži); če so spremenljivke tipa `integer` dolge recimo $k$ bitov, to pomeni, da bo negativno število $-a$ (za nek $a > 0$) predstavljeno z zaporedjem $k$ bitov, ki ima dejansko vrednost $2^k - a$. Če pa v pozitivnem številu $a$ negiramo vseh $k$ bitov, dobimo zaporedje bitov z dejansko vrednostjo $(2^k - 1) - a$; če temu še prištejemo 1, dobimo torej ravno zaporedje bitov z vrednostjo $2^k - a$, ki (če delamo s predznačenimi števili) predstavlja negativno število $-a$. Tako torej s prištevanjem vrednosti `not n2 + 1` pravzaprav odštevamo `n2`.

vrednostma obeh števil, pred tem pa postavimo `e` na $-1$, če sta bili različno predznačeni, sicer pa na 1, `f` pa dobi predznak deljenca `n1`. S tem zagotovimo, da za prvotni vrednosti `n1` in `n2` ter za števili, ki ju program na koncu izpiše (recimo jima `k` in `o`) vedno velja zveza `n1 = k * n2 + o`, pa tudi $|o| < |n2|$. Količnik se po deljenju zaokroži proti 0, torej navzdol, če je bil pozitiven, in navzgor, če je bil negativen.

(b) Ukinemo lahko spremenljivke `c`, `d`, `e` in `f`, pobrišemo vse pripadajoče stavke, ukinemo zanko in spremenimo izpis v:

```
        printf("%d %d", n1 / n2, n1 % n2);
oz.     WriteLn(n1 div n2, ' ', n1 mod n2);
```

Treba pa je priznati, da v primeru, ko sta `n1` in/ali `n2` negativna, ni tako zelo zanesljivo, če bo tako poenostavljen program res dajal enake rezultate kot prvotni, kajti pri negativnih operandih definirajo različni procesorji, jeziki in prevajalniki celi del količnika in ostanek na različne načine. Razlikujejo se predvsem po tem, ali natančni rezultat deljenja zaokrožijo vedno proti 0 ali pa vedno navzdol (ta razlika se seveda pozna le, če sta operanda različno predznačena in je rezultat deljenja zato negativen); ostanek je potem običajno določen tako, da velja `(n1 / n2) * n2 + (n1 % n2) = n1`. Pri zaokrožanju proti 0 je ostanek enako predznačen kot deljenec, pri zaokrožanju navzdol pa je ostanek predznačen enako kot delitelj. Še tretja možnost je, da bi zahtevali, naj bo ostanek vedno nenegativen. V vsakem primeru pa je ostanek po absolutni vrednosti manjši od delitelja.

| | | Zaokrožanje količnika proti 0 | | Zaokrožanje količnika navzdol | | Ostanek nenegativen | |
|---|---|---|---|---|---|---|---|
| $n_1$ | $n_2$ | $n_1/n_2$ | $n_1 \% n_2$ | $n_1/n_2$ | $n_1 \% n_2$ | $n_1/n_2$ | $n_1 \% n_2$ |
| 11 | 5 | 2 | 1 | 2 | 1 | 2 | 1 |
| 11 | $-5$ | $-2$ | 1 | $-3$ | $-4$ | $-2$ | 1 |
| $-11$ | 5 | $-2$ | $-1$ | $-3$ | 4 | $-3$ | 4 |
| $-11$ | $-5$ | 2 | $-1$ | 2 | $-1$ | 3 | 4 |

Pri jezikih C in C++ je bila na primer odločitev za vrsto zaokrožanja pri deljenju dolgo časa prepuščena piscem prevajalnikov (verjetno z namenom, da bi lahko vrnili kar rezultat, ki ga izračuna procesorjev ukaz za deljenje, pa kakršnokoli obliko zaokrožanja ta že pač uporablja), v standardu C99 pa so uvedli zaokrožanje proti 0. Intelovi procesorji zaokrožajo proti 0, zato je ta oblika zaokrožanja vsaj na njih prevladovala že prej. Java in C# zaokrožata proti 0, Python, Oberon in Haskell pa vedno zaokrožijo navzdol. Pri standardnem pascalu zaokroža **div** proti 0, **mod** pa je definiran le pri pozitivnem delitelju (pri negativnem velja računanje **mod** za napako) in to tako, da je vedno nenegativen; to žal pomeni, da velja `n2 * (n1 div n2) + (n1 mod n2) = n1` samo za nenegativne deljence `n1` (ali pa če se deljenje izide), sicer pa ne.

Vendar pa se mnogi pascalski prevajalniki te definicije ne držijo in računajo `n1 mod n2` kot `n1 - (n1 div n2) * n2`. Slabost definicije, ki uporablja zaokrožanje proti 0, je na primer naslednja: če povečamo `n1` za `n2`, bi nemara pričakovali, da se bo količnik povečal za 1, ostanek pa se ne bo spremenil; vendar pri tej definiciji to ne drži vedno (težava nastopi, če se spremeni predznak deljenca `n1`).[5]

(c) Preveriti moramo, če je drugo število enako 0, saj ne smemo deliti z 0 (obstoječi program bi se v tem primeru zaciklal, saj vrednosti `n1` v glavni zanki sploh ne bi spreminjal). Še ena slabost obstoječega programa je, da uporablja `n1 + not n2 + 1`, da bi izračunal `n1 - n2`. Vrednost `not n2 + 1` je enaka $-$`n2` v primeru, če so števila v računalniku predstavljena z dvojiškim komplementom, drugače pa to ni nujno res. Na kakšnih starih in/ali eksotičnih računalnikih bi utegnili naleteti tudi na drugačne predstavitve celih števil (na primer eniški komplement, kjer bi že `not n2` sam po sebi imel vrednost $-$`n2`).

(d) Program dvakrat izpiše 42 pri poljubnem paru celih števil $(a, b)$, za kateri velja, da je $a = 42(b + 1)$ in hkrati $b > 42$. Primer: 1848 in 43.

## R1997.2.2

N: 4

Za skrivanje sporočila ni treba drugega, kot da beremo znake enega za drugim, nato pa pri vsakem znaku jemljemo iz njega posamezne bite in vsakega vpišemo v najnižji bit enega piksla slike. Pri razkrivanju sporočila ravnamo ravno obratno; iz zaporednih pikslov jemljemo spodnji bit in te bite združujemo po osem skupaj v znake, ki jih potem izpisujemo. V obeh primerih si pomagamo s preprostimi operacijami nad biti: `n mod 2` vrne najnižji bit števila `n`, `n div 2` zamakne `n` za eno mesto v desno (najnižji bit se pri tem izgubi), `n * 2` pa zamakne `n` za eno mesto v levo (v najnižji bit pride vrednost 0).

```
const w = 256; h = 256;
var Slika: array [1..h, 1..w] of 0..255;

procedure SkrijBesedilo;
var
  iw, ih, j, ic, bit: integer;
  c: char;
begin
  iw := 1; ih := 1;
  while ih <= h do begin
    Read(c); ic := Ord(c);
    { Shrani 8 bitov znaka v zaporedne piksle, začni z najnižjim bitom (lsb). }
    { Predpostavimo, da je širina slike mnogokratnik 8. }
```

[5] Nekaj literature: obširna razprava o zaokrožanju pri deljenju je bila v skupini *comp.lang.c++.moderated* v začetku februarja 2000; Daan Leijen: *Division and modulus for computer scientists*, `http://www.cs.uu.nl/~daan/`; Raymond T. Boute: *The Euclidean definitions of the functions div and mod*, ACM Transactions on Programming Languages and Systems, 14(2):127–144, April 1992.

```
      for j := 1 to 8 do begin
        bit := ic mod 2; ic := ic div 2; { Izlušči naslednji bit znaka. }
        Slika[ih, iw] := (Slika[ih, iw] div 2) * 2 + bit;
        iw := iw + 1;
      end; {for}
      if iw > w then begin iw := 1; ih := ih + 1 end;
    end; {while}
end; {SkrijBesedilo}

procedure RazkrijBesedilo;
var
  ic, i, j: integer;
  k, kp: integer; { k šteje bite; kp = 2^k }
begin
  k := 0; kp := 1; ic := 0;
  for i := 1 to h do
    for j := 1 to w do begin
      { Izlušči zaporedne bite iz pikslov in jih združi po 8 v en znak. }
      { Pazimo na obratni vrstni red bitov: najnižji bit je v prvem pikslu. }
      ic := ic + kp * (Slika[i, j] mod 2); k := k + 1; kp := 2 * kp;
      if k >= 8 then begin { Znak je kompleten. }
        Write(Chr(ic));
        k := 0; kp := 1; ic := 0; { Pripravi se na nov znak. }
      end; {if}
    end; {for}
  WriteLn;
end; {RazkrijBesedilo}
```

N: 4

## R1997.2.3

Podnize lahko štejemo rekurzivno. Recimo, da bi radi prešteli pojavitve podniza $p[1..m]$ v nizu $s[1..n]$. Vsaka pojavitev podniza $p$ se mora končati pri eni od pojavitev njegovega zadnjega znaka, $p[m]$, v nizu $s$; če je to na primer $i$-ti znak $s$-ja, je preostanek neka pojavitev niza $p[1..m-1]$ v nizu $s[1..n-1]$. Pojavitve podniza $p$ v $s$, ki se končajo pri $s[i]$, lahko torej preštejemo z rekurzivnim klicem, ki bo obdeloval za en znak krajši podniz. Na koncu moramo sešteti vse tako najdene pojavitve, torej po vseh primernih $i$ (takih, za katere je $s[i] = p[m]$). Rekurzija se konča, ko je podniz dolg le še en znak — takrat moramo le prešteti vse pojavitve tega znaka v nizu $s$.

Če označimo število pojavitev podniza $p[1..i]$ v nizu $s[1..j]$ z $f(i,j)$, smo tako dobili rekurzivno zvezo:

$$f(i,j) = \sum_{1 \le k \le j,\ s[k]=p[i]} f(i-1,k-1) \qquad (\star)$$

s posebnimi primeri

$$\begin{array}{rcl} f(1,j) & = & \text{število pojavitev znaka } p[1] \text{ v nizu } s[1..j], \\ f(i,j) & = & 0, \text{če } i > j \text{ (niz prekratek za podniz).} \end{array}$$

Primera za $f(1,j)$ pravzaprav ni treba obravnavati posebej, saj lahko definiramo $f(0,j) = 1$ in potem $f(1,j)$ računamo po splošni formuli ($\star$).

Toda po formuli ($\star$) bi morali $f(i,j)$ računati z zanko po $k$. Temu se lahko izognemo in prihranimo nekaj časa, če rekurzivni razmislek zastavimo malo drugače. Vsaka pojavitev niza $p[1..i]$ v nizu $s[1..j]$ se ali konča pri znaku $s[j]$ (kar je seveda mogoče le, če je $p[i] = s[j]$; v tem primeru bo preostanek te pojavitve kar neka pojavitev niza $p[1..i-1]$ v nizu $s[1..j-1]$) ali pa ne (in v tem primeru je taka pojavitev hkrati tudi pojavitev celega niza $p[1..i]$ v nizu $s[1..j-1]$. Tako dobimo namesto ($\star$) preprostejšo formulo

$$f(i,j) = \begin{cases} f(i,j-1) + f(i-1,j-1) & : \text{ če } p[i] = s[j] \\ f(i,j-1) & : \text{ sicer.} \end{cases}$$

Primer programa, ki računa to funkcijo:

```
function PrestejPodnize(p, s: string): longint;

  { Izračuna, kolikokrat se pojavi p[1..i] v nizu s[1..j]. }
  function f(i, j: integer): integer;
  begin
    if i > j then f := 0
    else if i = 0 then f := 1
    else if s[j] = p[i] then f := f(i, j - 1) + f(i - 1, j - 1)
    else f := f(i, j - 1);
  end; {f}

var m, n: integer;
begin
  m := Length(p); n := Length(s);
  PrestejPodnize := f(m, n);
end; {PrestejPodnize}
```

Slabost tega postopka je, da je lahko zelo neučinkovit. Skupno število pojavitev podniza $p$ v nizu $s$, ki ga na koncu vrne podprogram PrestejPodnize, je nastalo v okviru rekurzivnih klicev podprograma f s seštevanjem enic in ničel, ki jih prispevata stavka f := 1 in f := 0. Torej, če ima $p$ v $s$-ju $r$ pojavitev, se mora stavek f := 1 izvesti $r$-krat, kar je neprijetno, če je $r$ velik. In že pri razmeroma kratkih nizih je število pojavitev podniza lahko veliko: niz $a_1a_2\cdots a_m$ (če so $a_1, \ldots, a_m$ same različne črke) ima v nizu $a_1a_1a_2a_2\cdots a_ma_m$ kar $2^m$ pojavitev.

Postopek lahko izboljšamo, če opazimo, da je vrednost, ki jo vrne funkcija f, odvisna le od vrednosti parametrov i in j — edine preostale zunanje vrednosti,

ki jih f uporablja, so oba niza in njuni dolžini, te pa se nikoli ne spreminjajo. Torej, če f dvakrat pokličemo z istima vrednostma parametrov, mora obakrat vrniti isti rezultat. Zato si je pametno ta rezultat ob prvem klicu zapomniti v kakšni tabeli in ob kasnejših klicih takoj vrniti to vrednost, namesto ta jo gremo računat ponovno od začetka (temu včasih pravijo *memoizacija*; lepše bi temu verjetno lahko rekli „pomnjenje“). Spodnji program si rezultat klica f(i, j) zapomni v tabeli t[i, j] (na začetku postavimo vse t[i, j] na $-1$, da bomo vedeli, da teh vrednosti še nismo izračunali).

```
function PrestejPodnize(p, s: string): longint;
var m, n: integer; t: array [1..MaxDolzPodniza, 1..MaxDolzNiza] of integer;

  function f(i, j: integer): integer;
  begin
    if i > j then f := 0
    else if i = 0 then f := 1
    else begin
      if t[i, j] < 0 then
        if s[j] = p[i] then t[i, j] := f(i, j - 1) + f(i - 1, j - 1)
        else t[i, j] := f(i, j - 1);
      f := t[i, j];
    end; {if}
  end; {f}

var i, j: integer;
begin
  m := Length(p); n := Length(s);
  for i := 1 to m do for j := 1 to n do t[i, j] := -1;
  PrestejPodnize := f(m, n);
end; {PrestejPodnize}
```

(Tabelo t bi lahko alocirali tudi dinamično, da bi imela natančno $m \times n$ celic.)

Zdaj se f(i, j) kliče največ dvakrat za vsak par vrednosti i in j — namreč ob prvem klicu f(i + 1, j + 1) in ob prvem klicu f(i, j + 1), ob kasnejših klicih teh dveh funkcij pa bosta onidve vrnili svoj rezultat iz tabele t in ne bosta rekurzivno klicali f(i, j). Količina časa, ki ga naš program tako porabi, je torej $O(m \cdot n)$, saj imamo z vsako celico tabele t konstantno veliko dela.

Program pa lahko naredimo še malce bolj elegantnega, če upoštevamo, katere rezultate rekurzivnih klicev utegnemo potrebovati, ko bomo računali $f(i, j)$: to sta le $f(i, j-1)$ in $f(i-1, j-1)$. Torej lahko takrat, ko računamo $f(i, j)$, že pozabimo vrednosti $f(i', j')$ za $j' < j - 1$. Če torej računamo vrednosti $f$ sistematično po naraščajočih $j$ in pri vsakem $j$-ju po vseh $i$, je dovolj, če v tabeli t hranimo vrednosti $f(i, j-1)$. Vrednosti $f(i, j)$ za trenutni $j$ pa je koristno računati po padajočih $i$, ker nam to zagotavlja, da po tistem, ko izračunamo $f(i, j)$, vrednosti $f(i, j-1)$ ne bomo več potrebovali, zato jo lahko

v tabeli takoj povozimo z vrednostjo $f(i, j)$. Rekurzivnim klicem se lahko tako povsem odpovemo in vse naredimo z dvema gnezdenima zankama:

```
function PrestejPodnize(p, s: string): longint;
var m, n, i, j, iMax: integer;
    t: array [0..MaxDolzPodniza] of integer;
begin
  m := Length(p); n := Length(s);
  { Izračunajmo najprej f(i, 1) za vse i. Tu gledamo le
    prvi znak niza s in f(i, 1) je 0 pri za vse i > 1. }
  t[0] := 1; { Ker smo videli, da je koristno vzeti f(0, j) = 1. }
  for i := 1 to m do t[i] := 0;
  if p[1] = s[1] then t[1] := 1;
  { Izračunajmo zdaj f(i, j) za vse večje j. }
  for j := 2 to n do begin
    if j > m then iMax := m else iMax := j;
    for i := iMax downto 1 do
      { V celicah t[i + 1..m] so vrednosti f(i + 1, j), ..., f(m, j),
        v celicah t[1..i - 1] pa vrednosti f(1, j - 1), ..., f(i, j - 1).
        Izračunajmo f(i, j) in jo vpišimo v t[i]. }
      if p[i] = s[j]
      then t[i] := t[i] + t[i - 1]    { f(i, j) = f(i, j - 1) + f(i - 1, j - 1) }
      else  t[i] := t[i];             { f(i, j) = f(i, j - 1) }
  end; {for j}
  PrestejPodnize := t[m];
end; {PrestejPodnize}
```

Tehniki, ki smo jo uporabili pri tej (in pravzaprav tudi pri prejšnji) različici, pravimo *dinamično programiranje*. Količina porabljenega časa je tudi tokrat $O(m \cdot n)$, čeprav je v praksi ta različica mogoče malo hitrejša od prejšnje, ker tista porabi precej časa za knjigovodske opravke pri rekurzivnih klicih. Prihranili smo tudi precej pomnilnika, saj za tabelo `t` potrebujemo le še $O(m)$ namesto $O(mn)$ pomnilnika. Je pa zato pri tej zadnji različici precej več možnosti, da se zmotimo pri kakšnih indeksih.

Prireditev `t[i] := t[i]` je v zgornjem podprogramu seveda povsem odveč in bi jo bilo v praksi najpametneje izpustiti. Notranja zanka (po $i$) torej spreminja tabelo `t` le pri tistih $i$, za katere je $p[i] = s[j]$, zato je dovolj, če obiščemo le te $i$, ostale pa preskočimo. Lahko bi si torej za vsak $j$ pripravili seznam vseh primernih $i$ in se potem le sprehodili po tem seznamu; časovna zahtevnost bi bila zdaj sorazmerna s skupno dolžino teh seznamov, torej s številom parov $(i, j)$, za katere je $p[i] = s[j]$. V najslabšem primeru jih je sicer $m \cdot n$ in takrat ničesar ne pridobimo, pri bolj realističnih nizih (ki nimajo samih enakih črk) pa jih utegne biti precej manj. Do takšnih seznamov lahko pridemo tako, da

črke $p$-ja in $s$-ja uredimo in „zlijemo“.[6]

```
function PrestejPodnize(p, s: string): longint;
var m, n, i, j, iu, ju: integer;
    t: array [0..MaxDolzPodniza] of integer;
    su, iuZadnji: array [1..MaxDolzNiza] of integer;
    pu: array [1..MaxDolzPodniza] of integer;
begin
  m := Length(p); n := Length(s);
  for ju := 1 to n do su[ju] := ju;
  Preuredi elemente tabele su tako, da bo za vse ju veljalo:
  (s[su[ju]] < s[su[ju + 1]]) or ((s[su[ju]] = s[su[ju + 1]]) and (su[ju] < su[ju + 1]));
  for iu := 1 to m do pu[iu] := iu;
  Preuredi elemente tabele pu tako, da bo za vse iu veljalo:
  (p[pu[iu]] < p[pu[iu + 1]]) or ((p[pu[iu]] = p[pu[iu + 1]]) and (pu[iu] < pu[iu + 1]));
  iu := 0;
  for ju := 1 to n do begin
    j := su[ju];
    while iu < m do begin
      i := pu[iu + 1];
      if (p[i] > s[j]) or ((p[i] = s[j]) and (i > j)) then break;
      iu := iu + 1;
    end; {while};
    iuZadnji[j] := iu;
  end; {for ju}
```

Za urejanje bi načeloma lahko uporabili katerega koli od mnogih znanih algoritmov za urejanje. Po tej inicializaciji imamo v tabeli pu števila $i$ od 1 do $m$, urejena po naraščajoči vrednosti $p[i]$ in pri enakih $p[i]$ še po naraščajočih $i$. Vrednost iuZadnji[j] = iu pa nam pove, da je pu[iu] zadnji element tabele pu, ki se še nanaša na črko s[j] (torej da je p[pu[iu]] = s[j]) in je istočasno tudi manjši ali enak $j$ (torej da je pu[iu] ≤ j). Če takega elementa sploh ni, je iuZadnji[j] enak 0 ali pa kaže na nek iu, pri katerem je p[pu[iu]] < s[j]. Nadaljujemo lahko tako kot pri prejšnji rešitvi, le glavno zanko bomo morali malo predelati:

```
  t[0] := 1; { Tu se ni nič spremenilo. }
  for i := 1 to m do t[i] := 0;
  if p[1] = s[1] then t[1] := 1;
  for j := 2 to n do begin { Tale glavna zanka se malo spremeni. }
    iu := iuZadnji[j];
    while iu > 0 do begin
      i := pu[iu]; if p[i] <> s[j] then break;
```

[6] J. W. Hunt, T. G. Szymanski: *A fast algorithm for computing longest common subsequences.* CACM, 20(5):350–353, May 1977.

```
        t[i] := t[i] + t[i − 1];
        iu := iu − 1;
      end; {while}
  end; {for j}
  PrestejPodnize := t[m];
end; {PrestejPodnize}
```

Če sta niza $p$ in $s$ dovolj dolga in pride do ujemanja $p[i] = s[j]$ pri dovolj malo parih $(i, j)$, bo prihranek zaradi manjšega števila izvajanj notranje zanke (**while** iu > 0 v zadnji različici rešitve) večji od časa, ki smo ga porabili na začetku za pripravo tabele iuZadnji.

## R1997.2.4

N: 5

Če bi na začetku vsak delež $d_i$ zaokrožili navzdol, torej na $\lfloor d_i \rfloor$, bi gotovo dobili vsoto, manjšo ali enako 100. Recimo, da je vsota $100 - k$; potem moramo popraviti $k$ deležev — za zdaj so vsi zaokroženi navzdol, mi pa jih moramo zaokrožiti navzgor. Pametno se je najprej lotiti tistih, pri katerih je zaokroževanje navzdol naredilo največjo napako, torej tisth z največjo $d_i - \lfloor d_i \rfloor$ (kajti pri teh bo napaka $\lceil d_i \rceil - d_i$ po zaokrožanju navzgor najmanjša). Če bi bilo deležev veliko, bi jih bilo koristno urediti po padajoči napaki, spodnji program pa vsakič poišče naslednjega kar s pregledom celotne tabele vseh deležev.

```
const n = 20;
var
  Delez: array [1..n] of real;
  Odst: array [1..n] of integer;
  Min: real;
  j, IndMin, Vsota: integer;
begin
  PreberiAliIzracunajDeleze;    { S tem se ne bomo ukvarjali, saj naloga tako ali
                                  tako pravi, da so deleži že podani v tabeli. }
  Vsota := 0;
  for j := 1 to n do begin
    Delez[j] := Delez[j] * 100;
    Odst[j] := Trunc(Delez[j]); Vsota := Vsota + Odst[j];
  end; {for}
  while Vsota < 100 do begin
    IndMin := 1; Min := Odst[IndMin] − Delez[IndMin];
    for j := 2 to n do
      if Odst[j] − Delez[j] < Min then
        begin IndMin := j; Min := Odst[j] − Delez[j] end;
    Odst[IndMin] := Odst[IndMin] + 1; Vsota := Vsota + 1;
  end; {while}
  for j := 1 to n do WriteLn(Odst[j]);
end.
```

Zanki **while** bi lahko prihranili nekaj ponovitev, če bi na začetku zaokrožili vsak delež k najbližjemu celemu številu, ne pa vseh navzdol. Potem bi morali ločiti dve možnosti: lahko je vsota zaokroženih deležev premajhna in moramo kot doslej povečati nekaj deležev, ki so bili zaokroženi navzdol (in sicer tiste z največjo $d_i - \lfloor d_i \rfloor$; lahko pa je vsota prevelika in moramo zmanjšati nekaj deležev, ki so bili zaokroženi navzgor (tiste z največjo $\lceil d_i \rceil - d_i$). Seveda bi lahko uporabili tudi kak algoritem za urejanje in števila eksplicitno uredili po tem, za koliko so se ob zaokrožanju spremenila.

## REŠITVE NALOG ZA TRETJO SKUPINO

N: 6

### R1997.3.1

Kazalca p1 in p2 potujeta vzdolž seznama s. Kazalec p1 se pri vsaki izvedbi telesa zanke **repeat** premakne naprej po seznamu s za en element, kazalec p2 pa za dva elementa. Kazalec p2 potuje torej dvakrat hitreje vzdolž seznama s kot kazalec p1.

V mejnem primeru, ko velja s = **nil** in $n = 0$, se telo zanke **repeat** izvede enkrat, funkcija DvojnaHitrost pa vrne false. Predpostavimo sedaj, da seznam ni prazen, torej $n > 0$. Če kazalec p2 med potovanjem vzdolž seznama dobi vrednost **nil**, je dosegel zadnji, torej $n$-ti element seznama in zanka **repeat** se izteče, funkcija pa vrne vrednost false. Pri $n$ elementih v seznamu kazalec p2 potrebuje $n$ premikov, da dobi vrednost **nil**, za to pa potrebuje $\lceil n/2 \rceil$ iteracij **repeat**. Obstaja pa še možnost, da kazalec p2 nikoli ne dobi vrednosti **nil**. To se zgodi v primeru, ko kazalec Naslednji $n$-tega elementa kaže na $k$-ti element seznama, za nek $k \leq n$. Seznam torej lahko vsebuje zanko, v kateri so elementi z indeksi $k, k+1, \ldots, n$. V $k$-tem koraku kazalec p1 doseže zanko elementov v seznamu, kazalec p2 pa je tudi že v zanki. A ker kazalec p1 potuje s korakom 1, kazalec p2 pa s korakom 2, se z vsako ponovitvijo zanke **repeat** razdalja med kazalcema v zanki zmanjša za 1. To pomeni, da bo kazalec p2 prej ali slej ujel kazalec p1, vrednosti kazalcev bosta enaki in zanka **repeat** se bo iztekla, funkcija DvojnaHitrost pa bo vrnila vrednost true.

Oštevilčimo v mislih elemente zanke s številkami od 0 ($k$-ti element seznama) do $n - k$ ($n$-ti element). Kazalec p1 je najprej naredil $k - 1$ korakov, da je dosegel zanko; kazalec p2 je v tem času naredil dvakrat toliko korakov, torej je po tistem, ko je prišel do zanke (do $k$-tega elementa seznama), naredil še $k - 1$ korakov. S tem je prišel na element s številko $(k - 1) \bmod (n - k + 1)$. Ker je p1 trenutno na elementu 0 in se mu p2 v vsaki iteraciji zanke približa za en element, bo porabil p2 še $(0 - (k - 1)) \bmod (n - k + 1)$ iteracij, da ga bo ujel. Tu je mišljeno, naj mod vedno vrača vrača vrednosti od 0 do $n - k$ (torej $a \bmod m := a - m\lfloor a/m \rfloor$; na primer: $(-22) \bmod 6 = 2$). Negativnega operanda $-(k - 1)$ se lahko znebimo tudi tako, da mu prištejemo nek večkratnik delitelja (saj na mod to ne bo vplivalo); na primer, ker je $k \leq n$,

je $n-k+1 \geq 1$, zato je $(k-1)(n-k+1) \geq (k-1)$; zato je

$$\begin{aligned}[-(k-1)] \bmod (n-k+1) &= [(k-1)(n-k+1)-(k-1)] \bmod (n-k+1)\\ &= (k-1)(n-k) \bmod (n-k+1),\end{aligned}$$

pri čemer je zdaj deljenec, $(k-1)(n-k)$, gotovo nenegativen. Dobljena formula odpove le v primerih, ko je $k=1$, saj sta tam `p1` in `p2` že na začetku skupaj in formula napove 0 iteracij; v resnici pa se takrat izvede $n$ iteracij. Tako smo dobili:

$$\textit{št. iteracij} = \begin{cases} 1, & \text{če } n=0\\ \lceil n/2 \rceil, & \text{če } n>0 \text{ in ni zanke}\\ n, & \text{če } n>0 \text{ in } k=1\\ k-1+[(k-1)(n-k) \bmod (n-k+1)] & \text{sicer.}\end{cases}$$

Skratka, funkcija `DvojnaHitrost` vrne vrednost `true`, če v seznamu `s` obstaja zanka, sicer pa vrednost `false`. V najslabšem primeru (pri praznem seznamu) se telo zanke **`repeat`** izvede največ $(n+1)$-krat (če je $n$ število elementov v seznamu).

## R1997.3.2

N: 6

Na to, kako se prenaša pomembnost med meščani, vpliva le njihova moč, ta pa se ob prenašanju pomembnosti nič ne spreminja. Torej je vseeno, v kakšnem vrstnem redu meščani prenašajo svojo pomembnost na svoje močnejše prijatelje; v vsakem primeru bomo na koncu dobili enak razpored pomembnosti. Poleg tega se pomembnost prenaša na vedno močnejše ljudi, ker pa je moč navzgor omejena (če imamo $n$ meščanov, ima lahko posameznik moč največ $n-1$, saj več kot toliko prijateljev pač ne more imeti), se mora tak postopek prenašanja pomembnosti zagotovo prej ali slej končati. Torej nam ni treba skrbeti, da bi se nam kak postopek zaciklal ali pa da rešitev ne bi bila enolična.

Meščani z močjo 0 so po definiciji brez prijateljev in se torej njihova pomembnost ne bo nikoli spreminjala; ničesar ne bodo razdajali in ničesar dobivali. Z njimi se nam torej sploh ni treba ukvarjati. — Meščani z močjo 1 ne bodo nikoli dobili nič pomembnosti od drugih: dobili bi jo lahko le od takih z močjo 0, toda če ima nekdo moč 0, pomeni, da nima prijateljev in torej svoje pomembnosti ne bo delil. Torej, ko meščani z močjo 1 razdelijo svojo začetno pomembnost (če jo imajo komu dati), se njihova pomembnost v bodoče prav gotovo ne bo več spreminjala. Zato je pametno najprej opraviti z njimi, potem pa vemo, da lahko odslej pozabimo nanje. — Ko smo opravili to, vidimo, da meščani z močjo 2 tudi ne bodo dobili nič več pomembnosti: lahko bi jo dobili le od takih z močjo 1, toda slednji so doslej že razdelili vse, kar so imeli, in tudi v bodoče ne bodo delili ničesar več. Če torej zdaj meščani z močjo 2 razdelijo svojo pomembnost med svoje močnejše prijatelje, bomo tudi pri njih že dobili končno stanje pomembnosti.

Tako lahko nadaljujemo proti vse močnejšim meščanom. Preden se začnemo ukvarjati s tistimi z močjo $k$, smo za vse šibkejše meščane že izračunali njihovo končno pomembnost, tako da meščani z močjo $k$ v prihodnje gotovo ne bodo prejeli nič dodatne pomembnosti več; meščani z močjo $k$ lahko potem razdelijo svojo pomembnost med svoje močnejše prijatelje in s tem dobimo njihove dokončne pomembnosti, ki se v bodoče ne bodo več spreminjale.

```
program Ravbarji;
const MaxN = ...;
var n, i, u, v, StMocnejsih: integer;
    StZMocjo, NaslZMocjo: array [0..MaxN − 1] of integer;
    Moc, PoMoci, Mocnejsi: array [1..MaxN] of integer;
    Pomembnost: array [1..MaxN] of real; Delez: real;
begin
  n := PreberiMescane;
  { Določimo moč vsakega meščana. }
  for u := 1 to n do begin
    Moc[u] := 0; Tekoci(u);
    while Naslednji <> 0 do Moc[u] := Moc[u] + 1;
  end; {for}
  { Koliko jih ima določeno moč? }
  for i := 0 to n − 1 do StZMocjo[i] := 0;
  for u := 1 to n do StZMocjo[Moc[u]] := StZMocjo[Moc[u]] + 1;
  { V tabeli PoMoci bomo pripravili številke meščanov po naraščajoči moči.
    NaslZMocjo[i] pove, kam je treba vpisati naslednjega meščana z močjo i. }
  NaslZMocjo[0] := 1;
  for i := 1 to n − 1 do NaslZMocjo[i] := NaslZMocjo[i − 1] + StZMocjo[i − 1];
  for u := 1 to n do begin
    PoMoci[NaslZMocjo[Moc[u]]] := u;
    NaslZMocjo[Moc[u]] := NaslZMocjo[Moc[u]] + 1;
  end; {for}
  { Prenašajmo pomembnosti od šibkejših k močnejšim. }
  for u := 1 to n do Pomembnost[u] := 1.0;
  for i := 1 to n do begin
    u := PoMoci[i];
    { Poglejmo, kateri u-jevi prijatelji so močnejši od njega. }
    Tekoci(u); StMocnejsih := 0; v := Naslednji;
    while v <> 0 do begin
      if Moc[v] > Moc[u] then
        begin StMocnejsih := StMocnejsih + 1; Mocnejsi[StMocnejsih] := v end;
      v := Naslednji;
    end; {while}
    if StMocnejsih > 0 then begin
      { Razdelimo u-jevo pomembnost med močnejše prijatelje. }
      Delez := Pomembnost[u] / StMocnejsih;
      Pomembnost[u] := 0;
```

```
        while StMocnejsih > 0 do begin
          v := Mocnejsi[StMocnejsih]; StMocnejsih := StMocnejsih − 1;
          Pomembnost[v] := Pomembnost[v] + Delez;
        end; {while}
      end; {if}
    end; {for}
    { Izpišimo meščane z neničelno pomembnostjo. }
    for u := 1 to n do if Pomembnost[u] > 0 then
      WriteLn(u, ' ', Pomembnost[u]:0:5);
end. {Ravbarji}
```

Če imamo $n$ meščanov in $m$ prijateljstev med meščani, je časovna zahtevnost tega postopka $O(n + m)$, prostorska pa $O(n)$ (če odmislimo prostor, ki ga verjetno uporabljajo PreberiMescane, Tekoci in Naslednji za hranjenje podatkov o meščanih in njihovih prijateljstvih). Če ima vsak meščan bolj malo prijateljev, je $m = O(n)$, lahko pa je vsak prijatelj skoraj vseh drugih in je tedaj $m = O(n^2)$.

Če ima vsak meščan enako število prijateljev, do prenašanja pomembnosti sploh ne pride — vsi obdržijo pomembnost 1.

## R1997.3.3

N: 7

Iz čipa beremo zapise enega za drugim. V tabelo Indeksi si zapišemo položaj začetka zapisa v čipu, v tabelo Cene pa vpišemo število skokov, ki jih moramo narediti, da pridemo od prvega zapisa do trenutnega. To je v najslabšem primeru za ena večje od števila skokov, ki nas privedejo do zapisa, ki stoji pred trenutnim. Mogoče pa gre tudi z manj skoki; zato preverimo, če lahko s spremembo velikosti kakšnega izmed prejšnjih zapisov pridemo do trenutnega in to tako, da se bo število skokov čimbolj zmanjšalo. Če smo tak zapis našli, si njegovo zaporedno številko zapišemo v tabelo OdKod. V nasprotnem primeru pa v tabelo OdKod zapišemo zaporedno številko prejšnjega zapisa.

Ko pridemo do zadnjega zapisa, se na zadnjem mestu v tabeli Cene nahaja število skokov, ki jih potrebujemo, da pridemo do zadnjega zapisa.

Na koncu moramo pot z najmanjšim številom skokov zapišati še v Flash ROM. Zdaj uporabimo tabelo Odkod, ki brana od zadaj na način Zapis := OdKod[Zapis] vsebuje vse zapise, ki jim moramo spremeniti velikost.

```
var
  Indeksi, Cene, OdKod: array [1..MaxFlashROM] of integer;
  Zadnji, Lokacija, Velikost, Zapis: integer;
begin
  Indeksi[1] := 1;
  Cene[1] := 0;
  OdKod[1] := 0;
  Lokacija := FlashROM[1] + 1;
  Zadnji := 1;
```

```
  while FlashROM[Lokacija] <> 0 do begin
    Zadnji := Zadnji + 1;
    Indeksi[Zadnji] := Lokacija;
    Cene[Zadnji] := Cene[Zadnji − 1] + 1;
    OdKod[Zadnji] := Zadnji − 1;
    for Zapis := 1 to Zadnji − 2 do begin
      if Cene[Zapis] + 1 < Cene[Zadnji] then begin
        { Splačalo bi se skočiti od zapisa Zapis do zapisa Zadnji;
          preverimo, če je tak skok sploh mogoč. }
        Velikost := (Lokacija − Indeksi[Zapis]) or FlashROM[Indeksi[Zapis]];
        if Indeksi[Zapis] + Velikost = Lokacija then begin
          Cene[Zadnji] := Cene[Zapis] + 1;
          OdKod[Zadnji] := Zapis;
        end; {if}
      end; {if}
    end; {for}
    Lokacija := Lokacija + FlashROM[Lokacija];
  end; {while}
  while Zadnji <> 1 do begin
    Zapis := OdKod[Zadnji];
    Vpisi(Indeksi[Zapis], Indeksi[Zadnji] − Indeksi[Zapis]);
    Zadnji := Zapis;
  end; {while}
end.
```

N: 8

## R1997.3.4

Ko dobimo od uporabnika nov zahtevek, moramo najprej preveriti, če so zahtevani podatki že v vmesnem pomnilniku. Če so, nam jih funkcija Poslji tudi vrne in jih moramo samo še poslati naprej uporabniku. Drugače pa nam Poslji pove številko toka, kamor naj pošiljamo te podatke (ko jih bomo dobili iz omrežja), da se bodo shranili v vmesni pomnilnik. Obenem bo pametno te podatke pošiljati tudi uporabniku, ki jih je sploh najprej zahteval. Oba tokova (uporabnikovega in tistega za vmesni pomnilnik) shranimo v eni od celic tabele Zahteve, za indeks pa uporabimo številko toka, po kateri prihajajo podatki do nas iz Interneta (ta tok nam vrne podprogram Zahtevaj).

```
const MaxZahtev = 10000; { Največ dovoljenih istočasnih zahtev. }
type DvaTokova = record DoUporabnika, VPomnilnik: integer end;
var Zahteve: array [1..MaxZahtev] of DvaTokova;

procedure Zahtevek(Naslov: string; Tok: integer);
var
  Podatki: string;
  VmTok, NovTok: integer;
begin
```

```
  VmTok := Shranjen(Naslov, Podatki);
  if VmTok = 0 then begin
    Poslji(Podatki, Tok);
    { Ker v nalogi ni opisano, kako se tokove zapira, si mislimo, da se
      tok zapre, če mu pošljemo kos podatkov ničelne dolžine. Konec koncev
      se takšno delovanje zahteva tudi od našega podprograma Prispelo,
      pa tudi on pričakuje enako od tokov, s katerimi dela. }
    Poslji('', Tok);
  else begin
    NovTok := Zahtevaj(Naslov);
    Zahteve[NovTok].DoUporabnika := Tok;
    Zahteve[NovTok].VPomniknik := VmTok;
  end; {if}
end; {Zahtevek}

procedure Prispelo(Podatki: string; Tok: integer);
begin
  Poslji(Podatki, Zahteve[Tok].DoUporabnika);
  Poslji(Podatki, Zahteve[Tok].VPomnilnik);
  if Length(Podatki) = 0 then begin
    Zahteve[Tok].DoUporabnika := 0;
    Zahteve[Tok].VPomnilnik := 0;
  end; {if}
end; {Prispelo}
```

Opisana rešitev ima lahko težave v primerih, ko prispe kmalu po prvem zahtevku za nek naslov še en zahtevek na isti naslov. Recimo, da smo nekaj podatkov s tega naslova že dobili (in shranili v vmesni pomnilnik), ne pa vseh; kaj zdaj vrne funkcija Shranjen, ko jo pokličemo ob drugem prejetem zahtevku? Če pravi, da podatki za ta naslov še niso shranjeni, bomo poslali v Internet še eno zahtevo za ta naslov in tako po nepotrebnem tratili pasovno širino; če pa reče, da so podatki za ta naslov že na voljo, nam bo vrnila seveda le tisto, kar se je dotlej za ta naslov že nakapljalo, mi pa bomo to uporabniku posredovali naprej, kot da bi bili to že vsi podatki za ta naslov. Boljša rešitev bi bila, da bi uporabniku v takem primeru takoj posredovali vse podatke, ki so s tega naslova že prišli, obenem pa bi njegov tok dodali na seznam „porabnikov“ (ali „odjemalcev“ ali „poslušalcev“); podprogram Prispelo bi posredoval na novo prispele podatke vmesnemu pomnilniku in še vsem porabnikom. Vsaka celica tabele Zahteve bi poleg toka za pisanje v vmesni pomnilnik vsebovala še kazalec na začetek seznama porabnikov. Potrebovali bi še način, kako ugotoviti, katera celica tabele Zahteve se nanaša na posamezni naslov (če sploh katera); lahko bi si pomagali z razpršeno tabelo ali pa naprtili funkciji Zahtevaj, naj to opravi namesto nas in nam vrne številko toka, s katerega že zdaj prihajajo podatki za isto zahtevo.

# REŠITVE NALOG TRETJEGA ZAKLJUČNEGA TEKMOVANJA IZ ZNANJA RAČUNALNIŠTVA

N: 9

## R1997.Z.1

Definirajmo sosede nekega mesta kot vsa tista mesta, ki so z njim povezana neposredno. Če se postavimo v neko poljubno mesto, lahko takole odkrijemo vsa mesta, ki so povezana z njim: neko mesto je povezano z našim, če je naš sosed, pa tudi, če je sosed nekega takega mesta, ki je povezano z našim. Vsakič, ko za neko mesto prvič odkrijemo, da je povezano z našim, moramo torej pregledati še njegove sosede, saj so tudi oni povezani z našim mestom. Da ne bi istega mesta odkrivali po večkrat (če se da na primer od našega mesta do tistega priti po več poteh), si v neki tabeli (Obiskani v spodnjem programu) zapomnimo, če smo ga že odkrili. Spodnji program uporablja tabelo Sklad, da si zapomni, katera mesta je sicer že odkril, ni pa še preiskal njihovih sosedov. Ko neko mesto prvič odkrijemo, ga torej dodamo na ta sklad, nato pa bo prej ali slej že prišlo na vrsto, da si bomo ogledali še njegove sosede. Ko ni na skladu nobenega mesta več, pomeni, da smo odkrili vsa mesta, ki so povezana z našim. To je torej ena skupina mest, zdaj pa lahko vsa ta mesta v mislih pobrišemo (dovolj bo že to, da so v tabeli Obiskani označena kot obiskana) in se posvetimo kakšnemu drugemu mestu, s katerim doslej še nismo imeli opravka. Tako bomo odkrili naslednjo skupino in tako dalje; ko pa so označena vsa mesta, je naše delo končano.

Za zgoraj opisani postopek je torej koristno, če imamo za vsako mesto pri roki seznam njegovih sosedov. To lahko brez težav pripravimo iz seznama neposrednih povezav, ki ga preberemo iz vhodne datoteke: neposredna povezava od $u$ do $v$ pomeni, da je $v$ sosed mesta $u$, mesto $u$ pa je sosed mesta $v$. V tabeli StSosedov bomo hranili podatek o tem, koliko sosedov ima posamezno mesto. Sezname sosedov bomo zaradi učinkovitosti in preprostosti hranili kar v eni sami dolgi tabeli: na začetku bodo sosedje prvega mesta, nato sosedje drugega mesta in tako naprej. Sosedje mesta $u$ se začnejo na indeksu PrviSosed[$u$]; teh vrednosti ni težko izračunati s pomočjo tabele StSosedov. Spodnji program upošteva še to, da bo moral teči v realnem načinu, kjer posamezna tabela ne more biti daljša od 64 KB. Če ima naše omrežje 30 000 neposrednih povezav, pomeni to skupno 60 000 sosedov, vsak od njih pa zasede v pomnilniku 2 byta, tako da bomo morali to tabelo razbiti na dva kosa. Zato element, ki bi moral priti na indeks x, hranimo v resnici v celici Sosedje[x **mod** 2]↑[x **div** 2].

```
program SkupineMest;
const
  MaxM = 30000; { Največje možno število mest. }
  MaxN = 30000; { Največje možno število neposrednih povezav. }
type
  SosedjeT = array [0..MaxN − 1] of integer;
```

```
  PSosedjeT = ↑SosedjeT;
  StSosedovT = array [1..MaxM] of word;
  ObiskaniT = array [1..MaxM] of boolean;
var
  { i-ta neposredna povezava povezuje mesti PovOd↑[i] in PovDo↑[i]. }
  PovOd, PovDo: PSosedjeT;
  { Sklad in tabelo Obiskani uporabljamo med preiskovanjem omrežja. }
  Sklad: PSosedjeT; Obiskani: ↑ObiskaniT;
  { Pri n neposrednih povezavah je lahko skupna dolžina vseh seznamov sosedov
    do 2 * n. Tolikšna tabela ne bi šla v en 64-KB segment, zato jo razbijmo na
    dva kosa. Element x zapišimo v Sosedje[x mod 2]↑[x div 2]. Sosedje mesta i
    so na indeksih od PrviSosed↑[i] do PrviSosed↑[i] + StSosedov↑[i] − 1. }
  Sosedje: array [0..1] of PSosedjeT;
  StSosedov, PrviSosed: ↑StSosedovT;
  f: text; n, m, i, u, v, SP, StSkupin: integer; j: word;
begin
  { Preberimo podatke o cestnem omrežju. }
  New(Sosedje[0]); New(Sosedje[1]);
  New(PovOd); New(PovDo); New(StSosedov); New(PrviSosed);
  Assign(f, 'input.txt'); Reset(f);
  ReadLn(f, m); ReadLn(f, n);
  for i := 1 to m do StSosedov↑[i] := 0;
  for i := 1 to n do begin
    ReadLn(f, u, v); PovOd↑[i] := u; PovDo↑[i] := v;
    StSosedov↑[u] := StSosedov↑[u] + 1;
    StSosedov↑[v] := StSosedov↑[v] + 1;
  end; {for}
  Close(f);
  { Zdaj za vsako mesto vemo, s kolikimi je neposredno povezano.
    Pripravimo sezname sosedov za vsako mesto. }
  PrviSosed↑[1] := 1;
  for i := 1 to m − 1 do PrviSosed↑[i + 1] := PrviSosed↑[i] + StSosedov↑[i];
  for i := 1 to m do StSosedov↑[i] := 0;
  for i := 1 to n do begin
    u := PovOd↑[i]; v := PovDo↑[i];
    { Dodajmo v med u-jeve sosede in u med v-jeve. Lahko je u = v in
      zato moramo StSosedov↑[u] povečati, še preden uporabimo StSosedov↑[v].
      No, lahko bi tudi že med branjem vhodne datoteke preskočili vse povezave
      od u do u in na koncu branja primerno zmanjšali n. }
    j := PrviSosed↑[u] + StSosedov↑[u];
    StSosedov↑[u] := StSosedov↑[u] + 1;
    Sosedje[j mod 2]↑[j div 2] := v;
    j := PrviSosed↑[v] + StSosedov↑[v];
    StSosedov↑[v] := StSosedov↑[v] + 1;
    Sosedje[j mod 2]↑[j div 2] := u;
  end; {for}
```

```
  Dispose(PovOd); Dispose(PovDo); New(Sklad); New(Obiskani);
  { Zdaj lahko poiščemo in preštejemo skupine povezanih mest. }
  StSkupin := 0; for i := 1 to m do Obiskani↑[i] := false;
  for i := 1 to m do if not Obiskani↑[i] then begin
    { Preglejmo vsa mesta, povezana z i. Na skladu si bomo zapomnili,
      kaj moramo še pregledati. SP je število mest na skladu. }
    StSkupin := StSkupin + 1; SP := 1; Sklad↑[SP] := i; Obiskani↑[i] := true;
    while SP > 0 do begin
      u := Sklad↑[SP]; SP := SP - 1;
      { Obiščimo zdaj vse u-jeve sosede, če jih nismo obiskali že kdaj prej. }
      for j := PrviSosed↑[u] to PrviSosed↑[u] + StSosedov↑[u] - 1 do begin
        v := Sosedje[j mod 2]↑[j div 2];
        if not Obiskani↑[v] then
          begin SP := SP + 1; Sklad↑[SP] := v; Obiskani↑[v] := true end;
      end; {for j}
    end; {while}
  end; {for i}
  { Izpišimo rezultate, počistimo pomnilnik. }
  Assign(f, 'output.txt'); Rewrite(F); WriteLn(f, StSkupin); Close(f);
  Dispose(Sosedje[0]); Dispose(Sosedje[1]); Dispose(StSosedov);
  Dispose(PrviSosed); Dispose(Sklad); Dispose(Obiskani);
end. {SkupineMest}
```

Ta program se ne meni za napake v vhodnih podatkih, ki jih omenja opomba na strani 10. Če bi hoteli dovoliti tudi mesto s številko 0, bi morali v definiciji tipov StSosedovT in ObiskaniT zamenjati 1..MaxM z 0..MaxM; zanke, ki gredo od 1 do $m$, bi morale iti od 0 do $m$; PrviSosed↑[1] := 0 bi zamenjali s PrviSosed↑[0] := 0; in da mesto s številko 0 v primerih, ko ga sploh ni v omrežju, ne bi dobilo samostojne skupine, bi morali na koncu pogledati, če je StSosedov↑[0] = 0 in v tem primeru zmanjšati StSkupin za 1.

N: 10

## R1997.Z.2

Mislimo si v ravnini nek pravokotnik velikosti $a \times b$, s stranicami, vzporednimi s koordinatnima osema; skratka takega, s kakršnimi se ukvarja naša naloga. Recimo, da se njegova leva stranica ne dotika nobene izmed danih $n$ točk. Potem bi lahko ta pravokotnik premaknili malo v desno (toliko, da bi se začela leva stranica dotikati kakšne točke), pa ne bi bilo zaradi tega premika v njem nič manj točk kot prej, saj ni na levi strani nobena točka padla iz njega, na desni pa je vanj mogoče celo prišla kakšna nova. Enako bi lahko razmišljali za premik navzgor, če se spodnja stranica pravokotnika prvotno ni dotikala nobene točke. Ta razmislek nam pove, da je dovolj, če se pri iskanju pravokotnika, ki vsebuje največ točk, omejimo na tiste, ki imajo tako na spodnji kot na levi stranici kakšno točko (ni pa nujno, da sta to dve različni točki — lahko je ena sama, če je ravno v spodnjem levem oglišču pravokotnika).

Recimo, da nas trenutno zanimajo pravokotniki, ki se z levo stranico dotikajo točke $(x_0, y_0)$. Ker je višina pravokotnikov vedno $b$, mora biti $y$-koordinata spodnje stranice v tem primeru z intervala $[y_0 - b, y_0]$ (če bi bila nižje, bi bil cel pravokotnik prenizko, da bi se še lahko dotikal točke $(x_0, y_0)$, če bi bila višje, pa bi se pravokotnik začel previsoko), $y$-koordinata zgornje stranice pa z $[y_0, y_0 + b]$. Širina naših pravokotnikov pa je $a$; torej, ko se zanimamo za pravokotnike s točko $(x_0, y_0)$ na levi stranici, bomo imeli opravka le s točkami s področja $[x_0, x_0 + a] \times [y_0 - b, y_0 + b]$.

Recimo zdaj, da smo nekako prišli do seznama vseh točk s tega področja (v skrajni sili bi lahko naredili kar zanko po vseh $n$ točkah in za vsako posebej preverili, če leži v njem): recimo tem točkam $(x_i, y_i)$ za $i = 1, \ldots, m$ (med njimi je tudi točka $(x_0, y_0)$. Pravokotniki, ki nas bodo zanimali, morajo imeti eno od njih na spodnji stranici; čim si izberemo to, je položaj pravokotnika že čisto določen in je treba le še prešteti, katere od teh točk so v njem. Lahko bi šli torej z $i$ po vseh tistih indeksih, ki imajo $y_i \le y_0$ (saj vemo, da spodnja stranica ne sme biti višje od $y_0$, če hočemo imeti točko $(x_0, y_0)$ na levi stranici), in pri vsakem prešteli, katere izmed teh $m$ točk so v pravokotniku velikosti $a \times b$, ki ima $(x_0, y_0)$ na levi in $(x_i, y_i)$ na desni stranici (to so obenem tudi že vse točke v tem pravokotniku, saj leži le teh $m$ točk na takem področju, da bi utegnile ležati v tem pravokotniku).

Zdaj torej že imamo preprosto rešitev naloge:

```
program Rozine1;
const MaxN = 10000;
type  TabelaI = array [1..MaxN] of integer;
      TabelaR = array [1..MaxN] of real;
var   PtX, PtY: ↑TabelaR;                { koordinate točk }
      N: integer;                        { število točk }
      A, B: real;                        { velikost poizvedovalnega okna }
      Neigh: ↑TabelaI; nNeigh: integer;  { okolica točke (X0, Y0) }
      i, j, k, Count, Best: integer; F: text;
      X0, Y0, Y1, YCur: real;
begin
  { Preberimo vhodne podatke. }
  New(PtX); New(PtY);
  Assign(F, 'input.txt'); Reset(F);
  ReadLn(F, A, B); ReadLn(F, N);
  for i := 1 to N do ReadLn(F, PtX↑[i], PtY↑[i]);
  Close(F);
  { Pojdimo po vseh točkah. }
  Best := 0; New(Neigh);
  for i := 1 to N do begin
    { Pregledali bomo pravokotnike z i-to točko, (X0, Y0), na levi stranici.
      V Neigh pripravimo točke z X0 ≤ X ≤ X0 + A, Y0 − B ≤ Y ≤ Y0 + B;
      samo te lahko namreč ležijo v kakšnem takem pravokotniku. }
```

```
    X0 := PtX↑[i]; Y0 := PtY↑[i]; nNeigh := 0;
    for j := 1 to N do
      if    (X0 <= PtX↑[j])       and (PtX↑[j] <= X0 + A)
      and  (Y0 − B <= PtY↑[j]) and (PtY↑[j] <= Y0 + B)
      then begin nNeigh := nNeigh + 1; Neigh↑[nNeigh] := j end;
    { Poskusimo vse možne položaje spodnje stranice pravokotnika. }
    for j := 1 to nNeigh do begin
      Count := 0; Y1 := PtY↑[Neigh↑[j]];
      if Y1 > Y0 then
        continue; { Brez tega pogoja porabi program precej več časa! }
      { Koliko točk pokriva pravokotnik s spodnjo stranico pri Y1? }
      for k := 1 to nNeigh do begin
        YCur := PtY↑[Neigh↑[k]];
        if (Y1 <= YCur) and (YCur <= Y1 + B) then Count := Count + 1;
      end; {for k}
      if Count > Best then Best := Count;
    end; {for j}
  end; {for i}
  { Izpišimo rezultat in počistimo pomnilnik. }
  Assign(F, 'output.txt'); Rewrite(F); WriteLn(F, Best); Close(F);
  Dispose(Neigh); Dispose(PtX); Dispose(PtY);
end. {Rozine1}
```

Vendar pa je lahko ta rešitev, če imamo smolo, precej neučinkovita. Če je poizvedovalno območje $a \times b$ dovolj veliko, točke pa dovolj blizu skupaj, se bo pri mnogih (recimo pri $O(n)$) vrednostih $i$ zgodilo, da bo v okolici $(x_0, y_0)$ veliko (recimo $O(n)$) točk, tako da bosta gnezdeni zanki po $j$ in $k$ vsakič zahtevali $O(n^2)$ časa in je skupna časovna zahtevnost našega postopka zato v najslabšem primeru $O(n^3)$, kar je že zelo počasi. (Preprosteje povedano, naš program ima tri gnezdene zanke, ki gredo lahko v najslabšem primeru vse od 1 do $n$.) Če je v okolici točke $(x_0, y_0)$ (torej: v pravokotniku $[x_0, x_0 + a] \times [y_0 - b, y_0 + b]$) povprečno $m$ točk, pa imata gnezdeni zanki po $j$ in $k$ skupaj zahtevnost $O(m^2)$, pred tem pa porabimo še $O(n)$ časa, da ugotovimo, katere točke so v okolici; zahtevnost celotnega postopka je zato $O(n(n + m^2))$.

Postopek bi lahko izboljšali takole: ko postavljamo spodnjo stranico pravokotnika na vse možne točke v področju $[x_0, x_0+a]\times[y_0-b, y_0]$, moramo pri vsaki prešteti, koliko točk potem vsebuje tak pravokotnik. Toda če se spodnja stranica le malo premakne, vsebuje pravokotnik še vedno približno iste točke kot prej. Če bi torej točke, na katere postavljamo spodnjo stranico, pregledovali po naraščajočih $y$-koordinatah, bi ob vsakem premiku spodnje stranice kakšna točka izpadla iz pravokotnika, vrhnji del pravokotnika bi pokril nekaj novih, večina pa bi ostala nespremenjenih. Izpadla bi le tista, ki je bila pri prejšnjem položaju ravno na spodnji stranici (po novem pa se je spodnja stranica premaknila na naslednjo višjo točko in torej tista prejšnja ni več v pravokotniku), na vrhu pa bi prišlo vanjo nekaj novih. Zdaj torej za štetje, katere so v pra-

vokotniku, ni treba uporabiti zanke s števcem $k$, ki bi šla po vseh $m$ točkah v okolici, pač pa lahko $k$ le pogleda, koliko točk je na novo prišlo v pravokotnik. Zdaj zanki s števcema $j$ in $k$ pravzaprav potekata istočasno (prepleteno), ne pa vgnezdeno; v času, ko pride $j$ od 1 do $m$, pride tudi $k$ ravno enkrat od 1 do $m$. Urejanje točk po $y$-koordinatah lahko opravimo na samem začetku, čim preberemo točke iz vhodne datoteke; nato bomo že ob pripravi soseščine (prva zanka po $j$) dobili sosednje točke v naraščajočem vrstnem redu $y$-koordinat. Časovna zahtevnost tako spremenjenega postopka je le še $O(n^2)$ (za urejanje porabimo največ $O(n^2)$, s kakšnim učinkovitejšim algoritmom še manj; nato pa pri vsaki točki porabimo $O(n)$, da ugotovimo, katere so v njeni okolici, in nato še $O(m)$, da se z intervalom višine $b$ sprehodimo čez te okoliške točke). Spodnji program vsebuje tri znane algoritme za urejanje točk po koordinatah: urejanje z izbiranjem (*selection sort*), urejanje z vstavljanjem (*insertion sort*) in quicksort; slednji je hitrejši od prvih dveh, vendar je tudi bolj zapleten.

```
program Rozine2;

const MaxN = 10000;
type  TabelaI = array [1..MaxN] of integer;
      TabelaR = array [1..MaxN] of real;
var   PtX, PtY: ↑TabelaR; { koordinate točk }
      N: integer;         { število točk }
      A, B: real;         { velikost poizvedovalnega okna }

  procedure SelectionSort; { Uredi točke po naraščajočih Y-koordinatah. }
  var i, j, k: integer; Y0, Y1, T: real;
  begin
    for i := 1 to N do begin
      { Naj bo k najnižja točka izmed točk i..n. }
      Y0 := PtY↑[i]; k := i;
      for j := i + 1 to N do begin
        Y1 := PtY↑[j];
        if Y1 < Y0 then begin Y0 := Y1; k := j end;
      end; {for j}
      { Zamenjajmo k s točko i. }
      T := PtX↑[i]; PtX↑[i] := PtX↑[k]; PtX↑[k] := T;
      T := PtY↑[i]; PtY↑[i] := PtY↑[k]; PtY↑[k] := T;
    end; {for i}
  end; {SelectionSort}

  procedure InsertionSort; { Uredi točke po naraščajočih Y-koordinatah. }
  var i, j: integer; X0, Y0: real;
  begin
    for i := 2 to N do begin
      { Zaporedje točk 1..i − 1 je že urejeno. Vstavimo vanj točko i. }
      j := i − 1; X0 := PtX↑[i]; Y0 := PtY↑[i];
```

```
        while j > 0 do begin
          if PtY↑[j] <= Y0 then break;
          PtX↑[j + 1] := PtX↑[j]; PtY↑[j + 1] := PtY↑[j]; j := j − 1;
        end; {while}
        PtX↑[j + 1] := X0; PtY↑[j + 1] := Y0;
      end; {for}
    end; {InsertionSort}

    procedure QuickSort(L, R: integer); { Uredi točke L..R po naraščajočih Y. }
    var i, j: integer; T, M: real;
    begin
      while L < R do begin
        M := PtY↑[(L + R) div 2]; i := L; j := R;
        { Razdelimo točke na tiste pod in tiste nad Y = M. }
        while i < j do begin
          { Invarianta: točke L..i imajo Y ≤ M, točke j..R pa Y ≥ M. }
          while PtY↑[i] < M do i := i + 1;
          while PtY↑[j] > M do j := j − 1;
          if i <= j then begin
            T := PtX↑[i]; PtX↑[i] := PtX↑[j]; PtX↑[j] := T;
            T := PtY↑[i]; PtY↑[i] := PtY↑[j]; PtY↑[j] := T;
            i := i + 1; j := j − 1;
          end; {if}
        end; {while}
        { Točke L..j imajo Y ≤ M, točke i..R pa Y ≥ M.
          Manjšo od teh dveh skupin uredimo z rekurzivnim klicem, večje pa
          se bomo lotili v naslednji ponovitvi zunanje zanke while. }
        if j − L < R − i then begin QuickSort(L, j); L := i end
        else begin QuickSort(i, R); R := j end;
      end; {while}
    end; {QuickSort}

var Neigh: ↑TabelaI; nNeigh: integer; { točke v okolici trenutne točke }

    { V Neigh vpiše točke z X0 ≤ X ≤ X0 + A, Y0 − B ≤ Y ≤ Y0 + B. }
    procedure FindNeigh1(X0, Y0: real);
    var i: integer;
    begin
      nNeigh := 0;
      for i := 1 to N do
        if    (X0 <= PtX↑[i])       and (PtX↑[i] <= X0 + A)
        and  (Y0 − B <= PtY↑[i]) and (PtY↑[i] <= Y0 + B)
        then begin nNeigh := nNeigh + 1; Neigh↑[nNeigh] := i end;
    end; {FindNeigh1}

var i, j, k, Best: integer; F: text; Y1: real;
begin {Rozine2}
```

```
  { Preberimo vhodne podatke. }
  New(PtX); New(PtY);
  Assign(F, 'input.txt'); Reset(F);
  ReadLn(F, A, B); ReadLn(F, N);
  for i := 1 to N do ReadLn(F, PtX↑[i], PtY↑[i]);
  Close(F);
  { Uredimo točke po naraščajočih Y-koordinatah. }
  InsertionSort; {QuickSort(1, N);} {SelectionSort;}
  { Pojdimo po vseh točkah. }
  Best := 0; New(Neigh);
  for i := 1 to N do begin
    { Preglejmo pravokotnike, ki imajo točko i na levi stranici. V Neigh pripravimo
      vse točke, ki bi utegnile biti v kakšnem takem pravokotniku. }
    FindNeigh1(PtX↑[i], PtY↑[i]);
    { Poskusimo vse možne položaje spodnje stranice pravokotnika. }
    k := 2;
    for j := 1 to nNeigh do begin
      Y1 := PtY↑[Neigh↑[j]];
      { Pravokotnik s spodnjo stranico pri Y1 pokriva točke Neigh↑[j..k − 1]. }
      while k <= nNeigh do
        if PtY↑[Neigh↑[k]] > Y1 + B then break else k := k + 1;
      if k − j > Best then Best := k − j;
    end; {for j}
  end; {for i}
  { Izpišimo rezultat in počistimo pomnilnik. }
  Assign(F, 'output.txt'); Rewrite(F); WriteLn(F, Best); Close(F);
  Dispose(Neigh); Dispose(PtX); Dispose(PtY);
end. {Rozine2}
```

Doslej smo, ko je bilo treba našteti točke iz območja $[x_0, x_0+a]\times[y_0-b, y_0+b]$, šli vsakič kar po vseh $n$ točkah in za vsako preverili, če leži v njem. Toda zdaj, ko imamo točke urejene po $y$-koordinatah, si lahko privoščimo naslednjo izboljšavo: z bisekcijo poiščemo najnižjo točko, ki ima $y \geq y_0 - b$, nato pa pojdimo po vrsti od te točke naprej in za vsako preverimo, če ustreza tudi pogoju $x_0 \leq x \leq x_0 + b$. Ko pridemo do kakšne z $y > y_0 + b$, pa lahko takoj nehamo, saj vemo, da ne bomo našli nobene točke s primernim $y$ več. Naštevanje točk v okolici nam zdaj v najslabšem primeru sicer še vedno lahko vzame $O(n)$ časa (če je veliko točk na pasu $y_0 - b \leq y \leq y_0 + b$), v praksi pa bo šlo najbrž vendarle hitreje (bisekcija sama zahteva le $O(\lg n)$ časa). V gornjem programu Rozine2 bi morali funkcijo FindNeigh1 zamenjati z:

```
procedure FindNeigh2(X0, Y0: real);
var i, L, R: integer;
begin
  { Bisekcija. }
  L := 1; R := N + 1;
```

```
    while L + 1 < R do begin
      { Invarianta: L = 1 ali pa PtY↑[L] ≤ Y0 − B < PtY↑[R]. }
      i := (L + R) div 2;
      if PtY↑[i] <= Y0 − B then L := i else R := i;
    end; {while}
    { Točke z Y ≥ Y0 + B se nahajajo na indeksih od L naprej. }
    nNeigh := 0;
    while L <= N do begin
      if PtY↑[L] > Y0 + B then break;
      if (X0 <= PtX↑[L]) and (PtX↑[L] <= X0 + A) then
        begin nNeigh := nNeigh + 1; Neigh↑[nNeigh] := L end;
      L := L + 1;
    end; {while}
  end; {FindNeigh2}
```

Razmislimo še o naslednji izboljšavi: ko razmišljamo o pravokotnikih, ki imajo na levi stranici točko $(x_0, y_0)$, nas zanimajo le točke iz pasu $x_0 \le x \le x_0 + a$; med temi točkami pa nas zanimajo le tiste, za katere velja še $y_0 - b \le y \le y_0 + a$. Da bomo poceni prišli do teh slednjih, je koristno imeti točke pasu urejene po $y$-koordinatah; da pa ne bi bilo treba pri vsakem $x_0$ znova ugotavljati, katere točke ležijo v pasu, je koristno točke pregledovati po naraščajočem $x_0$. Tako se pas pravzaprav počasi premika proti desni in ob vsakem premiku iz njega izpade dotedanja točka $x_0$, na desni strani pa lahko v pas pride kakšna nova točka. Od podatkovne strukture, s katero bomo predstavili množico točk v pasu $x_0 \le x \le x_0 + a$, torej pričakujemo učinkovite operacije za dodajanje in brisanje točk ter učinkovito naštevanje točk, ki ustrezajo pogoju $y_0 - b \le y \le y_0 + b$. Zato je koristno uporabiti binarno iskalno drevo, v katerem bodo točke urejene po naraščajoči $y$-koordinati. Takšno drevo podpira brisanje in dodajanje v času $O(\lg n)$, naštevanje točk z intervala $[y_0 - b, y_0 + b]$ pa v $O(m + \lg n)$, če je treba našteti $m$ točk.[7] Zdaj imamo torej tak postopek: ravnino „pometamo“ s pasom $x_0 \le x \le x_0 + a$; ko se pas premakne, pade ena točka iz njega, vanj pa pride nič ali več novih; na novem položaju naštejemo točke, ki ležijo na pasu in obenem ustrezajo pogoju $y_0 - b \le y \le y_0 + b$; prek teh pa se potem sprehodimo z dvema prepletenima zankama, tako kot pri prejšnji rešitvi. Zdaj imamo torej vsega skupaj $O(n)$ operacij z drevesom (vsako točko enkrat dodamo vanj in enkrat zbrišemo iz njega), kar zahteva skupaj $O(n \lg n)$ časa; toliko časa potrebujemo tudi za urejanje (če nimamo smole s quicksortom); in pri vsaki točki je še $O(m + \lg n)$ dela za naštevanje

[7] To dvoje se zanaša na predpostavko, da se nam drevo ne bo preveč izrodilo, ampak to se v praksi konec koncev res ne dogaja pretirano pogosto; če bi hoteli biti res varni, bi morali uporabiti kakšno bolj zapleteno različico binarnega iskalnega drevesa, npr. AVL-drevo ali rdeče-črno drevo. Lahko pa bi namesto tega (po zgledu drevesa segmentov iz rešitve naloge 1998.2.3) izkoristili dejstvo, da že vnaprej točno vemo, kakšnih $n$ možnih vrednosti utegne imeti $y$-koordinata točke, ki jo bomo dodali v drevo.

okoliških točk in sprehod čeznje z intervalom višine $b$. Tako imamo skupaj časovno zahtevnost $O(n(m + \lg n))$.

```
program Rozine3;
const MaxN = 10000;
type  TabelaI = array [1..MaxN] of integer;
      TabelaR = array [1..MaxN] of real;
      PTabelaI = ↑TabelaI; PTabelaR = ↑TabelaR;
var   PtX, PtY: ↑TabelaR; { koordinate točk }
      N: integer;         { število točk }
      A, B: real;         { velikost poizvedovalnega okna }
      { binarno iskalno drevo, ki vsebuje točke trenutnega pasu }
      Left, Right, Parent: ↑TabelaI; Root: integer; { urejene so po Y-koordinatah }
      { točke, urejene po eni od koordinat }
      XOrder, YOrder: ↑TabelaI; { v XOrder so vse točke, v YOrder pa neka okolica }

  { Uredi elemente tabele Order↑[L..R]. Elementi naj bodo indeksi v tabelo Coord,
    primerja pa se jih po ustrezni vrednosti v Coord. }
  procedure QuickSort(Coord: PTabelaR; Order: PTabelaI; L, R: integer);
  var i, j, T: integer; M: real;
  begin
    while L < R do begin
      M := Coord↑[Order↑[(L + R) div 2]]; i := L; j := R;
      while i < j do begin
        while Coord↑[Order↑[i]] < M do i := i + 1;
        while Coord↑[Order↑[j]] > M do j := j − 1;
        if i <= j then begin T := Order↑[i]; Order↑[i] := Order↑[j];
                             Order↑[j] := T; i := i + 1; j := j − 1 end;
      end; {while}
      if j − L < R − i then begin QuickSort(Coord, Order, L, j); L := i end
      else begin QuickSort(Coord, Order, i, R); R := j end;
    end; {while}
  end; {QuickSort}

  { V Dest doda (in poveča Count) vsa vozlišča (v poddrevesu, ki se
    začne pri Node), ki imajo Y-koordinato vsaj YMin in največ YMax. }
  procedure TreeQuery(Dest: PTabelaI; var Count: integer;
                      YMin, YMax: real; Node: integer);
  var YNode: real;
  begin
    YNode := PtY↑[Node];
    if (Left↑[Node] > 0) and (YMin <= YNode) then
      TreeQuery(Dest, Count, YMin, YMax, Left↑[Node]);
    if (YMin <= YNode) and (YNode <= YMax) then
      begin Count := Count + 1; Dest↑[Count] := Node end;
    if (Right↑[Node] > 0) and (YNode <= YMax) then
      TreeQuery(Dest, Count, YMin, YMax, Right↑[Node]);
```

```
  end; {TreeQuery}

  procedure TreeAdd(Node: integer);       { Doda vozlišče Node v drevo. }
  var P: integer; YParent, YNode: real;
  begin
    Left↑[Node] := 0; Right↑[Node] := 0;
    if Root = 0 then begin Root := Node; exit end;
    P := Root; YNode := PtY↑[Node];
    repeat
      YParent := PtY↑[P];
      if YNode < YParent then begin
        if Left↑[P] = 0 then begin Left↑[P] := Node; break end;
        P := Left↑[P];
      end else begin
        if Right↑[P] = 0 then begin Right↑[P] := Node; break end;
        P := Right↑[P];
      end; {if}
    until false;
    Parent↑[Node] := P;
  end; {TreeAdd}

  procedure TreeDel(Node: integer);       { Zbriše vozlišče Node iz drevesa. }
  var LC, RC, Next, P: integer;
  begin
    LC := Left↑[Node]; RC := Right↑[Node]; P := Parent↑[Node];
    if (LC = 0) or (RC = 0) then begin
      if LC = 0 then Next := RC else Next := LC;
    end else begin
      Next := RC;
      while Left↑[Next] > 0 do Next := Left↑[Next];
      TreeDel(Next);
      RC := Right↑[Node];
      Parent↑[LC] := Next; if RC > 0 then Parent↑[RC] := Next;
      Left↑[Next] := LC; Right↑[Next] := RC;
    end; {if}
    if Next > 0 then Parent↑[Next] := P;
    if P = 0 then Root := Next
    else if Left↑[P] = Node then Left↑[P] := Next
    else { Right↑[P] = Node } Right↑[P] := Next;
  end; {TreeDel}

var i, Best: integer; F: text;
    X0, Y0: real;           { koordinati trenutne točke; X0 določa levi rob pasu }
    Y1: real;               { trenutna koordinata spodnjega roba pravokotnika }
    XFrom, XTo: integer;    { v pasu so točke XOrder↑[XFrom..XTo − 1] }
    YCount: integer;        { koliko točk pasu ima Y-koordinato blizu Y0 }
    YFrom, YTo: integer;    { trenutni pravokotnik X0 ≤ X ≤ X0 + A, Y1 ≤ Y ≤ Y0 + B
```

```
                                    pokriva točke YOrder↑[YFrom..YTo − 1] }
begin {Rozine3}
  { Preberimo vhodne podatke. }
  New(Left); New(Right); New(Parent);
  New(XOrder); New(YOrder); New(PtX); New(PtY);
  Assign(F, 'input.txt'); Reset(F);
  ReadLn(F, A, B); ReadLn(F, N);
  for i := 1 to N do ReadLn(F, PtX↑[i], PtY↑[i]);
  Close(F);
  { Uredimo točke po naraščajočem X. }
  for i := 1 to N do XOrder↑[i] := i;
  QuickSort(PtX, XOrder, 1, N);

  { Drevo je na začetku prazno. Pometajmo ravnino.}
  { Začetni položaj levega roba pasu je tak, da gre skozi najbolj levo točko. }
  Best := 1; XFrom := 0; XTo := 1; Root := 0;
  while XFrom <= N − Best do begin

    { Premaknimo levi rob pasu do naslednje točke (proti desni).
      Prejšnja zato izpade iz pasu, lahko pa vanj pride nekaj novih. }
    if XFrom > 0 then TreeDel(XOrder↑[XFrom]);
    XFrom := XFrom + 1; X0 := PtX↑[XOrder↑[XFrom]];
    while XTo <= N do begin
      if PtX↑[XOrder↑[XTo]] > X0 + A then break;
      TreeAdd(XOrder↑[XTo]); XTo := XTo + 1;
    end; {while}
    if XTo − XFrom <= Best then continue; { Brezupno. }

    { Zanimali nas bodo pravokotniki, ki se z levo stranico dotikajo
      točke XOrder↑[XFrom]. Pripravimo si seznam točk,
      ki so v pasu in imajo Y0 − B ≤ Y ≤ Y0 + B. }
    Y0 := PtY↑[XOrder↑[XFrom]]; YCount := 0;
    TreeQuery(YOrder, YCount, Y0 − B, Y0 + B, Root);
    if YCount <= Best then continue; { Brezupno. }

    { Preletimo ta seznam z intervalom širine B. }
    YFrom := 0; YTo := 1;
    while (YFrom <= YCount − Best) and (YTo <= YCount) do begin
      { Interval bo pokrival koordinate od Y1 do Y1 + B. }
      YFrom := YFrom + 1; Y1 := PtY↑[YOrder↑[YFrom]];
      while YTo <= YCount do
        if PtY↑[YOrder↑[YTo]] > Y1 + B then break else YTo := YTo + 1;
      { Na intervalu so točke YOrder↑[YFrom..YTo − 1]. }
      if YTo − YFrom > Best then Best := YTo − YFrom;
    end; {prelet seznama YOrder z intervalom višine B}

  end; {prelet ravnine s pasom širine A}

  { Izpišimo rezultat in počistimo pomnilnik. }
  Assign(F, 'output.txt'); Rewrite(F); WriteLn(F, Best); Close(F);
```

```
  Dispose(Left); Dispose(Right); Dispose(Parent);
  Dispose(XOrder); Dispose(YOrder); Dispose(PtX); Dispose(PtY);
end. {Rozine3}
```

Opisane rešitve smo preizkusili na računalniku s procesorjem PIII 800 MHz; testni podatki so bili prav taki, kot so jih uporabljali na tekmovanju leta 1997. Spodnje meritve kažejo pravzaprav le porabo procesorjevega časa, saj odpade zaradi dovolj velikega diskovnega predpomnilnika na branje vhodnih podatkov in izpisovanje rezultatov le zanemarljivo malo časa. Program je bil prilagojen tako, da je v zanki obdelal vseh deset testnih primerov in ga ni bilo treba poganjati za vsak testni primer posebej. Pognali smo ga večkrat (desetkrat pri Rozine1 in Rozine2 + FindNeigh1 ter petdesetkrat pri Rozine2 + FindNeigh2 in Rozine3) in izračunali povprečni čas obdelave vseh desetih testnih primerov.

| | | | | |
|---|---|---|---|---|
| Rozine1 | 9,70 s | | | |
| Rozine2 | | SelectionSort | InsertionSort | QuickSort |
| FindNeigh1 | | 5,28 s | 4,66 s | 4,52 s[8] |
| FindNeigh2 | | 2,02 s | 1,40 s | 0,337 s |
| Rozine3 | 0,200 s | | | |

Podobna naloga, le da temelji na diskretni mreži, je tudi #10360 v zbirki na `online-judge.uva.es` (4. naloga s študentskega tekmovanja tehnične univerze v Darmstadtu, 23. junija 2001).

[8] Tu se bo bralec mogoče vprašal, zakaj je razlika med InsertionSort in QuickSort v kombinaciji s FindNeigh1 tako majhna, v kombinaciji s FindNeigh2 pa tako velika. V resnici je razlika med trajanjem urejanja v obeh primerih enaka, težava pa je v tem, da porabi FindNeigh1 za svoje delo precej več časa, če so bili podatki urejeni s quicksortom, kot pa če so bili urejeni z vstavljanjem. Levji delež te razlike med InsertionSort + FindNeigh1 in QuickSort + FindNeigh1 nastopi pri enem samem patološkem testnem primeru, v katerem so točke kar vsi pari $(x, y)$ za $x, y \in \{1, \ldots, 100\}$, v vhodni datoteki pa so navedeni po naraščajočih $x$ in pri vsakem $x$ še po naraščajočih $y$. Urejanje z vstavljanjem je stabilno in torej s tem, ko uredi točke po $y$, ostanejo pri vsakem $y$ urejene tudi po naraščajočem $x$; quicksort pa jih sicer uredi po $y$, vendar jih ob tem znotraj vsakega $y$ tudi pošteno premeče, tako da niso več urejene naraščajoče po $x$. Ker se izvedejo v stavku **if** v FindNeigh1 obakrat natanko iste primerjave pri istih i (da nam ne bi prevajalnik mešal štrene, smo tisti pogoj celo razcepili v več stavkov oblike **if not** DelPogoja **then continue**), si razlike v času izvajanja ne znamo razlagati drugače kot s tem, da je pri dostopanju do podatkov, urejenih s quicksortom, procesor pogosteje čaka, da bo dobil podatke iz pomnilnika ali predpomnilnika. To potrjuje tudi naslednji poskus: če v pogoju stavka **if** v podprogramu FindNeigh1 najprej preverimo koordinato $y$ in šele nato koordinato $x$, je čas izvajanja enak ne glede na postopek, s katerim smo podatke urejali; to je smiselno, saj v tem primeru uspe pogoj glede koordinate PtY↑[i] obakrat pri natanko istih i in zato tudi koordinato PtX↑[i] preberemo obakrat pri natanko istih i, tako da je vzorec dostopov do predpomnilnika obakrat enak (je pa čas izvajanja podprograma FindNeigh1 zdaj počasnejši kot prej, celo kot prej pri quicksortu). Kakorkoli že, če QuickSort spremenimo tako, da primerja dve točki še po $x$, če sta imeli enak $y$, traja FindNeigh1 potem prav toliko, kot če bi uporabili urejanje z vstavljanjem, tako da se številka 4,52 s v gornji tabeli spremeni v 3,69 s.

## R1997.Z.3

N: 11

Radi bi, da bi bile vse števke na velikem in malem števcu različne. Ker na veliki števec ne moremo vplivati drugače kot z vožnjo, moramo vsekakor prevoziti toliko kilometrov, da bo število na velikem števcu sestavljeno iz samih različnih števk. Ko pridemo do kakega takega števila (recimo $x$), nam ostanejo še štiri števke, ki se ta hip na velikem števcu ne pojavljajo in morajo torej nastopiti na malem števcu. Za mali števec obstaja torej zdaj (pri trenutnem stanju velikega števca) $4! = 24$ možnih vrednosti.[9] Za vsako od njih moramo preveriti, ali bi lahko mali števec nekako spravili v to stanje nekoč v tistem času, ko bo imel veliki števec vrednost $x$. Tu sta dve možnosti. (1) Do vrednosti $m$ lahko mali števec pride, če smo prevozili vsaj $\lfloor m/10 \rfloor$ kilometrov, saj lahko v tem primeru mali števec v primernem trenutku (namreč ravno $\lfloor m/10 \rfloor$ kilometrov prej, preden je dosegel veliki števec svojo trenutno vrednost $x$) resetiramo, pa bo imel še dovolj časa, da bo prilezel od 0000 do vrednosti $10\lfloor m/10 \rfloor$ ravno v trenutku, ko bo veliki števec dosegel vrednost $x$; nato pa lahko prevozimo še ($m$ mod 10)-krat po sto metrov, pa bo veliki števec še vedno imel isto vrednost, mali pa bo prišel na želeno vrednost $m$. (2) Druga možnost je, da imamo srečo in se mali števec znajde v pravem stanju že kar sam od sebe, ne da bi ga kaj spreminjali. Če je imel na začetku vrednost $m_0$, mi pa smo prevozili $d$ kilometrov, bo imel v trenutku, ko dobi veliki števec vrednost $x$, mali števec vrednost $M := (m_0 + 10d) \bmod 10000$, v okviru tega kilometra pa bo dobil še naslednjih devet vrednosti, torej do vključno $M' := (m_0 + 10d + 9) \bmod 10000$, preden se bo veliki števec premaknil naprej (na $x + 1$). Zdaj moramo torej preveriti, če je želena vrednost $m$ nekje na tem intervalu: $M \le m \le M'$. Posebej bi morali obravnavati še primer, ko pade malemu števcu vrednost z 9999 na 0000; takrat je $M'$ manjše od $M$, tako da je vrednost $m$ mogoče doseči, če je ali $M \le m$ ali pa $m \le M'$. Toda v tem primeru je $M$ zagotovo $\ge 9991$ (sicer še ne bi prišlo do padca na 0000), $M'$ pa je zagotovo $\le 0008$ (to vrednost dosežemo, če je bil $M = 9999$; če je bil $M$ manjši, pa še tega ne). Tedaj imajo torej vsi $m$, ki ustrezajo eni od enačb $M \le m$ in $m \le M'$, prve tri števke enake, tak pa naš iskani $m$ zagotovo ne bo, saj mora vendar imeti štiri različne števke. Zato se nam s tem posebnim primerom v resnici ni treba ukvarjati.

Spodnji program kar preprosto povečuje veliki števec v korakih po en kilometer, nato pa vsakič preveri, če ga sestavlja šest različnih števk; če je tako, pregleda vseh 24 možnih stanj malega števca in za vsako od njih preveri, če ga je mogoče kako doseči. Mogoče bi si lahko poskušali možne vrednosti velikega števca (torej take, ki imajo šest različnih števk) pripraviti tudi vnaprej v kakšni tabeli, da ne bi števca povečevali v korakih po en kilometer, ampak

[9] Na primer: če imamo na voljo števke 1, 2, 3 in 4, so možne naslednje vrednosti malega števca: 1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4321. Mimogrede, pri tem razmisleku bomo stanje malega števca vedno gledali kot celo število, torej kot da v njem ne bi bilo decimalne vejice.

bi vedno kar skočili na naslednjo vrednost s šestimi različnimi števkami; toda takih vrednosti je kar $10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 = 151200$, torej sploh ne malo, tako da je vprašljivo, če bi s tem res kaj prihranili.

Za naštevanje možnih vrednosti malega števca (pri danih štirih števkah, iz katerih mora biti sestavljen) moramo nekako priti do vseh 24 permutacij teh štirih števk. Čisto dobro bi bilo že, če bi v kakšni tabeli navedli vseh 24 načinov, kako razmestiti štiri števke. Lahko pa poskusimo permutacije tudi oštevilčiti od 0 do 23 in jih tako našteti. Tu ravnamo nekako tako, kot da bi imeli številski sestav s spremenljivo osnovo. Recimo, da bi gledali permutacije $k$ elementov. Poglejmo, kateri po velikosti (štejmo jih od 1 do $k-1$) je element na prvem mestu, torej $\pi(1)$; to pomnožimo s $(k-1)!$. Nato poglejmo, kateri po velikosti ($0..k-2$) izmed vseh preostalih $k-1$ elementov je tisti, ki je na drugem mestu v permutaciji; to vrednost (0 naj predstavlja najmanjšega, 1 drugega najmanjšega in tako naprej) pomnožimo s $(k-2)!$. Nato poglejmo, kateri po velikosti izmed preostalih $k-2$ elementov je element na tretjem mestu; to pomnožimo s $(k-3)!$. Tako nadaljujemo do konca in nazadnje vse te zmnožke seštejemo. Oglejmo si na primer razpored 8957. Tu imamo štiri elemente, torej $k = 4$. Na prvem mestu je drugi najmanjši element, kar nam da $1 \cdot (k-1)! = 6$. Ostanejo elementi $9, 5, 7$ in na drugem mestu je 9, ki je tretji najmanjši med njimi; to nam da $2 \cdot (k-2)! = 4$. Ostaneta elementa 5 in 7; na tretjem mestu je manjši od obeh, kar nam da $0 \cdot (k-3)! = 0$. Ostane element 7, ki je pač sam pri sebi neizogibno tudi najmanjši in nam da $0 \cdot (k-4)! = 0$. Če vse to seštejemo, dobimo $6 + 4 + 0 + 0 = 10$, torej razporedu 8957 ustreza število 10. Na enak način bi lahko oštevilčili tudi druge razporeditve števk 8, 9, 5 in 7 ter tako dobili vsa cela števila od 0 do $4! - 1 = 23$. Naš program pa mora uporabiti ravno obraten postopek, da bo iz števil $0, \ldots, 23$ dobil konkretne razporede. V ta namen številko razporeda najprej delimo s $(k-1)!$; celi del količnika nam pove, kateri najmanjši element je na prvem mestu, ostanek pa potem po enakem postopku dekodiramo v razpored na ostalih mestih (od drugega mesta naprej).

```
program Stevec;
var F: text; i, j, k: longint;
    Km, Mali, KmDelta, M: longint; R: real;
    Stevke, Stevke0: array [0..9] of integer;
    Ostale: set of 0..9; Nasel: boolean;
begin
  Assign(F, 'input.txt'); Reset(F);
  ReadLn(F, Km, R); Close(F);
  { Pozor: uporabiti moramo Round, ne Trunc. Slednji vedno zaokroža navzdol,
    kar ni dobro, saj je lahko kakšno decimalno število, npr. 0,1, predstavljeno
    z nekim približkom, ki je manjši od njega in je potem tudi njegov
    desetkratnik manjši od ustreznega celega števila. Če bi ga zaokrožili
    navzdol, bi dobili za 1 premajhno vrednost. }
```

```
  Mali := Round(10 * R); KmDelta := 0;
  while true do begin
    { Katere števke se ne pojavljajo na velikem števcu? }
    Ostale := [0..9]; j := Km;
    for i := 0 to 5 do begin Ostale := Ostale − [j mod 10]; j := j div 10 end;
    { Shranimo jih v tabelo Stevke0. }
    j := 0; for i := 0 to 9 do if i in Ostale then
      begin Stevke0[j] := i; j := j + 1 end;
    { Če so ostale samo štiri števke, jih poskusimo na vseh 4! = 24 načinov
      sestaviti v stanje malega števca. }
    Nasel := false;
    if j = 4 then for i := 0 to 23 do begin
      { Pripravimo (v številu M) i-to permutacijo naših štirih števk. }
      for j := 0 to 3 do Stevke[j] := Stevke0[j];
      j := i div 6; k := i mod 6; M := 1000 * Stevke[j];
      while j < 3 do begin Stevke[j] := Stevke[j + 1]; j := j + 1 end;
      j := k div 2; k := k mod 2; M := M + 100 * Stevke[j];
      while j < 2 do begin Stevke[j] := Stevke[j + 1]; j := j + 1 end;
      if k = 1 then M := M + 10 * Stevke[1] + Stevke[0]
      else M := M + 10 * Stevke[0] + Stevke[1];
      { Mogoče lahko do te vrednosti malega števca pridemo tako, da ga med
        vožnjo v primernem trenutku resetiramo. Vožnja mora biti dovolj dolga,
        da bo imel po resetiranju čas priti od 0000 do vrednosti Mali. }
      if M div 10 <= KmDelta then
        begin Nasel := true; break end;
      { Mogoče pa bo dosegel mali števec to vrednost v tem kilometru že kar
        brez našega posredovanja. V trenutnem kilometru bo imel vrednosti
        od Mali do Mali + 9. }
      if (Mali <= M) and (M <= Mali + 9) then
        begin Nasel := true; break end;
    end; {for i}
    if Nasel then break;
    { Prevozimo še en kilometer. }
    Km := (Km + 1) mod 1000000; Mali := (Mali + 10) mod 10000;
    KmDelta := KmDelta + 1;
  end; {while}
  for i := 0 to 5 do begin Stevke[i] := Km mod 10; Km := Km div 10 end;
  Assign(F, 'output.txt'); Rewrite(F);
  for i := 5 downto 0 do Write(F, Stevke[i]);
  WriteLn(F); Close(F);
end. {Stevec}
```

Program se bo zagotovo vedno ustavil — primerne vrednosti velikega števca bomo vedno znova srečevali (ne more nam jih zmanjkati), prej ali slej pa bo naša vožnja postala tudi dovolj dolga (KmDelta dovolj velika), da bi lahko z resetiranjem malega števca v primernem trenutku poskrbeli, da bo imel ta na

koncu ravno želeno vrednost. Če rešimo nalogo za vseh $10^{10}$ možnih začetnih stanj obeh števcev,[10] se izkaže, da je treba prevoziti povprečno po 948,27 km, vendar je dolžina vožnje močno odvisna od začetnega stanja: standardna deviacija je 2694,68 km, najdaljša potrebna vožnja pa je dolga kar 24702 km (do tega pride v primerih, ko je veliki števec na začetku enak 987643 in moramo voziti tako dolgo, dokler ne pride v stanje 012345).

[10]To ne pomeni, da moramo zgornji program pognati $10^{10}$-krat — z nekaj majhnimi spremembami nam lahko izračuna rešitve za vseh 10000 možnih začetnih stanj malega števca pri nekem danem začetnem stanju velikega števca; pri tem pa ne porabi 10000-krat toliko časa kot zgornji program za eno samo začetno stanje malega števca, pač pa le nekajkrat toliko (pri naših poskusih približno štirikrat toliko). Tako lahko do rešitev za vseh $10^{10}$ začetnih stanj obeh števcev pridemo v nekaj minutah.