19. državno tekmovanje v znanju računalništva (1995)

NALOGE ZA PRVO SKUPINO

1995.1.1

Tekmovalna komisija je v naglici napisala spodnji program. **Kaj izpiše program?** Odgovor primerno utemelji!

Rešitev: str. 10

```
program Enigma(Output);
type
  z = packed array [1..255] of char;
var
  p, a, r: z;
  LP, LA, LR: integer;
  i, j, rks, k: integer;
begin
  p := 'ba'; LP := 2; { dolžina p }
  r := 'baba'; LR := 4; { dolžina r }
  a := 'alibaba in štirideset barbarov.'; LA := 31; { dolžina a }
  { v preostanku tabel r, p in a se nahajajo presledki }
  j := 0; rks := 1;
  while rks <> 0 do begin
    i := j + 1; k := 1;
    repeat
      if a[i] = p[k] then begin
        i := i + 1; k := k + 1;
      end else begin
        i := i − k + 2; k := 1;
      end;
    until (k > LP) or (i > LA);
    if k > LP then begin
      i := i − LP; rks := i; j := i;
      if LP < LR then begin
        for k := LA + 100 downto i + LP do a[k] := a[k − (LR − LP)];
        for k := 1 to LR do a[k + i − 1] := r[k];
      end else begin
        for k := i to LA − (LP − LR) do a[k] := a[k + (LP − LR)];
        for k := 1 to LR do a[k + i − 1] := r[k];
      end;
    end else begin
      rks := 0; j := 0;
    end; {if}
  end; {while}
  WriteLn(a:LA);
end. {Enigma}
```

Opomba: pri delu z nizi se različna narečja pascala med seboj razlikujejo v raznih podrobnostih, zato bi zgornji program pri nekaterih prevajalnikih deloval drugače, kot je bil zamišljen. Prireditev, kot je `p := 'ba'`, v standardnem pascalu pravzaprav sploh ni dovoljena, ker niza nimata enakega števila elementov (`p` ima 255 elementov, konstanto `'ba'` pa jezik šteje za tabelo z dvema elementoma). Mnogi prevajalniki pa bi takšno prireditev vendarle dovolili in bi preostanek tabele `p` zapolnili s presledki (znaki `' '`); na takšno delovanje se zanaša tudi naš zgornji program.

Klic oblike WriteLn(a:n) pa naj bi, če je a tipa **packed array** [1..m] **of** char in je m > n, izpisal le prvih n znakov tabele a, ne glede na to, kaj je v preostanku te tabele. Takšno obnašanje predpisuje standardni pascal in ga uporablja tudi zgornji program.[1]

Rešitev: str. 12

1995.1.2

Podano imamo množico delavcev, ki jih označimo s celimi števili od 1 do n. Vsak delavec ima lahko enega ali več neposrednih šefov. Dana je tudi funkcija Sef(x, y: integer): boolean, ki vrne true, če je x neposredni šef delavcu y. **Napiši del programa** (ali podprogram), ki za danega delavca (njegovo številko preberemo) izpiše vse njegove šefe (posredne in neposredne).

Primer: v spodnjem primeru sta Tonetova neposredna šefa Janez in Gregor, njegov posredni šef pa je Dare.

Delavec	*Šef*
Tone	Janez
Tone	Gregor
Marjan	Gregor
Marjan	Janez
Janez	Dare
Gregor	Dare
Tadej	Dare

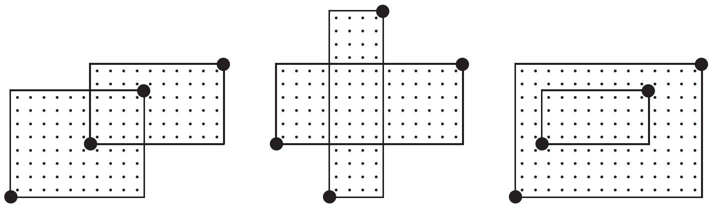
Rešitev: str. 12

1995.1.3

Napiši program, ki bo izračunal površino unije dveh pravokotnikov v ravnini. Stranice pravokotnikov so vzporedne s koordinatnima osema.

Pravokotniki so podani s koordinatama spodnjega levega in zgornjega desnega oglišča pravokotnika. Točke so podane kot pari realnih števil (koordinati X in Y), ki jih program prebere.

Nekaj primerov: Površina unije je šrafirana s pikicami. Podani točki za posamezni pravokotnik sta označeni s črno piko.



Rešitev: str. 14

1995.1.4

Računalnik je sestavljen iz enega glavnega in petdesetih med seboj enakih pomožnih procesorjev. V nevihti je strela udarila v omrežje prav blizu računalnika in odpovedalo je nekaj pomožnih procesorjev. Kateri?

Napisati je potrebno **program**, s katerim bo glavni procesor ugotovil, kateri pomožni procesorji so pokvarjeni. Na voljo je funkcija Enaka(a, b: integer): boolean, ki na pomožnih procesorjih s številkama a in b izvede določene operacije in vrne true, če so dale na obeh procesorjih enak rezultat in false, če ne.

[1] Nekateri nestandardni prevajalniki pa bi v takem primeru izpisali kar vse znake tabele a, ne glede na predpisano širino polja n. Nekateri prevajalniki z izpisovanjem znakov prenehajo že tudi pred n-tim znakom, če naletijo v nizu na znak Chr(0).

Predpostavimo lahko, da je pokvarjenih manj kot pol procesorjev in da je funkcija Enaka dovolj natančna, da ne more po naključju spregledati pokvarjenega procesorja (torej, če primerjamo pokvarjenega in pravilno delujočega, gotovo vrne false, če primerjamo dva pravilno delujoča, pa gotovo vrne true). Žal pa ni nujno, da so vsi pokvarjeni procesorji pokvarjeni na enak način; napake so lahko različne ali enake, torej funkcija Enaka pri primerjanju dveh pokvarjenih procesorjev lahko vrne true ali false.

NALOGE ZA DRUGO SKUPINO

1995.2.1

Kaj vrne funkcija KajVrnem v odvisnosti od parametra x? Parameter t je tabela naključno izbranih števil. x je enak nekemu elementu tabele t. Odgovor utemelji!

Rešitev: str. 15

```
const Dolz = 10000;
type Tabela = array [1..Dolz] of integer;

function KajVrnem(var t: Tabela; x: integer): integer;
var
  a, b, c: integer;
begin
  a := 1; b := Dolz;
  while a < b do begin
    while t[a] < x do a := a + 1;
    while t[b] > x do b := b − 1;
    if a < b then begin
      c := t[a]; t[a] := t[b]; t[b] := c;
    end; {if}
  end; {while}
  KajVrnem := a;
end; {KajVrnem}
```

1995.2.2

Računalniki so iz dneva v dan hitrejši in imajo več pomnilnika. Pri resnem delu pa je ozko grlo še vedno branje in pisanje na trdi disk, ki je približno 100000-krat počasnejše kot branje in pisanje v pomnilnik. Zato skoraj vedno žrtvujemo del pomnilnika za t.i. priročni pomnilnik (angl. cache), ki ima del podatkov iz diska vedno shranjen v pomnilniku in če uporabnik zahteva kakšnega od njih, ni potrebno dosegati podatkov na disku. To seveda zelo pohitri delo na računalniku.

Rešitev: str. 15

Kako vse skupaj deluje? Disk je razdeljen na manjše enote, ki jih imenujmo bloki in tipično vsebujejo 512 (ali več) znakov. Do operacijskega sistema prihajajo zahtevki za branje in pisanje blokov na disk. Ker se v praksi pokaže, da v nekem obdobju uporabnik pogosto prebere isti blok večkrat in čez isti blok večkrat zapiše različno vsebino, si pomagamo s priročnim pomnilnikom, kamor shranjujemo uporabnikove zahtevke in podatke. Če hoče uporabnik zapisati blok podatkov na disk, tega ne zapišemo zares, ampak ga shranimo v priročni pomnilnik. Tam je, dokler nam ne zmanjka prostora v priročnem pomnilniku in se izkaže, da je bil ta blok največ časa neuporabljen — takrat ga dejansko zapišemo na disk. Če v času, ko se blok nahaja v pomnilniku, pride še en zahtevek po tem, da ga prepišemo, ga preprosto prepišemo le v pomnilniku. Če pride zahtevek po branju bloka, ki se ne nahaja v priročnem pomnilniku, ga moramo najprej prebrati z diska, nato pa ga vrnemo uporabniku in ga še shranimo v priročni pomnilnik. V primeru, ko uporabnik zahteva blok, ki je že v priročnem pomnilniku, mu vrnemo kar tega.

V rešitvi **opiši postopek**, ki ga morata izvajati podprograma za pisanje in branje z diska,

```
procedure ZapisiBlok(StBloka: integer; P: Podatki);
procedure PreberiBlok(StBloka: integer; var P: Podatki);
```

ki pa imata za pomoč priročni pomnilnik, v katerega lahko shraniš do MaksPPBlokov blokov. Opiši tudi podatkovne strukture, ki omogočajo čim hitrejše delovanje obeh podprogramov. Ko je podatek zares potrebno fizično zapisati oz. prebrati z diska, uporabi podprograma:

```
procedure ZapisiBlokNaDisk(StBloka: integer; P: Podatki); external;
procedure PreberiBlokZDiska(StBloka: integer; var P: Podatki); external;
```

Predpostavi, da ima vsak blok na disku svojo številko (celo število med 1 in MaksBlokov) in tabelo znakov, ki predstavljajo podatke:

```
type Znak = 0..255;
type Podatki = array [1..512] of Znak;
```

Rešitev: str. 17

1995.2.3

Prek računalniškega omrežja želimo razširjati zaporedje znakov, npr. besedila dokumentov, ki stalno, a neenakomerno sproti nastajajo. Naš računalnik sprejema znake po eni vhodni zvezi, posredovati pa jih mora sproti, ne da bi najprej čakal na konec dokumenta, naprej drugim računalnikom, ki jih je nekaj ducatov (neka konstanta), do vsakega od njih pa vodi ena izhodna zveza.

Na voljo imamo dva podprograma:

function Beri(**var** ch: char): boolean;

- če je na vhodni liniji na voljo kakšen nov znak, ga prebere in vrne kot parameter, vrednost funkcije je true;
- če na vhodu ni nobenega znaka, je vrednost funkcije false, ch pa ni definiran.

Funkcija se vedno izvede takoj (ne čaka na znake). Če funkcije nekaj časa ne pokličemo, se tok prihajajočih znakov začasno ustavi (znaki se ne bodo izgubili).

function Pisi(iz: integer; ch: char): boolean;

Na izhodno linijo iz (številka med 1 in številom izhodnih linij) poskusi poslati znak ch. Če je bilo pošiljanje uspešno, vrne true, sicer pa false. Funkcija se vedno izvede takoj (npr. ne čaka, da se linija sprosti).

Zveze niso enako hitre, vendar tudi najpočasnejša večino časa dohaja dotok novih besedil — če pa ga kdaj ne dohaja, se znaki ne smejo izgubljati, prenos se lahko le upočasni. Da prenosi po hitrejših linijah ne bi trpeli na račun počasnejših (razen izjemoma, npr. pri prenašanju zelo dolgih besedil), uporabimo v programu večji kos pomnilnika (vrsto/buffer) za začasno hranjenje še neodposlanih znakov. Na razpolago je le toliko pomnilnika, da gre vanj večina dokumentov v celoti, ne pa najdaljši.

Napiši program, ki bo skrbel za razširjanje besedil.

1995.2.4 V mreži računalnikov se nahaja sto vozlišč (računalnikov), ki si med seboj lahko pošiljajo sporočila. Posamezno vozlišče je lahko povezano z največ petimi sosednjimi vozlišči. Vsak računalnik ima svoj enoten mrežni naslov, ki je neko pozitivno celo število, vendar pa ne ve, s katerimi sosednjimi računalniki in po katerih povezavah je povezan.

Rešitev: str. 18

Da bi prenos sporočil potekal kar najhitreje, mora vsak računalnik poznati najkrajše poti do ostalih računalnikov v mreži. Za to naj poskrbi **program**, ki se bo hkrati pognal in izvajal na vseh računalnikih. Njegova naloga je izvedeti, v katero izmed petih smeri naj posamezen računalnik pošlje sporočilo, če ga želi poslati nekemu drugemu računalniku. Program naj si zgradi tabelo najkrajših smeri za vseh sto računalnikov v mreži.

Pri pisanju programa si pomagajte z naslednjimi podprogrami:

function MojNaslov: integer;
: Vrne naslov računalnika, ki je število od 1 do 100.

function Poslji(VSmer: integer; Sporocilo): boolean;
: Pošlje sporočilo v želeno smer. Sporočilo definirajte sami. Če računalnik ne more poslati sporočila, ker v dani smeri ni povezave, funkcija vrne false, sicer pa true. Upoštevajte, da se sporočila nikoli ne izgubljajo.

function Sprejmi(**var** IzSmeri: integer; **var** Sporočilo): boolean;
: V primeru, da je prispelo novo sporočilo, ga zapiše v parametra IzSmeri in Sporočilo ter vrne true. Če sporočila ni, vrne false.

Pri tem upoštevajte, da smeri pomenijo številke povezav med računalniki; to so števila od 1 do 5.

NALOGE ZA TRETJO SKUPINO

1995.3.1 Preveč samozavesten programer je napisal naslednji program. **Kaj ta program izpiše?** Odgovor primerno utemelji! Kdaj izbrani algoritem ne bi deloval pravilno?

Rešitev: str. 19

```
program Enigma(Output);
type z = packed array [1..255] of char;
var
  p, a, r: z;
  j: integer;

function Length(s: z):integer;
var i: integer;
begin
  i := 255; while (i > 1) and (s[i] = ' ') do i := i − 1;
  if (i = 1) and (s[1] = ' ') then Length:= 0 else Length:= i;
end;

procedure Replace(LP, LR, LA: integer; var a, r: z);
var k: integer;
begin
  if LP < LR then begin
    for k := LA + 100 downto j + LP do a[k] := a[k − (LR − LP)];
```

```
      for k := 1 to LR do a[k + j − 1] := r[k];
    end else begin
      for k := j to LA − (LP − LR) do a[k] := a[k + (LP − LR)];
      for k := 1 to LR do a[k + j − 1] := r[k];
    end;
end;

function RKSearch(var a, p, r: z):integer;
const
  q = 33554393; { praštevilo }
  d = 32;
var
  h1, h2, dM, i, LP, LA, LR: integer;
begin
  LP := Length(p); LA := Length(a); LR := Length(r);
  dM := 1; for i := 1 to LP − 1 do dM := (d * dM) mod q;
  h1 := 0; for i := 1 to LP do h1 := (h1 * d + Ord(p[i])) mod q;
  h2 := 0; for i := 1 to LP do h2 := (h2 * d + Ord(a[i + j])) mod q;
  i := j + 1;
  while (h1 <> h2) and (i <= LA − LP) do begin
    h2 := (h2 + d * q − Ord(a[i]) * dM) mod q;
    h2 := (h2 * d + Ord(a[i + LP])) mod q;
    i := i + 1;
  end; {while}
  if h1 = h2 then begin
    RKSearch := i; j := i; Replace(LP, LR, LA, a, r);
  end else begin
    RKSearch := 0; j := 0;
  end; {if}
end;

begin
  r := 'baba'; p := 'ba'; a := 'alibaba in štirideset barbarov.';
  { v preostanku tabel r, p in a se nahajajo presledki }
  j := 0;
  while RKSearch(a, p, r) <> 0 do begin
    WriteLn(a:Length(a)); WriteLn('^':j);
    j := j + Length(r) − Length(p);
  end;
end.
```

Opomba: pri delu z nizi se različna narečja pascala med seboj razlikujejo v nekaterih podrobnostih; glej opombo pri prvi nalogi za prvo skupino, str. 1.

Rešitev: str. 21

1995.3.2

Na zaslonu je niz N sedemsegmentnih (digitalnih) števk. Predstavitev števk s segmenti je naslednja:

0 1 2 3 4 5 6 7 8 9

Za prižgane segmente vemo, da so pravilni, medtem ko za ugasnjene segmente ne vemo, ali so pravilni ali pokvarjeni. Poleg tega vemo, da je vsota vseh števk deljiva z 10.

Opiši postopek, ki najde tak niz vrednosti števk, da ustreza omejitvam in ima najmanj popravljenih segmentov.

Primer:  ima možni rešitvi 3-7 in 8-2. Rešitev je 3-7, ker ima samo en popravljen segment, medtem ko ima 8-2 pet popravljenih segmentov.

1995.3.3

Rešitev: str. 28

Podano imamo množico delavcev, ki jih označimo s celimi števili od 1 do n. Vsak delavec ima lahko enega ali več neposrednih šefov. Dana je tudi funkcija **function** JeSef(x, y: integer): boolean, ki vrne true, če je x šef delavcu y (neposredni ali posredni). **Napiši algoritem**, ki za dano množico delavcev izračuna množico vseh njihovih najbližjih skupnih šefov.

Najbližji skupni šef množice delavcev S je delavec:

1. ki je šef vsakemu delavcu iz množice S in
2. ne obstaja njemu podrejeni delavec, ki bi bil prav tako šef vsem delavcem iz množice S.

Primer: V primeru, da je relacija **function** JeSef(x, y: integer): boolean definirana z naslednjo tabelo, sta Gregor in Janez najbližja skupna šefa Tonetu in Marjanu.

Delavec	*Šef*
Tone	Janez
Tone	Gregor
Marjan	Gregor
Marjan	Janez
Janez	Dare
Gregor	Dare
Tadej	Dare

1995.3.4

Rešitev: str. 30

V mreži računalnikov se nahaja poljubno število vozlišč (računalnikov), ki si med seboj lahko pošiljajo sporočila. Posamezno vozlišče je lahko povezano z največ enajstimi sosednjimi vozlišči. Računalniki ob vključitvi nimajo informacije o tem, po katerih povezavah in s katerimi sosednjimi računalniki so povezani, niti o tem, kje se nahajajo. Zato želimo vse računalnike, ki so priključeni v mrežo, označiti z enotnimi identifikacijskimi številkami, ki nam bodo kasneje služile kot mrežni naslovi. To naj naredi program, ki se zažene vzporedno na vseh računalnikih ob vzpostavitvi mreže.

Napišite omenjeni **program** in pri tem uporabite naslednji funkciji in podprogram:

function Poslji(VSmer: integer; Sporocilo): boolean; **external**;
: Pošlje sporočilo v želeno smer. Sporočilo definirajte sami. Če računalnik ne more poslati sporočila, ker v dani smeri ni povezave, funkcija vrne false, sicer pa true. Upoštevajte, da se sporočila nikoli ne izgubljajo.

function Sprejmi(**var** IzSmeri: integer; **var** Sporocilo): boolean; **external**;
: Če je prispelo novo sporočilo, ga zapiše v spremenljivki IzSmeri in Sporocilo ter vrne true. Če pa sporočila ni bilo, vrne false.

procedure NastaviSvojNaslov(Naslov: integer); **external**;
: Ko računalnik dožene, kakšen naslov mu pripada, s tem podprogramom nastavi svoj mrežni naslov.

Pri tem upoštevajte, da smeri pomenijo številke povezav med računalniki, ki so lahko od 1 do 11. Izjema je le prvo sporočilo, ki do naključno izbranega računalnika pride od uporabnika (iz smeri 0) in pomeni ukaz za začetek označevanja. Program napišite tako, da, ko je označevanje na vseh računalnikih končano, eden izmed računalnikov to sporoči uporabniku (pošlje naj kakršnokoli sporočilo v smer 0).

PRVO ZAKLJUČNO TEKMOVANJE V ZNANJU RAČUNALNIŠTVA

Rešitev: str. 31

1995.Z Dani sta datoteki `vzorci.txt` in `besede.txt`. V vsaki vrstici obeh datotek se nahaja po ena beseda v velikih črkah. Dolžina besede je od 1 do 12 znakov.

Napiši program, ki za vsako besedo iz datoteke `vzorci.txt` poišče vse bližnje besede iz datoteke `besede.txt` in jih zapiše v datoteko `rezultat.txt`. Beseda B (iz `besede.txt`) sodi med bližnje besede besedi V (iz `vzorci.txt`), če B vsebuje vse znake iz besede V, vsebuje pa lahko še 2 dodatna znaka.

Primera: Besedi UNIVERZA (iz `vzorci.txt`) so bližnje besede RAZVIDENJU, REVANŠIZMU, UNIVERZA, UNIVERZAH, UNIVERZUMA (iz `besede.txt`). Besedi MOJSTRANA je bližnja beseda ASTRONOMIJA.

Naloga je napisati program, ki bo čim hitreje iskal bližnje besede. Čas reševanja je 3 ure. V tem času si lahko na področju (direktoriju), ki vam je določen za delo, poleg programov zgradite tudi indekse in druge pomožne datoteke, ki vam pomagajo pri iskanju bližnjih besed. Pri tem lahko vse datoteke na področju zapolnijo največ 5 MB prostora. Po končanem reševanju nalog bomo vsebino področij vseh tekmovalcev prenesli na skupen računalnik, kjer bomo izmerili hitrost izvajanja programov na spremenjeni datoteki `vzorci.txt`, ki bo vsebovala nekaj deset besed v enakem formatu kot testna datoteka. Zmagal bo tekmovalec, katerega program bo najhitreje poiskal vse bližnje besede besedam iz nove datoteke `vzorci.txt`.

Čas bomo merili s programom `mericas.pas`, ki ga poženete iz DOSa s parametrom, ki je ime vašega programa skupaj s podaljškom `exe`. Primer: Če je datoteki s programom, ki ste ga napisali, ime `isci.pas`, prevedeni pa `isci.exe`, izmerite čas izvajanja iz DOSa z ukazom:

```
c:\>mericas isci.exe
```

Naša inačica programa bo poleg merjenja časa še preverjala, ali vaš program izpiše vse možne besede.

V pomoč vam je lahko naivna izvedba programa za iskanje bližnjih besed, ki se nahaja v datoteki `naivna.pas`. Program, ki ga morate napisati, se mora obnašati enako kot spodnji primer, le vrstni red izpisanih besed je lahko drugačen. Več možnosti za zmago pa boste seveda imeli, če bo vaš program deloval hitreje.

```
program NaivnaResitev;
var
  fVzorci, fBesede, fRezultat: text;
  CelVzorec, Vzorec, CelaBeseda, Beseda: string;
  StZnaka, NajdenZnak: integer;
begin
```

```
  Assign(fRezultat, 'rezultat.txt'); Rewrite(fRezultat);
  Assign(fVzorci, 'vzorci.txt'); Reset(fVzorci);
  while not Eof(fVzorci) do begin
    ReadLn(fVzorci, CelVzorec );
    Assign(fBesede, 'besede.txt'); Reset(fBesede);
    while not Eof(fBesede) do begin
      ReadLn(fBesede, CelaBeseda);
      Vzorec := CelVzorec; Beseda := CelaBeseda;
      for StZnaka := 1 to Length(CelVzorec) do begin
        NajdenZnak := Pos(CelVzorec[StZnaka], Beseda);
        if NajdenZnak <> 0 then begin
          Delete(Beseda, NajdenZnak, 1);
          Delete(Vzorec, Pos(CelVzorec[StZnaka], Vzorec), 1);
        end; {if}
      end; {for}
      if (Length(Vzorec) <= 0) and (Length(Beseda) <= 2) then
        WriteLn(fRezultat, CelaBeseda);
    end; {while};
    Close(fBesede);
  end; {while};
  Close(fVzorci); Close(fRezultat);
end. {NaivnaResitev}
```

Pri nalogah, ki se jih rešuje na računalnikih, vedno obstaja nerodnost, da s prihodom hitrejših procesorjev in večjih količin pomnilnika včasih odpadejo omejitve, zaradi katerih je bila naloga prej težka; marsikatera naloga postane lažja, včasih celo zelo lahka. Mogoče najpomembnejši tovrstni omejitvi na finalnem tekmovanju leta 1995 sta bili majhna količina pomnilnika (programi so tekli v realnem načinu, kar pomeni, da so imeli za svoje delo na voljo le nekaj sto KB pomnilnika, vsekakor pa ne več kot 640 KB) in pa dejstvo, da niti disk niti operacijski sistem najbrž nista imela kakšne resne količine diskovnega predpomnilnika.

Datoteko `besede.txt` in druge datoteke, povezane s tem tekmovanjem, naj bi se dalo dobiti na `http://www.brank.org/rtk/`. Če bralec te datoteke nima pri roki, mu bo pri načrtovanju rešitve mogoče prišel prav vsaj podatek, da je bilo v datoteki `besede.txt` 112539 besed s skupno 883193 črkami (število besed po dolžini od eno- do dvanajstčrkovnih: 29, 387, 1612, 4659, 10202, 14896, 18296, 18848, 16286, 12971, 8844, 5509). Vse besede so sestavljene iz velikih črk `A`, …, `Z` (tudi `Q`, `W`, `X` in `Y`), `Č`, `Š` in `Ž`.

To tekmovanje je potekalo dan po običajnem tekmovanju, torej v nedeljo, 14. maja 1995. Udeležilo se ga je 18 tekmovalcev (najboljši iz 2. in 3. skupine ter nekaj mladih raziskovalcev). Za osem izmed njih so organizirali še drugo tekmovanje (v soboto, 27. maja 1995); naloga je bila enaka kot pri prvem, le da vzorci niso bili podani v datoteki, ampak je moral program rešiti problem za en sam vzorec, ki ga je dobil v ukazni vrstici (argv v C/C++, ParamStr v Turbo Pascalu). Pri tem drugem tekmovanju ni šlo za klavzurno reševanje naloge, ampak je nalogo vsak reševal doma v dveh tednih med obema tekmovanjema, sámo drugo tekmovanje pa je bilo namenjeno zgolj preizkušanju in merjenju hitrosti nastalih programov. V nadaljevanju sledi opis naloge za drugo tekmovanje.

Dana je datoteka besed `besede.txt`, ki v vsaki vrstici vsebuje eno besedo dolžine od 1 do 12 znakov. **Napiši program**, ki za posamezno besedo (podano kot podatek iz ukazne vrstice) poišče vse bližnje besede iz datoteke `besede.txt` in jih zapiše v datoteko `rezultat.txt`. Beseda B (iz `besede.txt`) sodi med bližnje besede besedi V (podani iz ukazne vrstice), če B vsebuje vse znake iz besede V, vsebuje pa lahko še 2 dodatna znaka.

Primera: Besedi `UNIVERZA` so bližnje besede `RAZVIDENJU`, `REVANŠIZMU`, `UNIVERZA`, `UNIVERZAH`, `UNIVERZUMA` (iz `besede.txt`). Besedi `MOJSTRANA` je bližnja beseda `ASTRONOMIJA`.

Naloga je napisati program (z imenom `najdi`), ki bo čim hitreje iskal bližnje besede. Dovoljeno je, da si na delovnem področju (direktoriju) poleg programov zgradite tudi indekse in druge pomožne datoteke, ki vam pomagajo pri iskanju bližnjih besed. Pri tem lahko vse datoteke na področju zapolnijo največ 5 MB prostora. Na tekmovanju bomo vsebino področij vseh tekmovalcev prenesli na skupen računalnik, kjer bomo izmerili hitrost izvajanja programov na nekaj deset vzorčnih besedah. Zmagal bo tekmovalec, katerega program bo najhitreje poiskal vse bližnje besede podanim vzorčnim besedam. Program mora biti napisan v Turbo Pascalu 7.0 ali Borland C 3.1 za DOS, z uporabo osnovnega pomnilnika (do 640 KB).

Poleg programa morajo tekmovalci napisati dokumentacijo v angleškem jeziku dolžine med 1000 in 1500 besed. Dokumentacija mora vsebovati opis algoritma za iskanje bližnjih besed in opis programske izvedbe tega algoritma.

Na samem tekmovanju bodo tekmovalci zagovarjali svoj program v pet- do desetminutni predstavitvi v angleškem jeziku. Dovoljena je uporaba grafoskopa. Prosojnice smejo vsebovati le slike in diagrame, ne smejo pa vsebovati besedila, ki bi ga tekmovalec bral.

Čas bomo merili s programom `mericas.pas`, ki ga poženete iz DOSa s parametrom, ki je ime vašega programa skupaj s podaljškom `exe`. Primer: Če je datoteki s programom, ki ste ga napisali, ime `najdi.pas`, prevedeni pa `najdi.exe`, izmerite čas izvajanja iz DOSa z ukazom:

```
c:\>mericas najdi.exe
```

Program `mericas.pas` jemlje vzorčne besede eno po eno iz datoteke `vzorci.txt` in jih daje kot parameter klicu vašega programa. Naša inačica programa bo poleg merjenja časa še preverjala, ali vaš program izpiše vse možne besede.

V pomoč vam je lahko naivna izvedba programa za iskanje bližnjih besed, ki se nahaja v datoteki `naivna.pas`. Program, ki ga morate napisati, se mora obnašati enako kot spodnji primer, le vrstni red izpisanih besed je lahko drugačen. Več možnosti za zmago boste seveda imeli, če bo vaš program deloval hitreje.

```
program NaivnaResitev;
var
  fBesede, fRezultat: text;
  CelVzorec, Vzorec, CelaBeseda, Beseda: string;
  StZnaka, NajdenZnak: integer;
begin
  Assign(fRezultat, 'rezultat.txt'); Rewrite(fRezultat);
  CelVzorec := ParamStr(1);
  Assign(fBesede, 'besede.txt'); Reset(fBesede);
  while not Eof(fBesede) do begin
    ReadLn(fBesede, CelaBeseda);
    Vzorec := CelVzorec; Beseda := CelaBeseda;
    for StZnaka := 1 to Length(CelVzorec) do begin
      NajdenZnak := Pos(CelVzorec[StZnaka], Beseda);
      if NajdenZnak <> 0 then begin
        Delete(Beseda, NajdenZnak, 1);
        Delete(Vzorec, Pos(CelVzorec[StZnaka], Vzorec), 1);
      end; {if}
    end; {for}
    if (Length(Vzorec) <= 0) and (Length(Beseda) <= 2) then begin
      WriteLn(fRezultat, CelaBeseda); WriteLn(CelaBeseda);
    end; {if}
  end; {while};
  Close(fBesede);
  Close(fRezultat);
end. {NaivnaResitev}
```

REŠITVE NALOG ZA PRVO SKUPINO

Naloga: str. 1

R1995.1.1

Program poskuša zamenjati pojavitve niza `p` v nizu `a` z nizom `r`. V vsaki iteraciji zunanje zanke (zanka **while**) naj bi išče v nizu `a` od vključno indeksa $j + 1$ naprej (zato pri inicializaciji postavimo j na 0, da bomo iskali od začetka niza `a`).

Zanka **repeat** poišče naslednjo pojavitev podniza p v nizu a (torej prvo pojavitev od vključno indeksa j + 1 naprej). Dokler vidi ujemanje med nizoma a in p, se premika naprej po obeh (povečuje i in k), dokler ne pride do konca niza p (to je pri pogoju k > LP). Če pa se trenutna znaka a[i] in p[k] ne ujemata, poizkusi z naslednjim možnim položajem niza p in začne tam primerjati spet od začetka. Če smo nazadnje primerjali a[i] in p[k], pomeni, da smo začeli pri a[i − k + 1] in p[1]; ko niz p v mislih premaknemo za eno mesto naprej vzdolž niza a, bomo morali torej začeti primerjati pri a[i − k + 2] in p[1].

Notranja zanka **repeat** se konča, če pride k do konca niza p (kar pomeni, da smo našli pojavitev niza p v nizu a) ali pa i do konca niza a (ko je i > LA, kar pomeni, da smo v a-ju odkrili že vse pojavitve p-ja in zato nehamo: v ta namen postavimo rks na 0 (vse dotlej je imel neničelno vrednost), da se bo zunanja zanka prekinila).

Če smo našli pojavitev p-ja v a-ju (k > LP), jo je treba zamenjati z r-jem. Ko indeks i zmanjšamo za LP, kaže spet na začetek te pojavitve p-ja v a-ju. Zdaj je mogoče, da r ni tako dolg kot p in bo treba zato tisti preostanek a-ja, ki ne pripada trenutni pojavitvi p-ja, prestaviti.

Če je r daljši od p-ja, bo treba vsako celico a[k] prestaviti v a[k + (LR − LP)]; da pa pri tem ne bi povozili prejšnje vsebine te celice, bomo počeli to od konca niza proti začetku in k pri tem zmanjševali. Tu program napačno predpostavi, da je niz a dolg LA in za „boljše“ delovanje se izvaja znaka od indeksa LA + 100 navzdol (ob začetku izvajanja programa je dolŽina niza a res LA, toda med izvajanjem programa se lahko dolžina a-ja spreminja in program ne spreminja vrednosti spremenljivke LA). Varno (toda malo bolj potratno) napisana zanka bi prestavila vse znake od konca tabele, to je od celice 255 nazaj.

Če je r krajši od p-ja, ravnamo podobno, le da se tu vsebina a-ja seli na nižje indekse, zato moramo iti po naraščajočih k namesto po padajočih. Nato še prepišemo vsebino r-ja v tabelo a na indekse od i naprej.

Ob zamenjavi trenutne pojavitve p-ja z r-jem postavimo j na i (rks pa tudi, ampak glede tega je čisto vseeno, samo da bo rks neničeln, kar pa zagotovo je, ker je i zagotovo neničeln). Naslednja ponovitev zunanje zanke začne preverjati pojavitve p-ja na indeksih od i + 1 naprej, torej eno mesto za trenutno pojavitvijo.

Program vsebuje napako: ker se iskani niz p pojavlja kot podniz v zamenjavnem nizu r in ker iščemo naslednjo pojavitev p-ja na naslednjem mestu za zadnjo najdeno pojavitvijo p-ja (to je pri i + 1) namesto na koncu zamenjanega (vrinjenega) r-ja), bi zdaj znotraj tega vrinjenega r-ja našel novo pojavitev p-ja, zamenjal še to in tako naprej. Niz a bi se pri tem napihoval v nedogled, če ga ne bi rešila še druga napaka v programu: ko zamenja neko pojavitev p-ja z r-jem, se dolžina niza a poveča za LR − LP, program pa spremenljivke LA ne spreminja. Zato se ne meni za tiste pojavitve p-ja v a-ju, ki se ne pojavljajo v celoti znotraj prvih LA znakov. Ob koncu izvajanja programa vsebuje a niz

„`alibababababababababababababababa in štirideset barbarov.`“

(v preostanku tabele a pa so presledki). Ker program (neupravičeno) zaupa vrednosti LA in izpiše le prvih LA znakov niza a, bomo dobili izpis

„`alibababababababababababababa`“.

Naloga: str. 2

R1995.1.2

V spodnjem programu je rešitev naloge pravzaprav le podprogram Nadrejeni, ostalo pa so pomožne deklaracije oz. stvari, za katere je bilo v nalogi rečeno, da lahko predpostavimo, da že obstajajo (npr. funkcija Sef).

Podprogram IzpisiNadrejene hrani v tabeli Nadrejeni podatke o tem, za katere delavce že ve, da so x-ovi šefi. Na začetku označi, da je x šef samemu sebi, nato pa za vsakega že znanega šefa doda v tabelo vse njegove šefe. Če pridemo enkrat skozi celo tabelo Nadrejeni, ne da bi odkrili še kakšnega novega šefa, smo našli vse in lahko končamo (spremenljivka Konec).[2]

Spodnja rešitev je do neke mere neučinkovita, ker vedno znova prečesava celotno tabelo Nadrejeni, čeprav bi bilo v resnici dovolj že, če bi preverila le tiste šefe, ki jih je dodala v prejšnjem prehodu čez to tabelo — lahko bi torej na novo odkrite šefe dodajala v neko pomožno tabelo, ki bi jo potem uporabljala kot vrsto za šefe, ki smo jih že odkrili, a si jih moramo še podrobneje ogledati.

```
const n = ......; { Število oseb. }
type Oseba = 1..n; { Številka osebe. }
function Sef(s, d: Oseba): boolean; external;

procedure IzpisiNadrejene(x: Oseba); { Izpiše vse ljudi, nadrejene delavcu x. }
var Konec: boolean;
   i, j: Oseba;
   Nadrejeni: array [Oseba] of boolean;
begin
  { Inicializacija tabele Nadrejeni. }
  for i := 1 to n do Nadrejeni[i] := false;
  Nadrejeni[x] := true;
  { Poišči vse nadrejene šefe. }
  repeat
    Konec := true;
    for i := 1 to n do if Nadrejeni[i] then
      for j := 1 to n do
        if Sef(j, i) and not Nadrejeni[j] then
          begin Konec := false; Nadrejeni[j] := true end
  until Konec;
  Nadrejeni[x] := false; { x v resnici ni sam svoj šef. }

  { Izpišimo nadrejene. }
  Write('Nadrejeni(', x, '):');
  for i := 1 to n do if Nadrejeni[i] then Write(' ', i);
  WriteLn;
end; {IzpisiNadrejene}
```

Naloga: str. 2

R1995.1.3

Lažje kot unijo dveh pravokotnikov (katerih stranice so vzporedne koordinatnima osema) je določiti njun presek: presek pravokotnikov $\langle(x_a, y_a), (x_b, y_b)\rangle$ in $\langle(x'_a, y'_a), (x'_b, y'_b)\rangle$ je pravokotnik

$$\langle(\max\{x_a, x'_a\}, \max\{y_a, y'_a\}), (\min\{x_b, x'_b\}, \min\{y_b, y'_b\})\rangle.$$

[2] Podprogram IzpisiNadrejene na koncu postavi Nadrejeni[x] na false, ker si pač mislimo, da x ne more biti samemu sebi šef. Vendar bi bilo konec koncev mogoče tudi to — klic Sef(x, x) bi lahko vrnil true ali pa bi šlo vodstvo v ciklih (npr. x in nek y bi bila lahko šefa drug drugemu). V tem primeru bi smeli na koncu postaviti Nadrejeni[x] na false le, če ni x neposredni šef nobenemu od odkritih šefov (niti samemu sebi).

Pri tem prvi par koordinat predstavlja spodnje levo, drugi pa zgornje desno oglišče pravokotnika. Paziti moramo še na možnost, da je presek prazen; to se zgodi, če je $\max\{x_a, x'_a\} \geq \min\{x_b, x'_b\}$ ali $\max\{y_a, y'_a\} \geq \min\{y_b, y'_b\}$.

Ploščino unije pravokotnikov lahko zdaj dobimo tako, da seštejemo ploščini obeh pravokotnikov in od vsote odštejemo ploščino preseka (saj smo ga v vsoti šteli dvakrat).

```
program UnijaPravokotnikov(Input, Output);

type Tocka = record x, y: real end;
     Pravokotnik = record a, b: Tocka end;

function Max(a, b: real): real;
  begin if a > b then Max := a else Max := b end;

function Min(a, b: real): real;
  begin if a < b then Min := a else Min := b end;

procedure Zamenjaj(var a, b: real);
var tmp: real;
begin
  tmp := a; a := b; b := tmp;
end; {Zamenjaj}

procedure Uredi(var a, b: Tocka);
begin
  if a.x > b.x then Zamenjaj(a.x, b.x);
  if a.y > b.y then Zamenjaj(a.y, b.y);
end; {Uredi}

function Presek(var pa, pb: Pravokotnik): real;
var PresekP: Pravokotnik;
begin
  Uredi(pa.a, pa.b);
  Uredi(pb.a, pb.b);
  PresekP.a.x := Max(pa.a.x, pb.a.x);
  PresekP.a.y := Max(pa.a.y, pb.a.y);
  PresekP.b.x := Min(pa.b.x, pb.b.x);
  PresekP.b.y := Min(pa.b.y, pb.b.y);
  Presek := Max(PresekP.b.x − PresekP.a.x, 0) *
            Max(PresekP.b.y − PresekP.a.y, 0);
end; {Presek}

function Unija(var pa, pb: Pravokotnik): real;
begin
  Unija :=   (pa.b.x − pa.a.x) * (pa.b.y − pa.a.y)
           + (pb.b.x − pb.a.x) * (pb.b.y − pb.a.y)
           − Presek(pa, pb);
end; {Unija}

var a, b: Pravokotnik;
begin {UnijaPravokotnikov}
  Write('1. pravokotnik, točka A: '); ReadLn(a.a.x, a.a.y);
  Write('1. pravokotnik, točka B: '); ReadLn(a.b.x, a.b.y);
  Write('2. pravokotnik, točka A: '); ReadLn(b.a.x, b.a.y);
  Write('2. pravokotnik, točka B: '); ReadLn(b.b.x, b.b.y);
```

```
  WriteLn('Presek = ', Presek(a, b):0:3);
  WriteLn('Unija  = ', Unija(a, b):0:3);
end. {UnijaPravokotnikov}
```

Naloga: str. 2

R1995.1.4

Preverimo, če prvi pomožni procesor deluje — to storimo tako, da ga primerjamo z vsemi ostalimi procesorji in če večina pravi, da je pokvarjen (različen od njih), potem je pač res pokvarjen. Vemo namreč, da je večina procesorjev še vedno pri pameti in torej zmožna pravilno ocenjevati druge procesorje. Nato preverimo drugi procesor, pa tretjega in tako naprej. Ko naletimo na delujoč procesor, zaključimo z iskanjem. Nato delujoči procesor primerjamo z vsemi pomožnimi procesorji, da vidimo, kateri so pokvarjeni.

```
program Preverjaj;
const N = 50;
var Pokvarjen: array [1..N] of boolean;
    i, j, Delujocih: integer;
begin
  j := 0;
  repeat
    j := j + 1;
    { Na tem mestu že vemo, da so vsi procesorji od 1 do j − 1 pokvarjeni
      (sicer bi se tale zanka že končala). O procesorju j zaenkrat predpostavimo,
      da je tudi pokvarjen. Spremenljivka Delujocih šteje v resnici to, koliko
      procesorjev (ne vštevši j-ja samega) meni, da j deluje pravilno. }
    Pokvarjen[j] := true; Delujocih := 0;
    { Poglejmo zdaj, kaj pravijo o j-ju ostali procesorji.
      Tistih od 1 do j − 1 nima smisla spraševati, ker vemo, da so pokvarjeni. }
    for i := j + 1 to N do
      if Enaka(j, i) then Delujocih := Delujocih + 1;
    { Vemo, da je pokvarjenih manj kot pol procesorjev. Ločimo naslednje možnosti:
      N = 2K + 1 (lih), torej vsaj K + 1 dobrih, največ K slabih.
        j pokvarjen → vsaj K + 1 jih bo reklo, da je zanič;
                      torej jih bo največ K − 1 reklo, da je dober.
        j dober     → vsaj K jih bo reklo, da je dober.
      N = 2K (sod), torej vsaj K + 1 dobrih, največ K − 1 slabih.
        j pokvarjen → vsaj K + 1 jih bo reklo, da je zanič;
                      torej jih bo največ K − 2 bo reklo, da je dober.
        j dober     → vsaj K jih bo reklo, da je dober.
      Torej je j dober natanko tedaj, ko je vsaj K = N div 2 drugih procesorjev reklo, }
  until Delujocih >= N div 2;                                     { da je dober. }
  { To, da smo prišli do sem, pomeni, da smo ugotovili, da deluje j pravilno.
    Primerjajmo zdaj ostale procesorje z njim. }
  for i := 1 to N do Pokvarjen[j] := not Enaka(j, i);
  { Zdaj imamo za vse procesorje v tabeli Pokvarjen podatke o tem,
    ali delujejo pravilno ali ne. }
end. {Preverjaj}
```

REŠITVE NALOG ZA DRUGO SKUPINO

R1995.2.1 Naloga: str. 3

Funkcija KajVrnem pove, kateri po velikosti je element x v tabeli t. Program ima drobno napako: če je v tabeli t več elementov z vrednostjo x, se funkcija vrti v neskončni zanki. O tem se lahko prepričamo takole: prej ali slej eden od števcev a in b pride do ene od celic z vrednostjo x. Ko se nato izvede zamenjava, je zdaj pač drugi števec pri celici z vrednostjo x; zato se vsaj ta števec pri naslednji iteraciji zunanje zanke ne bo nič premaknil. Tisti števec, ki se še premika, pa bo prej ali slej naletel na še kakšno drugo celico z vrednostjo x in odtlej se števca ne bosta nikoli več premaknila, program pa bo besno zamenjeval najdena x-a. (Če je v tabeli x en sam, do tega problema ne more priti, saj prej ali slej oba števca prideta do celice z vrednostjo x, ker pa zdaj oba kažeta na isto celico, pogoj a < b v zunanji zanki ni izpolnjen in zanka se konča.) Težavi se izognemo tako, da pogoj t[b] > x spremenimo v t[b] >= x ali pa po vsaki zamenjavi povečamo a za 1 in zmanjšamo b za 1; eno ali drugo nam zagotovi, da se a in b v vsaki iteraciji zunanje zanke zbližata za vsaj eno mesto, tako da se zanka ne more izvajati v nedogled.

Ta funkcija tudi preuredi elemente tabele: tisti, ki so manjši od x, pridejo v levi del tabele, tisti, ki so večji od x, pa v desni del tabele. Tako funkcijo se običajno uporablja pri algoritmu quicksort za urejanje podatkov: ker so zdaj vsi elementi levega dela manjši ali enaki od vseh v desnem delu, bomo tabelo čisto uredili že s tem, da bomo (z rekurzivnim klicem) uredili levi del posebej in nato še desni del posebej.

R1995.2.2 Naloga: str. 3

O vsakem bloku, ki se trenutno nahaja v pomožnem pomnilniku, moramo hraniti poleg številke tega bloka in njegove vsebine (struktura Podatki) še čas, kdaj je bil nazadnje uporabljen (da bomo vedeli, kateri najdlje ni bil uporabljen in ga je zato pametno zavreči, če potrebujemo prostor v pomnilniku), in zastavico, ki pove, ali je bil, odkar smo ga prebrali z diska, že kaj spremenjen (da bomo vedeli, ali ga je treba zapisati na disk, preden ga zavržemo iz pomnilnika).

Za čas zadnjega dostopa ni treba, da je to pravi čas; lahko imamo kar nek števec kot globalno spremenljivko in jo ob vsakem dostopu povečamo za 1. Že to je dovolj, da bomo lahko videli, ali je bil nek blok nazadnje uporabljen kasneje kot nek drug blok.

Podprograma ZapisiBlok in PreberiBlok si lahko pomagata s pomožnim podprogramom, ki se spodaj imenuje ZagotoviProstorZaEnBlok. Njegova naloga je poskrbeti, da bo v pomnilniku dovolj prostora za nov blok; če je treba, bo zavrgel nek drug blok iz pomnilnika. Preden zavržemo blok iz pomnilnika, je treba seveda tudi preveriti, če je bil kaj spremenjen, in ga v tem primeru najprej fizično zapisati na disk.

```
procedure ZapisiBlok(StBloka: integer; P: Podatki);
Če bloka StBloka še ni v pomnilniku:
    ZagotoviProstorZaEnBlok;
    Dodaj v pomnilnik blok StBloka s podatki P;
Sicer:
    Podatki bloka StBloka v pomnilniku := P;
Označi, da je bil blok StBloka spremenjen
in da je bil pravkar opravljen dostop do njega.
```

```
procedure PreberiBlok(StBloka: integer; var P: Podatki);
Če bloka StBloka še ni v pomnilniku:
    ZagotoviProstorZaEnBlok;
    PreberiBlokZDiska(StBloka, podatki tega bloka);
    Označi blok StBloka v pomnilniku kot nespremenjenega.
P := podatki tega bloka v pomnilniku;
Označi, da je bil pravkar opravljen dostop do bloka StBloka.

procedure ZagotoviProstorZaEnBlok;
Če je v pomnilniku že MaksPPBlokov blokov:
    Naj bo b tisti blok v pomnilniku, do katerega
    že najdlje nismo dostopali.
    Če je bil b že kdaj spremenjen:
        ZapisiBlokNaDisk(b, njegovi podatki);
    Zavrzimo b iz pomnilnika.
```

Ostane nam še vprašanje, kako naj bodo zapisi o posameznih blokih organizirani v pomnilniku. Želimo si, da bi naša podatkovna struktura čim bolj učinkovito podpirala naslednji dve operaciji:

1. preveriti, ali je nek blok že v pomnilniku (in priti do njegovih podatkov);

2. poiskati tisti blok, ki že najdlje ni bil uporabljen (tu moramo upoštevati tudi, saj se bo čas zadnje uporabe blokom pogosto spremenil, pač ob vsaki uporabi).

Bloke bi lahko na primer povezali v dvosmerno povezano verigo (*doubly linked list*), v kateri bi bili urejeni po času zadnjega dostopa. Na začetku seznama bi bil tisti, do katerega smo nazadnje dostopali, tako da bi bil vedno pri roki. Ko nek blok na novo uporabimo, moramo njegovo celico prestaviti na začetek seznama, kar pri dvosmerno povezanem seznamu ni težko. Kako pa bi najenostavneje ugotovili, ali je nek blok že v pomnilniku (in prišli do njemu pripadajoče celice v prej omenjenem dvosmernem seznamu)? Lahko bi preiskali cel seznam, vendar ima lahko ta do MaksPPBlokov elementov, kar najbrž vendarle ni tako malo. Lahko bi imeli tabelo, kjer bi za vsak blok na disku pisalo, ali je (oz. kje je) v pomnilniku; potem bi bilo preverjanje, ali je nek blok v pomnilniku, hitro, vendar bi imela ta tabela MaksBlokov elementov, kaj je najbrž precej preveč (saj je sorazmerno z velikostjo diska). Verjetno je še najbolje pripraviti razpršeno tabelo (*hash table*), kjer je poraba pomnilnika sorazmerna z MaksPPBlokov, čas iskanja pa je približno konstanten (neodvisen od MaksPPBlokov), če se bloki dovolj enakomerno razpršijo.

Našega upravitelja predpomnilnika bi bilo koristno dopolniti tudi s tem, da bi v ozadju občasno (npr. če že kakšno sekundo ni bil izveden dostop do diska) shranil na disk nekaj izmed tistih blokov, ki so trenutno v predpomnilniku in so označeni kot spremenjeni (kar pomeni, da je vsebina tega bloka že stara in neveljavna, trenutno aktualni podatki zanj pa so le v pomnilniku); na primer take, ki se že najdlje niso spremenili (pri takih upamo, da se tudi v bližnji prihodnosti ne bodo, tako da naše zapisovanje trenutne vsebine teh blokov na disk ne bo le nekoristna potrata časa). S tem bi skrbeli za to, da bi ob primeru izpada elektrike ali česa podobnega izgubili čim manj podatkov. Pri tem pa takšno občasno zapisovanje na disk v ozadju ne bi uporabnika nič motilo niti kako drugače upočasnjevalo dela z računalnikom.

R1995.2.3

Naloga: str. 4

Za začasno hranjenje znakov, ki smo jih že prebrali z vhodne linije, nismo pa jih še posredovali naprej vsem izhodnim linijam, bomo uporabili vrsto (tabela Vrsta v spodnjem programu). Znake, ki jih prebiramo z vhodne linije, dodajajmo na konec vrste (Rep). Izhodne linije pa imajo vsaka svoj kazalec na prvi znak v vrsti, ki ga še nismo posredovali posamezni izhodni liniji (Glava). Izmenično bomo poskušali prebirati vhodne znake in pisati na izhode; pri branju vhoda se ustavimo, če vhodnih znakov zmanjka ali pa vrsta doseže največjo dovoljeno dolžino (saj naloga pravi, da je količina pomnilnika, ki je na voljo za vrsto, omejena). Pri pisanju na izhode poskusimo pisati na vsako izhodno linijo po vrsti od prvega znaka, ki ga na to linijo še nismo uspešno poslali; ko ne moremo več pošiljati nanjo ali pa smo poslali že vse, pa se lotimo naslednje izhodne linije. Ker je največja dolžina vrste omejena vnaprej, lahko vrsto oblikujemo kot preprost krožni pomnilnik (*ring buffer*) — ko pri branju ali pisanju pridemo do konca vrste, začnemo spet na začetku.

```
program Multicasting(Output);
const
  IzhodM = 20;                       { število izhodnih kanalov (linij) }
  VrstaM = 10000;                    { največje možno število znakov v čakalni vrsti }
var
  Vrsta: array [1..VrstaM] of char;  { krožni izravnalni pomnilnik }
  Rep: integer;                      { kazalec na prvo prazno mesto v tabeli Vrsta }
  Glava:                             { za vsak izhodni kanal svoj kazalec na rep vrste }
    array [1..IzhodM] of integer;
  Dolz:          { za vsak izhodni kanal dolžina vrste (št. čakajočih znakov) }
    array [1..IzhodM] of integer;
  MaxDolz: integer;                  { dolžina najdaljše vrste: max(Dolz[*]) }
  Prebrano: integer;                 { število pravkar prebranih znakov }
  iz: integer;                       { številka izhodnega kanala }
  Konec: boolean;

  function Beri(var ch: char): boolean; external;
  function Pisi(iz: integer; ch: char): boolean; external;

begin
  Rep := 1;
  for iz := 1 to IzhodM do begin Glava[iz] := 1; Dolz[iz] := 0 end;
  while true do begin
    { Ugotovi število znakov v spominu (dolžino najdaljše vrste). }
    MaxDolz := Dolz[1];
    for iz := 2 to IzhodM do
      if Dolz[iz] > MaxDolz then MaxDolz := Dolz[iz];
    { Beri, dokler ne zmanjka znakov ali prostora v pomnilniku. }
    Konec := false; Prebrano := 0;
    while (MaxDolz + Prebrano < VrstaM) and not Konec do
      if not Beri(Vrsta[Rep]) then Konec := true
      else begin Prebrano := Prebrano + 1;
                 Rep := (Rep mod VrstaM) + 1 end;
    { Piši na vsak izhod, dokler se ne zatakne ali ne zmanjka znakov. }
```

```
    for iz := 1 to IzhodM do begin
      Dolz[iz] := Dolz[iz] + Prebrano;
      Konec := false;
      while (Dolz[iz] > 0) and not Konec do
        if not Pisi(iz, Vrsta[Glava[iz]]) then Konec := true
        else begin Dolz[iz] := Dolz[iz] − 1;
                   Glava[iz] := (Glava[iz] mod VrstaM) + 1 end;
    end; {for}
  end; {while}
end. {Multicasting}
```

Naloga: str. 5

R1995.2.4

Računalnik najprej sporoči svoj naslov vsem sosedom. Ob predpostavki, da so vsi računalniki ravnali enako, lahko računalnik sedaj začne sprejemati naslove svojih sosedov, kasneje pa še naslove vseh ostalih računalnikov, ki so povezani v omrežje. Ko sprejme sporočilo, računalnik najprej preveri, če je bilo to prvo sporočilo z danega naslova. To preveri s pomočjo tabele SmeriRacunalnikov, kjer je za vsak naslov shranjena smer, iz katere je prispelo prvo sporočilo. Prvo sporočilo je seveda prišlo po najkrajši poti, zato je tabela SmeriRacunalnikov obenem tudi tabela najkrajših poti. Ko ugotovi, da je sprejel prvo sporočilo z danega naslova, smer sprejema shrani v tabelo, sporočilo pa pošlje še v vse ostale smeri. Na tak način računalnik sprejema in posreduje sporočila, dokler ne izve smeri za vseh devetindevetdeset naslovov.

```
program KamNajPosljem;

  function MojNaslov: integer; external;
  function Poslji(VSmer: integer; Sporocilo: integer): boolean; external;
  function Sprejmi(var IzSmeri: integer; var Sporocilo: integer): boolean; external;

const
  SteviloRacunalnikov = 100;
  SteviloSmeri = 5;

var
  VSmer, IzSmeri, SprejetiNaslov: integer;
  SmeriRacunalnikov: array [1..SteviloRacunalnikov] of integer;
  Stevec: integer;
  SteviloPoznanihRacunalnikov: integer;

begin
  SteviloPoznanihRacunalnikov := 1;
  for Stevec := 1 to SteviloRacunalnikov do
    SmeriRacunalnikov[Stevec] := 0;
  for VSmer := 1 to SteviloSmeri do Poslji(VSmer, MojNaslov);
  while SteviloPoznanihRacunalnikov < SteviloRacunalnikov do begin
    repeat until Sprejmi(IzSmeri, SprejetiNaslov);
    if (SmeriRacunalnikov[SprejetiNaslov] = 0) and
            (SprejetiNaslov <> MojNaslov) then begin
      SteviloPoznanihRacunalnikov := SteviloPoznanihRacunalnikov + 1;
      SmeriRacunalnikov[SprejetiNaslov] := IzSmeri;
      for VSmer := 1 to SteviloSmeri do
        if VSmer <> IzSmeri then Poslji(VSmer, SprejetiNaslov);
    end; {if}
```

```
  end; {while}
end. {KamNajPosljem}
```

REŠITVE NALOG ZA TRETJO SKUPINO

R1995.3.1 Najzanimivejši del je podprogram RKSearch.[3] V osnovi gre za iskanje enega niza v drugem v času, ki je sorazmeren vsoti dolžin obeh nizov (namesto produktu dolžin, kar velja za naivni postopek iskanja podniza v nizu). Naloga: str. 5

Naiven postopek za iskanje niza p (dolžine LP) v nizu a (dolžine LA) bi pregledal vse podnize a-ja, dolge po LP znakov (torej a[1..LP], a[2..LP + 1], ..., a[LA − LP + 1..LA]), in vsakega primerjal z nizom p. Če imamo smolo in se p vedno čisto ujema z opazovanim podnizom, bo vsaka taka primerjava izvedla LP primerjav posameznih znakov. Časovna zahtevnost celotnega postopka je zato približno sorazmerna s produktom LA · LP.

Recimo pa, da bi znali vsakemu nizu prirediti nek celoštevilski „indeks“. Enak niz dobi vedno enak indeks, različni nizi pa po možnosti različne indekse (čeprav se ne da povsem izogniti temu, da včasih različni nizi dobijo isti indeks). Potem lahko za začetek primerjamo indeks niza p z indeksom posameznega podniza a[i..i + LP − 1]; če se indeksa razlikujeta, vemo, da je p različen od a[i..i + LP − 1]. Če pa sta indeksa enaka, je prav mogoče, da sta tudi niza enaka, povsem nujno pa to ni, zato ju moramo primerjati še na tradicionalen način, znak po znak (resna slabost programa iz naše naloge je, da tega ne dela; srečo ima, da pri nizih, s katerimi ima pri tej nalogi opravka, to ne pripelje do težav, pri kakšnih drugih nizih pa bi bile lahko težave).

Lepo pri tem postopku je, da lahko indekse definiramo tako, da je mogoče indeks niza a[i + 1..i + LP] izračunati zelo poceni, če že poznamo indeks niza a[i..i + LP − 1].

Izberimo si neki naravni števili d in q. Niz $s = s_1 s_2 \ldots s_n$ si predstavljajmo kot velikansko celo število, zapisano v d-iškem sestavu; posamezni znaki $s_1, \ldots, s_n$ so pri tem njegove „števke“. Niz s torej predstavlja število

$$d^{n-1} s_1 + d^{n-2} s_2 + \ldots + d^2 s_{n-2} + d\, s_{n-1} + s_n.$$

Za njegov indeks pa vzemimo vrednost

$$(d^{n-1} s_1 + d^{n-2} s_2 + \ldots + d^2 s_{n-2} + d\, s_{n-1} + s_n) \bmod q.$$

Indeksi so torej vedno cela števila iz množice $\{0, 1, \ldots, q-1\}$. Za q je koristno vzeti kakšno praštevilo; tako naredi tudi naš program.

Označimo indeks podniza a[i..i + LP − 1] z x; indeks naslednjega podniza, a[i + 1..i + LP], pa z x'. Iz gornje formule sledi, da je

$$x = (d^{LP-1} a[i] + d^{LP-2} a[i+1] + \ldots + d a[i+LP-2] + a[i+LP-1]) \bmod q$$

in

$$x' = (d^{LP-1} a[i+1] + d^{LP-2} a[i+2] + \ldots + d a[i+LP-1] + a[i+LP]) \bmod q.$$

[3] Imenuje se po Rabinu in Karpu, ki sta prva predlagala takšen postopek za iskanje podnizov v nizih. Glej npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 34.2 v prvi izdaji, 32.2 v drugi.

Koristna lastnost operacije mod (ostanek po deljenju) je, da jo lahko opravimo kadarkoli, pa vseeno dobimo enak odgovor. Drugače povedano, če izračunamo ostanek deljenja s q po vsaki aritmetični operaciji (tako bomo imeli vedno opraviti samo z majhnimi števili), dobimo enak rezultat, kot če bi opravili vse aritmetične operacije in šele potem izračunali ostanek deljenja s q.

Če primerjamo izraza za x in x' in upoštevamo lastnosti operacije mod, vidimo:

$$x' = ((x - d^{LP-1}a[i]) \cdot d + a[i + LP]) \bmod q.$$

Zaradi lastnosti operacije mod tudi sledi, da lahko v tem izrazu namesto d^{LP-1} vzamemo $d^{LP-1} \bmod q$ (to vrednost si naš program pripravi v spremenljivki dM). Tako vidimo, da lahko x' izračunamo iz x le s peščico računskih operacij, ne glede na to, kako velik je LP.

Podprogram Replace preprosto zamenja najdeni niz z nizom r: najprej premakne rep niza za potrebno število znakov levo ali desno in potem vpiše v niz nove vrednosti.

Program izpiše vrednost niza po vsaki zamenjavi in popravi mesto zadnjega iskanja tako, da se to začne po koncu zadnje zamenjave. Zadnja izpisana vrstica je torej: „`alibababababa in stirideset babarbabarov.`“

Poleg že omenjenega dejstva, da se zadovolji z ujemanjem indeksov in ne gre preverjat še ujemanja nizov, je v programu še nekaj drugih zanikrnosti. V podprogramu Replace imamo zanko:

```
for k := LA + 100 downto j + LP do a[k] := a[k − (LR − LP)];
```

Namen te zanke je premakniti „rep“ niza a: a[j + LP..LA] → a[j + LR..LA + LR − LP]. Nekaj napak v njej: (1) Če je LA + LR − LP (kar bo dolžina niza a po zamenjavi) večje od dolžine tabele (v našem primeru 255 znakov), bi morali javiti napako. (2) Če je začetna vrednost k = LA + 100 večja od 255, celica a[k] sploh ne obstaja. (3) Če pa bi bila razlika LR − LP večja od 100, ta zanka ne bi premaknila celega „repa“, ampak le sto znakov. Problema (2) in (3) rešimo tako, da zanko začnemo pri k = LA + LR − LP. (4) Zanka gre do k = j + LP namesto k = j + LR. Tako si v najboljšem primeru nakopava nekaj nepotrebnega dela (ker bo tisto, kar vpiše v a[j + LP..j + LR − 1], takoj povozila zanka v naslednji vrstici), če pa imamo opravka z neko pojavitvijo p-ja bolj na začetku niza a, je j majhen in celica a[k − (LR − LP)] je neveljavna.

Neroden je tudi stavek

```
j := j + Length(r) − Length(p);
```

v glavnem bloku programa. Ob vrnitvi iz podprograma RKSearch nam j pove (če je neničeln), da je RKSearch našel pojavitev niza p na mestih a[j..j + LP − 1]. Po zamenjavi se ta podniz zamenja z nizom r, ki pokrije mesta a[j..j + LR − 1]. Preiskovanje niza a se torej spodobi nadaljevati od a[j + LR] naprej. Podprogram RKSearch, ko ga pokličemo, primerja p najprej z a[j + 1..j + LP], torej bi morali mi zdaj pred naslednjim klicem RKSearch postaviti

```
j := j + Length(r) − 1;
```

Različica iz besedila naloge, torej tista z Length(p) namesto 1, bi na primer iz a = yxx, p = yx, r = xxxy dobila niz xxxxxxy namesto pričakovanega xxxyx.

Še ena majhna slabost našega programa je, da se kliče RKSearch po enkrat za vsako zamenjavo in pri tem vsakič znova tudi izračuna indeks niza p. Če bi bilo v nizu a veliko pojavitev p-ja, bi bilo lahko to zelo potratno; takrat bi res morali izračunati indeks p-ja samo enkrat in si ga potem zapomniti v kakšni spremenljivki.

R1995.3.2

Naloga: str. 6

Preprosta rešitev problema bi bila lahko takšna: za vsako mesto na zaslonu poglejmo, katere števke bi utegnilo predstavljati (torej: pri katerih števkah bi bili res prižgani vsi tisti segmenti, ki so prižgani na zaslonu). Potem preglejmo (npr. z rekurzijo) vse kombinacije števk, ki imajo prižgane vse segmente, vidne na zaslonu; pri primeru iz besedila naloge bi na primer za prvo mesto dobili števki 3 in 8, za drugo pa 0, 2, 3, 7, 8 in 9, torej bi morali pregledati 12 kombinacij: 30, 32, 33, 37, 38, 39, 80, 82, 83, 87, 88 in 89. Za vsako kombinacijo števk preverimo, če je njihova vsota deljiva z 10; če je tako, je ta kombinacija ena od kandidatk za pravo rešitev naloge.

Naloga zahteva, naj poiščemo kombinacijo z najmanj popravljenimi segmenti, torej z najmanjšo razliko med kombinacijo in stanjem na zaslonu; ker pa pridejo v poštev tako ali tako le kombinacije, pri katerih gorijo vsi segmenti, ki so prižgani tudi na zaslonu, je zahteva po najmanj popravljenih segmentih enakovredna zahtevi po najmanjšem skupnem številu prižganih segmentov. Če torej trenutna kombinacija ustreza ostalim zahtevam in ima poleg tega še manj prižganih segmentov kot najboljša doslej znana kombinacija, si jo zapomnimo kot novo kandidatko za najboljšo rešitev.

Ko z rekurzijo postopoma sestavljamo kombinacije števk, je zelo koristno že sproti, po postavitvi vsake števke, preveriti, če ni skupno število prižganih segmentov slučajno že preseglo števila prižganih segmentov pri najboljši doslej znani rešitvi. Če se namreč zgodi kaj takega, nima smisla postavljati še preostalih števk, saj tako dobljena rešitev gotovo ne bo boljša od najboljše doslej znane. Tako prihranimo veliko časa, ki bi ga drugače po nepotrebnem potratili za pregledovanje neobetavnih kombinacij. Takšnemu pristopu k pregledovanju prostora možnih kombinacij zato pogosto pravijo „razveji in omeji“ (*branch and bound*) — ob vsakem rekurzivnem klicu se iskanje razveji, ko poskušamo na trenutno mesto postavljati različne možne števke; obenem pa se poskušamo s pogoji, kot je v našem primeru tale s številom segmentov, čim bolj omejiti, da nam ne bi bilo treba v celoti pregledati neobetavnih delov prostora.

```
program PokvarjeniZaslon(Output);
type
  Stevka = 0..9;
  Segment = 1..7;
  Segmenti = set of Segment;
const
  OpisiStevk: array [Stevka] of Segmenti = (
    [1, 2, 3, 5, 6, 7], [3, 6], [1, 3, 4, 5, 7], [1, 3, 4, 6, 7],
    [2, 3, 4, 6], [1, 2, 4, 6, 7], [1, 2, 4, 5, 6, 7],
    [1, 3, 6], [1, 2, 3, 4, 5, 6, 7], [1, 2, 3, 4, 6, 7] );
  SegmentovPriStevki: array [Stevka] of integer= (6, 2, 5, 5, 4, 5, 6, 3, 7, 6);
  MaxN = 10;
type ZaslonT = array [1..MaxN] of Segmenti;

procedure PoisciVrednosti(var Zaslon: ZaslonT; N: integer);
var
```

```
  { Trenutna in najboljša doslej znana kombinacija števk. }
  Trenutna, Najboljsa: array [1..MaxN] of integer;
  { Za vsako mesto na zaslonu (od 1 do N) imamo seznam števk,
    ki imajo prižgane vse tam vidne segmente. To so Mozne[i, 1..StMoznih[i]]. }
  StMoznih: array [1..MaxN] of integer;
  Mozne: array [1..MaxN, 1..10] of Stevka;
  MinStSegmentov, StSegmentov: integer;

  procedure Rekurzija(Mesto: integer);
  var i, j, Vsota, StaroStSegmentov: integer;
  begin
    StaroStSegmentov := StSegmentov;
    for j := 1 to StMoznih[Mesto] do begin
      Trenutna[Mesto] := Mozne[Mesto, j];
      StSegmentov := StaroStSegmentov + SegmentovPriStevki[Trenutna[Mesto]];
      if StSegmentov >= MinStSegmentov then
        continue; { Brezupno, segmentov je že zdaj preveč. }
      if Mesto < N then begin Rekurzija(Mesto + 1); continue end;

      { Preverimo, če ima trenutna rešitev vsoto, deljivo z 10. }
      Vsota := 0;
      for i := 1 to N do Vsota := Vsota + Trenutna[i];
      if Vsota mod 10 <> 0 then continue;

      { To je najboljša doslej znana rešitev — zapomnimo si jo. }
      MinStSegmentov := StSegmentov;
      for i := 1 to N do Najboljsa[i] := Trenutna[i];
    end; {for j}
    StSegmentov := StaroStSegmentov;
  end; {Rekurzija}

var i: integer; s: Stevka;
begin {PoisciVrednosti}
  { Za vsako mesto na zaslonu poglejmo, katere števke bi utegnile
    povzročiti tako stanje segmentov, kot ga vidimo na zaslonu. }
  for i := 1 to N do begin
    StMoznih[i] := 0;
    for s := 0 to 9 do if Zaslon[i] − OpisiStevk[s] = [ ] then
      begin StMoznih[i] := StMoznih[i] + 1; Mozne[i, StMoznih[i]] := s end;
    if StMoznih[i] = 0 then exit; { Do takega stanja sploh ne more priti! }
  end; {for i}

  MinStSegmentov := 7 * N + 1; StSegmentov := 0;
  Rekurzija(1);

  if MinStSegmentov > 7 * N then WriteLn('Ni rešitev!')
  else begin
    Write('Najboljša rešitev:');
    for i := 1 to N do Write(' ', Najboljsa[i]);
    WriteLn;
  end; {if}
end; {PoisciVrednosti}

const
  Primer1: ZaslonT = ([1, 3, 4, 6, 7], [1, 3], [ ], [ ], [ ], [ ], [ ], [ ], [ ], [ ]);
  Primer2: ZaslonT = ([1, 3, 4, 6, 7], [1, 3], [2, 4], [1, 7], [1, 2], [4], [6, 7],
```

```
                    [1, 7], [1, 2, 3, 4], []);
begin {PokvarjeniZaslon}
  PoisciVrednosti(Primer1, 2);
  PoisciVrednosti(Primer2, 9);
end. {PokvarjeniZaslon}
```

Če naš zaslon nima veliko števk (torej: če N ni prevelik) in če na posameznem mestu ni možnih prav veliko števk, bo ta rešitev čisto dobra. Pri večjih zaslonih pa bi lahko število kombinacij, ki bi jih moral naš rekurzivni postopek pregledati, toliko naraslo, da bi se morali začeti ozirati za učinkovitejšo rešitvijo.

Zato si oglejmo še, kako bi se lahko naloge lotili z mehanizmi, ki se uporabljajo v logičnem programiranju z omejitvami (Constraint Logic Programming — CLP).[4] Problem bomo opisali z množico spremenljivk; vsaka od njih ima neko zalogo možnih vrednosti, poleg tega pa obstajajo tudi omejitve, ki zahtevajo, da morajo biti vrednosti več različnih spremenljivk v določeni zvezi. Omejitve nam lahko pomagajo, da nekatere vrednosti spremenljivk že vnaprej prepoznamo kot nemogoče; tako lahko upamo, da bomo pri izbiranju konkretnih vrednosti posameznih spremenljivk preizkusili čim manj neobetavnih možnosti.[5]

Na primer, recimo, da imamo spremenljivke x, y in z z zalogo vrednosti $\{1, 2, \ldots, 10\}$, poleg tega pa imamo omejitve $z < 7$, $x + y = z$ in $x \geq 2$. Na podlagi omejitve $z < 7$ lahko zalogo vrednosti spremenljivke z takoj zmanjšamo na $\{1, 2, \ldots, 6\}$. Zaradi omejitve $x + y = z$ potem vemo, da bo vsota vsaj 2, ker sta x in y oba velika vsaj 1; zato lahko zalogo vrednosti spremenljivke z zmanjšamo na $\{2, \ldots, 6\}$; obenem pa vemo, da niti x niti y ne smeta biti večja od 5 (ker je drugi seštevanec vsaj 1 in bi bila vsota sicer zanesljivo večja od 6, kar pa z ne sme biti), tako da lahko njuni zalogi vrednosti zmanjšamo na $\{1, \ldots, 5\}$. Nato lahko zaradi omejitve $x \geq 2$ zmanjšamo zalogo vrednosti spremenljivke x na $\{2, \ldots, 5\}$. Omejitev $x + y = z$ nam zdaj pove, da mora biti z vsaj 3 (ker je x vsaj 2 in y vsaj 1), tako da lahko z-jevo zalogo vrednosti zmanjšamo na $\{3, \ldots, 6\}$; pa tudi za y zdaj vemo, da sme biti največ 4, ker je x vsaj 2 in bi drugače prekoračili največjo dovoljeno vrednost z-ja, namreč 6: tako lahko y-ovo zalogo vrednosti zmanjšamo na $\{1, \ldots, 4\}$. Če nimamo nobenih drugih omejitev, zalog vrednosti ne bomo mogli nič bolj zmanjšati; končno stanje je torej: $x \in \{2, \ldots, 5\}$, $y \in \{1, \ldots, 4\}$, $z \in \{3, \ldots, 6\}$.

Tako oklеščene zaloge vrednosti so dobro izhodišče za preizkušanje konkretnih vrednosti spremenljivk. Spremenljivki x lahko poskusimo eno za drugo prirediti neko konkretno vrednost iz doslej dobljene zaloge vrednosti (torej iz $\{2, \ldots, 5\}$). To si lahko predstavljamo kot začasno dodatno omejitev oblike $x = \langle\text{neka konstanta}\rangle$. Ta dodatna omejitev nam omogoča še zmanjšati zalogi vrednosti ostalih spremenljivk. Nato poskusimo izbrati neko konkretno vrednost spremenljivke y, spet oklestiti zaloge vrednosti, nato pa enako storimo še za z. Če uspemo vsem prirediti konkretne vrednosti, ne da bi se ob usklajevanju omejitev zaloga vrednosti kakšne spremenljivke izpraznila, pomeni, da izbrane vrednosti spremenljivk res ustrezajo vsem omejitvam. V našem gornjem primeru bi lahko na primer začeli s tem, da bi izbrali $x = 3$, kar bi nam

[4]Za več o CLP gl. npr. Ivan Bratko, *Prolog Programming for Artificial Intelligence*, 3. izd. (2001), 14. pogl.; Thom Frühwirth et al., *Constraint Logic Programming: An Informal Introduction*, Tech. Rept. ECRC-93-5, `http://www.clps.de/html/reports.html`; Manuel Carro et al., *An Introductory Course on Constraint Logic Programming*, dec. 1998, `http://clip.dia.fi.upm.es/~vocal/public_info/index.html`.

[5]S tem postopkom usklajevanja omejitev se podrobneje ukvarja tudi naloga 1996.3.3.

omogočilo zmanjšati zalogo vrednosti z-ja na $\{4, \ldots, 6\}$ (ker je y vsaj 1) in nato zalogo y-a na $\{1, 2, 3\}$ (ker je z največ 6). Nato bi izbrali nek konkreten y, na primer $y = 2$, omejitev $x+y = z$ bi nam zmanjšala zalogo vrednosti z-ja na $\{5\}$, po izboru vrednosti $z = 5$ pa vse omejitve še vedno veljajo, kar pomeni, da smo našli eno od možnih rešitev: $x = 3$, $y = 2$ in $z = 5$ ustrezajo vsem omejitvam z začetka primera. Izbiranje vrednosti spremenljivk bi izvajali rekurzivno, tako da se lahko ob vrnitvi iz rekurzivnega klica posvetimo še drugim vrednostim trenutne spremenljivke. Tako bi na primer poskusili še $y = 1$ in $y = 3$, vse to še vedno pri $x = 3$, ko pa bi končali s tem, bi se lotili še drugih vrednosti spremenljivke x in tam spet preizkušali razne možne y in z. Glavno pa je, da po vsakem izboru neke vrednosti spet pregledamo omejitve, v katerih tista spremenljivka nastopa, da vidimo, če lahko zaradi tega izbora vrednosti kaj zmanjšamo zaloge vrednosti preostalih spremenljivk.

Pri našem problemu bi bilo koristno imeti po eno spremenljivko za vsako števko opazovanega zaslona; recimo jim $c_1, \ldots, c_n$. Njena začetna zaloga vrednosti bi bile kar vse tiste števke (od 0 do 9), ki vsebujejo vse segmente, ki so vidni na tem delu zaslona. Zdaj bi potrebovali omejitev, da mora biti vsota deljiva z 10; da pa posamezne omejitve ne bodo prezapletene, uvedimo raje posebno spremenljivko v in omejitvi $v = c_1 + \cdots + c_n$ ter $v \bmod 10 = 0$. Preostane nam še zahteva, da mora biti razlika v številu segmentov med c_i in tem, kar se res vidi na zaslonu, čim manjša; v ta namen uvedino spremenljivke r_i ter omejitve $r_i = \mathit{Razlika}(c_i, \mathit{Zaslon}[i])$, kar nam bo predstavljalo zahtevo, da je r_i razlika v številu vidnih segmentov med vrednostjo c_i in tem, kar se res vidi na i-ti števki zaslona. Končno je tu še spremenljivka r z omejitvama $r = r_1 + \cdots + r_n$ in $r < R$, pri čemer je R konstanta, ki predstavlja skupno razliko pri najboljši doslej najdeni rešitvi. Vsakič, ko najdemo neko novo rešitev, ki ustreza vsem dosedanjim omejitvam, bomo R zmanjšali in program s tem prisilili, da bo v bodoče odkrival le še boljše rešitve (če jih je kaj).

Program lahko zdaj deluje tako, da najprej uskladi vse omejitve, kolikor se to le da. Vsakič, ko se zaloga vrednosti neke spremenljivke zmanjša, si je treba ponovno ogledati omejitve, v katerih ta spremenljivka nastopa, ker nam lahko te zdaj omogočijo zmanjšati zaloge vrednosti še kakšnih drugih spremenljivk. Zato vzdržujemo med usklajevanjem množico spremenljivk, ki se jim je zaloga vrednosti spremenila in bo treba njihove omejitve ponovno pregledati; ko se ta množica izprazni, vemo, da smo z usklajevanjem zmanjšali zaloge vrednosti, kolikor se je le dalo.

V nadaljevanju izvajamo rekurzivne klice, ki skušajo po vrsti izbrati neke konkretne vrednosti za spremenljivke $c_1, \ldots, c_n$, po vsakem pa spet poženemo usklajevanje, da bi čim bolj zmanjšali zaloge vrednosti. Če se zaloga vrednosti kakšne spremenljivke pri tem izprazni, rekurzija ne bo mogla nadaljevati in vemo, da z doslej izbranimi vrednostmi pač ni mogoče dobiti nobene rešitve.

Spodnji program je napisan v pythonu, ker je tako lahko krajši, kot bi bil v pascalu (pa še tako je dovolj dolg). Nekaj opomb glede pythonove sintakse: metode z imenom **__init__** so pravzaprav konstruktorji, izrazi v oglatih oklepajih so seznami (pravzaprav dinamične tabele), izraz oblike [f(a) **for** a **in** L **if** p(a)] pa sestavi seznam, v katerem je vrednost f(a) za vsak tak element a seznama L, ki ustreza pogoju p(a).

```
StSegmentov = 7
OpisiStevk = [   [1, 2, 3, 5, 6, 7], [3, 6], [1, 3, 4, 5, 7], [1, 3, 4, 6, 7],
```

```
                  [2, 3, 4, 6], [1, 2, 4, 6, 7], [1, 2, 4, 5, 6, 7],
                  [1, 3, 6], [1, 2, 3, 4, 5, 6, 7], [1, 2, 3, 4, 6, 7]   ]
StStevk = len(OpisiStevk)

def Podmnozica(pod, mnozica):
  for i in pod:
    if not i in mnozica: return False
  return True

def RazlikaMnozic(odstej, odMnozice):
  return [i for i in odMnozice if not i in odstej]

class Spremenljivka:
  # Atributi tega razreda: zaloga — seznam možnih vrednosti;
  # zaloge — sklad prejšnjih seznamov „zaloga“ (za backtracking);
  # omejitve — seznam omejitev, v katerih nastopa ta spremenljivka;
  # min, max — najmanjša in največja vrednost iz seznama „zaloga“.
  vse = [ ]    # statičen seznam vseh spremenljivk v sistemu
  def __init__(self, zaloga = [ ]):
    self.omejitve = [ ]; self.NovaZaloga(zaloga[:], [ ])
    Spremenljivka.vse.append(self); self.zaloge = [ ]
  def NovaZaloga(self, novaZaloga, spremenjene):
    self.zaloga = novaZaloga
    if self.zaloga == [ ]: self.min = 9999; self.max = −9999
    else: self.min = min(self.zaloga); self.max = max(self.zaloga)
    if not self in spremenjene: spremenjene.append(self)

def PushZaloge():
  for s in Spremenljivka.vse: s.zaloge.append(s.zaloga[:])
def PopZaloge():
  for s in Spremenljivka.vse: s.NovaZaloga(s.zaloge.pop(), [ ])

class Omejitev: pass

class Manjsa(Omejitev):
  # Atributa: s — neka spremenljivka; meja — neko celo število.
  # To predstavlja omejitev „s < meja“.
  def __init__(self, spremenljivka, meja):
    self.s = spremenljivka; self.meja = meja
    self.s.omejitve.append(self)
  def uskladi(self, spremenjene):
    if self.s.max >= self.meja:
      self.s.NovaZaloga([i for i in self.s.zaloga if i < self.meja], spremenjene)

class Vsota(Omejitev):
  # Atributa: v — neka spremenljivka; s — seznam spremenljivk.
  # Predstavlja omejitev „v == s[0] + s[1] + ... + s[len(s) − 1]“.
  def __init__(self, vsota, sestevanci):
    self.v = vsota; self.sestevanci = sestevanci;
    self.v.omejitve.append(self)
    for s in self.sestevanci: s.omejitve.append(self)
  def uskladi(self, spremenjene):
    while True:
      minVsota = 0; maxVsota = 0; spremembe = False
      # Seštevanci lahko vplivajo na vsoto.
      for s in self.sestevanci: minVsota += s.min; maxVsota += s.max;
```

```
        if len(self.v.zaloga) > 0 and (minVsota > maxVsota
                                        or (self.v.min < minVsota or self.v.max > maxVsota)):
            self.v.NovaZaloga([i for i in self.v.zaloga
                                  if minVsota <= i <= maxVsota], spremenjene)
            spremembe = True
        # Vsota lahko vpliva na seštevance.
        for s in self.sestevanci:
            maxS = self.v.max − (minVsota − s.min)
            minS = self.v.min − (maxVsota − s.max)
            if len(s.zaloga) > 0 and (len(self.v.zaloga) == 0
                                        or (s.min < minS or maxS < s.max)):
                s.NovaZaloga([i for i in s.zaloga if minS <= i <= maxS], spremenjene)
                spremembe = True
        if not spremembe: return

class Ostanek(Omejitev):
    # Atributi: s — neka spremenljivka; d, o — celi števili.
    # Predstavlja omejitev „s % d == o“.
    def __init__(self, spremenljivka, delitelj, ostanek):
        self.s = spremenljivka; self.d = delitelj; self.o = ostanek
        self.s.omejitve.append(self)
    def uskladi(self, spremenjene):
        novaZaloga = [i for i in self.s.zaloga if i % self.d == self.o]
        if len(novaZaloga) != len(self.s.zaloga):
            self.s.NovaZaloga(novaZaloga, spremenjene)

class Razlika(Omejitev):
    # Atributi: r, c — dve spremenljivki; vidni — seznam segmentov.
    # Predstavlja omejitev „r == moč mnozice(OpisiStevk[c] − vidni)“.
    def __init__(self, razlika, stevka, vidniSegmenti):
        self.r = razlika; self.c = stevka; self.vidni = vidniSegmenti
        self.r.omejitve.append(self); self.c.omejitve.append(self)
    def uskladi(self, spremenjene):
        while True:
            spremembe = False
            # Števka lahko vpliva na razlike.
            mozneRazlike = [len(RazlikaMnozic(self.vidni, OpisiStevk[i])) for i in self.c.zaloga]
            novaZaloga = [i for i in self.r.zaloga if i in mozneRazlike]
            if len(novaZaloga) != len(self.r.zaloga):
                self.r.NovaZaloga(novaZaloga, spremenjene); spremembe = True
            # Razlike lahko vplivajo na števko.
            mozneStevke = [i for i in self.c.zaloga
                              if len(RazlikaMnozic(self.vidni, OpisiStevk[i])) in self.r.zaloga]
            if len(mozneStevke) != len(self.c.zaloga):
                self.c.NovaZaloga(mozneStevke, spremenjene); spremembe = True
            if not spremembe: return

def UskladiOmejitve(omejitve):
    while len(omejitve) > 0:
        spremenjene = [ ]    # Spremenljivke, ki se jim je zaloga vrednosti spremenila.
        for o in omejitve: o.uskladi(spremenjene)
        # V nadaljevanju bo treba uskladiti vse omejitve,
        # ki vsebujejo kakšno spremenljivko iz seznama „spremenjene“.
        omejitve = [ ]
        for s in spremenjene:
```

```
        for o in s.omejitve:
            if not o in omejitve: omejitve.append(o)

def PrirediVrednost(i):
    global n, spStevke, spRazlike, spVsota, spVsotaRazlik, omejitve

    if i == n:
        print "Rezultat: %s, vsota = %s, vsota razlik = %s" % (
            [c.zaloga[0] for c in spStevke], spVsota.zaloga, spVsotaRazlik.zaloga)
        assert len(spVsotaRazlik.zaloga) == 1
        # Našli smo novo najboljšo rešitev. Ta zdaj postane omejitev
        # pri nadaljnjem iskanju. Zadnja omejitev v seznamu „omejitve“
        # je ravno omejitev „vsota razlik < doslej najboljša znana vsota razlik“.
        omejitve[len(omejitve) − 1].meja = spVsotaRazlik.zaloga[0]
        return

    zaloga = spStevke[i].zaloga[:]
    for j in zaloga:
        PushZaloge()
        spStevke[i].NovaZaloga([j], [ ]) # Izberimo naslednjo možno vrednost j-te števke.
        UskladiOmejitve(spStevke[i].omejitve)
        PrirediVrednost(i + 1)
        PopZaloge()    # backtrack
        # Če smo medtem našli kakšno novo boljšo rešitev, lahko zaloge vrednosti
        UskladiOmejitve([omejitve[−1]])                     # še malo poklestimo.

def BruteForce(i, stevke, vsota, razlika, vidniSegmenti):
    global bfBest, bfBestR
    if i == n:
        if vsota % 10 == 0 and razlika < bfBestR:
            print "BruteForce: %s (vsota = %d, razlika = %d)" % (stevke, vsota, razlika)
            bfBest = stevke; bfBestR = razlika
    else:
        for j in spStevke[i].zaloga:
            novaRazlika = razlika + len(RazlikaMnozic(vidniSegmenti[i], OpisiStevk[j]))
            if novaRazlika < bfBestR:
                BruteForce(i + 1, stevke + [j], vsota + j, novaRazlika, vidniSegmenti)

def PoisciVrednosti(vidniSegmenti):
    global n, spStevke, spRazlike, spVsota, spVsotaRazlik, omejitve
    import time

    n = len(vidniSegmenti)
    # Pripravimo primerke razreda Spremenljivka.
    spStevke = [ ]      # spStevke[i] predstavlja števko na i-tem mestu na zaslonu,
    spRazlike = [ ]     # spRazlike[i] predstavlja stevilo segmentov, po katerih se to mesto
    for i in range(n):                          # zaslona razlikuje od števke spStevke[i].
        # Začetna zaloga vrednosti vsake števke je množica
        # tistih števk, ki vsebujejo vse segmente, ki so vidni na tem mestu.
        spStevke.append(Spremenljivka([j for j in range(StStevk)
                                    if Podmnozica(vidniSegmenti[i], OpisiStevk[j])]))
        spRazlike.append(Spremenljivka(range(0, StSegmentov + 1)))
    spVsota = Spremenljivka(range(0, n * (StStevk − 1) + 1))        # vsota števk
    spVsotaRazlik = Spremenljivka(range(0, n * StSegmentov + 1))    # vsota razlik

    # Za primerjavo poženimo še branch-and-bound.
```

```
  global bfBestR; bfBestR = StSegmentov * n + 1; t = time.clock()
  BruteForce(0, [], 0, 0, vidniSegmenti)
  print "Brute-force je tekel %.3f s." % (time.clock() − t)

  t = time.clock()
  omejitve = []
  omejitve.append(Vsota(spVsota, spStevke))        # spVsota je vsota števk
  omejitve.append(Ostanek(spVsota, 10, 0))         # spVsota % 10 == 0
  for i in range(n):
    omejitve.append(Razlika(spRazlike[i], spStevke[i], vidniSegmenti[i]))   # razlike
  omejitve.append(Vsota(spVsotaRazlik, spRazlike))   # vsota razlik
  # Naslednja omejitev je zdajle trivialna, kasneje pa bomo njeno mejo zmanjševali,
  # ko bomo odkrili kakšno rešitev. Tako bo ta omejitev vedno zahtevala,
  # da mora biti naslednja rešitev boljša od doslej najboljše znane.
  omejitve.append(Manjsa(spVsotaRazlik, StSegmentov * n))
  UskladiOmejitve(omejitve)
  PrirediVrednost(0)    # poženimo izbiranje vrednosti spremenljivk spStevke[0..n−1]
  print "CLP je tekel %.3f s." % (time.clock() − t)

PoisciVrednosti([[1, 3, 4, 6, 7], [1, 3]]) # Primer iz besedila naloge.
# Še en večji primer:
PoisciVrednosti([[1, 3, 4, 6, 7], [1, 3], [2, 4], [1, 7], [1, 2], [4], [6, 7], [1, 7], [1, 2, 3, 4]])
```

Naloga: str. 7

R1995.3.3

Najprej poiščimo vse ljudi, ki so skupni šefi vsem delavcem iz dane množice S; nato pa preverjajmo, če ni kakšen od njih šef kakšnega drugega (v tem primeru vemo, da tisti drugi ne more biti najbližji skupni šef). Kar ostane, so najbližji skupni šefi.

```
const n = ......; { Število oseb. }
type Oseba = 1..n; { Številka osebe. }
     MnozicaOseb = array [Oseba] of boolean;
function Sef(s, d: Oseba): boolean; external;

{ Funkcija vrne vrednost true, če je s posredni ali neposredni šef osebe d.
  Pri tem si pomaga s funkcijo Sef(s, d), ki pove, če je s neposredni šef osebe d.
  Pri reševanju naloge predpostavljamo, da je ta funkcija dana. }
function JeSef(s, d: Oseba): boolean;
var i: Oseba;
begin
  if Sef(s, d) then
    JeSef := true
  else begin
    JeSef := false;
    for i := 1 to n do
      if Sef(s, i) then if JeSef(i, y) then
        begin JeSef := true; exit end;
  end; {if}
end; {JeSef}

{ Funkcija vrne vrednost true, če je delavec x šef vseh delavcev iz
  množice S. V nasprotnem primeru funkcija vrne vrednost false. }
function SefVseh(x: Oseba; S: MnozicaOseb): boolean;
var i: Oseba;
begin
  SefVseh := true;
```

```
  for i := 1 to n do
    if S[i] then
      if not JeSef(x, i) then
        begin SefVseh := false; exit end;
end; {SefVseh}

{ Podprogram poišče vse skupne šefe delavcev iz množice S. }
procedure VsiSkupniSefi(var S: MnozicaOseb);
var vss: MnozicaOseb;
    i: Oseba;
begin
  for i := 1 to n do
    if not S[i] then vss[i] := SefVseh(i, S)
    else vss[i] := false;
    S := vss;
end; {VsiSkupniSefi}

{ Podprogram izloči iz mnozice šefov vse, ki niso najbližji
  glede na relacijo JeSef. }
procedure NajblizjiSefi(var S: MnozicaOseb);
var i, j: Oseba;
begin
  for i := 1 to n do
    for j := 1 to n do
      if S[i] and S[j] and (i <> j) then
        if JeSef(j, i) then S[j] := false;
end; {NajblizjiSefi}

var S: MnozicaOseb; i: Oseba;
begin {Sefi}
  for i := 1 to n do S[i] := false;
  S[123] := true; S[456] := true;
  VsiSkupniSefi(S);
  NajblizjiSefi(S);
end. {Sefi}
```

Tip MnozicaOseb bi bil lahko tudi **set of** Oseba, vendar bi bile s tem v praksi lahko težave. Mnogi prevajalniki ne dovolijo množic nad tipi z več kot nekim določenim številom možnih vrednosti, torej pri velikih n tipa **set of** Oseba najbrž ne bi mogli uporabiti. Poleg tega ni standardnega načina za dodajanje in brisanje posameznih elementov množice (nekateri prevajalniki imajo v ta namen podprograma Include in Exclude ali kaj podobnega); zato bi morali uporabiti konstrukt oblike S := S + [x], ki pa je lahko neučinkovit (če niso bili pisci prevajalnika posebej pozorni na to možnost) — če zares skonstruira množico [x] v neki pomožni spremenljivki in nato računa unijo, bo poraba časa za to operacijo sorazmerna z n, ne pa konstantna, kar bi pri dodajanju posameznega elementa sicer pričakovali.

Gornji program bi se dalo še izboljšati. Podprogram JeSef (ki sicer ni del naloge, ampak naloga pravi, da lahko predpostavimo, da že obstaja) lahko izvede ogromno rekurzivnih klicev za ene in iste ljudi, če so šefovski odnosi neugodno strukturirani. Na primer: a je šef b-ju in c-ju, tadva d-ju, slednji e-ju in f-ju, tadva pa g-ju; za povrhu naj bo še a šef y-u in slednji z-ju. Če zdaj kličemo JeSef(a, z), se bodo najprej izvedli rekurzivni klici za (b, z), (d, z), (e, z), (g, z), (f, z), (g, z), (c, z), (e, z), (g, z), (f, z), (g, z), šele nato pa (y, z), ki bi ugotovil, da je y (in zato tudi a) res nadrejen z-ju. Če bi imel tudi g podrejene delavce in slednji nekega skupnega podrejenega človeka, itd., bi se število klicev še povečevalo (eksponentno hitro glede na globino teh nadrejenosti). Če bi funkcija Sef dopuščala ciklične odnose (npr. x je šef y-u in obratno), pa bi se lahko celo zaciklali. Najmanj, kar bi lahko naredili, bi bila globalna tabela, v katero bi si JeSef zapisoval, katere pare delavcev je že obdelal in ali je v tistem paru res prvi šef drugega ali ne.

Še bolje bi verjetno bilo, če bi namesto podprograma JeSef(x, y) dobili podprogram, ki nam vrne vse nadrejene določenega človeka (ta bi se namesto na relacijo Sef(x, y), kakršno uporablja zdaj, verjetno opiral na neko funkcijo, ki bi za danega delavca vrnila seznam vseh njegovih neposrednih šefov). Podprogram VsiSkupniSefi bi potem le izračunal presek teh množic nadrejenih po vseh delavcih iz skupine S. Za učinkovito računanje presekov (če nimamo le 12 delavcev, ampak recimo na tisoče) bi bilo koristno, če bi bile te množice nadrejenih predstavljene z urejenimi seznami ali pa z razpršenimi tabelami. Taka predstavitev množic bi tudi omogočila, da bi šli zanki v podprogramu NajblizjiSefi le po članih množice šefov S, ne pa po vseh osebah, kar bi bil verjetno velik prihranek.

Lahko bi gledali tudi v obratno smer: za vsakega človeka bi pripravili množico vseh podrejenih in preverili, če vsebuje vse delavce iz skupine S. Tako bi ugotovili, ali je ta človek skupni šef vsem iz skupine S. Vendar pa je treba v tem primeru to ponoviti za vse ljudi (v prejšnjem odstavku pa le za vse iz S), poleg tega pa ima v tipični (drevesasti) organizaciji vsakdo le malo nadrejenih, podrejenih pa imajo nekateri ljudje zelo veliko.

Naloga: str. 7

R1995.3.4

Pri pisanju programa smo si pomagali z metodo pošiljanja žetona. Računalniki si preko omrežja pošiljajo natanko en žeton, na katerem je zapisan tekoči naslov. Prvi računalnik (tisti, ki je sporočilo sprejel iz smeri nič) na žeton zapiše številko ena. Žeton potem pošlje svojemu sosedu. Ko mu sosed vrne žeton, ga pošlje naslednjemu sosedu in tako naprej. Nazadnje vrne žeton pošiljatelju.

Tudi sosedje ravnajo podobno. Vsak sosed poveča naslov na žetonu za ena in vrne žeton pošiljatelju šele potem, ko je žeton že obhodil ves okoliš. Včasih se lahko zgodi, da se žeton vrne do pošiljatelja po krožni poti; v tem primeru ga pošiljatelj nespremenjenega vrne v smer, od koder je prispelo. Tako so na koncu označena vsa vozlišča, žeton pa je vsako povezavo prepotoval natanko dvakrat; v vsako smer enkrat.

```
program KdoSem;

  function Poslji(VSmer: integer; Sporocilo: integer); external;
  function Sprejmi(var IzSmeri: integer; var Sporocilo: integer): boolean; external;
  procedure NastaviSvojNaslov(Naslov: integer); external;

const SteviloSmeri = 11;

var
  PrvotnaSmer, Smer, StevecSmeri, Naslov: integer;
  PoznaneSmeri: array [1..SteviloSmeri] of boolean;
begin
  for StevecSmeri := 1 to SteviloSmeri do
    PoznaneSmeri[StevecSmeri] := false;
  repeat until Sprejmi(PrvotnaSmer, Naslov);
```

```
  if PrvotnaSmer = 0 then Naslov := 1
  else begin
    Naslov := Naslov + 1;
    PoznaneSmeri[PrvotnaSmer] := true;
  end; {if}
  NastaviSvojNaslov(Naslov);
  for StevecSmeri := 1 to SteviloSmeri do begin
    if not ZnanaSmer[StevecSmeri] then begin
      PoznaneSmeri[StevecSmeri] := true;
      if Poslji(StevecSmeri, Naslov) then
        repeat
          repeat until Sprejmi(Smer, Naslov);
          if Smer <> StevecSmeri then begin
            PoznaneSmeri[Smer] := true;
            Poslji(Smer, Naslov);
          end; {if}
        until Smer = StevecSmeri;
    end; {if}
  end; {for}
  Poslji(PrvotnaSmer, Naslov);
end. {KdoSem}
```

REŠITEV NALOGE PRVEGA ZAKLJUČNEGA TEKMOVANJA IZ ZNANJA RAČUNALNIŠTVA

Naloga: str. 8

R1995.Z Naš program naj bi se pognal za vsak vzorec posebej; torej je koristno, če z diska ne nalagamo celotnega slovarja, ker bomo morali to početi pri vsakem vzorcu znova in bomo s tem le zapravljali čas (sploh pa celoten slovar najbrž tako ali tako ne bi šel v pomnilnik, saj imajo besede iz datoteke `besede.txt` skupno 883 193 črk, naš program pa bo načeloma tekel v realnem načinu in lahko izkoristi največ 640 KB pomnilnika). Delo z diskom je v primerjavi z delom s podatki v pomnilniku precej počasno, zato naj naš program z diska pri vsakem vzorcu prenese čim manj podatkov. Zavedati se moramo tudi tega, da so pri delu z diskom naključni dostopi veliko dražji od zaporednih, zato je lahko neka rešitev, ki prebere več podatkov, boljša od neke take, ki jih prebere manj, vendar so ti bolj razsuti po datoteki in je potrebno zato veliko čakati, da se glava znajde na pravem mestu.

Besede bomo poskušali na disku organizirati tako, da pri iskanju bližnjih besed ne bo treba prebrati vseh, ampak le tiste, ki imajo res nekaj možnosti, da bi se izkazale kot bližnje. Tiste, ki jih bo pri delu s posameznim vzorcem vendarle treba prebrati, pa naj bodo čim bolj skupaj, ne pa razmetane naokoli po disku, tako da jih bo čim ceneje prebrati. Preprost kriterij, s katerim si lahko pomagamo, je dolžina: če je naš vzorec dolg n_0 črk, ima vsaka bližnja beseda vsaj n_0 in največ $n_0 + 2$ črk, torej bo za iskanje bližnjih besed dovolj, če preberemo le besede take dolžine. Še en koristen kriterij je število različnih črk v besedi: če ima naš vzorec m_0 različnih črk, jih imajo bližnje besede vsaj m_0 in največ $m_0 + 2$. Oba kriterija lahko tudi združimo in vidimo, da so ugodne le besede z naslednjimi lastnostmi (prvo število v paru pove dolžino besede, drugo pa število različnih črk v njej): (n_0, m_0), $(n_0 + 1, m_0)$, $(n_0 + 2, m_0)$, $(n_0 + 1, m_0 + 1)$, $(n_0 + 2, m_0 + 1)$ in $(n_0 + 2, m_0 + 2)$. Ostale tri možnosti, $(n_0, m_0 + 1)$, $(n_0, m_0 + 2)$ in $(n_0 + 1, m_0 + 2)$, odpadejo; na primer, če je beseda le za en znak daljša od našega vzorca, pa je vendarle bližnja, mora vsebovati vse črke vzorca in ima torej lahko za povrhu največ eno tako črko, ki je vzorec nima; zato ima lahko m_0 ali $m_0 + 1$, ne pa $m_0 + 2$ različnih črk.

Lahko bi torej besede v indeksni datoteki uredili tako, da pridejo za vsak par (n, m) skupaj podatki o besedah dolžine n z m različnimi črkami. Tej skupini besed recimo $W(n, m)$. Naš program bi izračunal dolžino n_0 in število različnih črk m_0 dobljenega vzorca in se potem posvetil ustreznim šestim skupinam besed, vsaki posebej (mogoče še manj kot šestim: če je na primer $n_0 = 12$, besed s 13 in 14 črkami pač ni; poleg tega pa za nekatere pare (n, m) sploh ni nobene take besede, na primer nobene z dvanajstimi enakimi črkami; v resnici je le 58 množic $W(n, m)$ nepraznih). Ko preberemo besede v pomnilnik, bi pač za vsako posebej preverili, ali je bližnja našemu vzorcu ali ne; tu se nam z učinkovitostjo ni treba toliko ukvarjati, saj bo program veliko večino časa najbrž tako ali tako porabil za branje z diska.

Vendarle pa je po svoje škoda, da bomo brali z diska vse te besede, saj se bo večinoma izkazalo, da je med njimi le peščica bližnjih. Če bi na primer kot vzorce zapovrstjo vzeli vse besede iz danega slovarja in pri vsaki pogledali, koliko besed moramo prebrati pri zgoraj opisanem postopku, koliko pa je zares bližnjih vzorcu, bi videli, da bi naš program prebral v povprečju pri vsakem vzorcu okoli 22 837,56 besed, bližnjih pa je povprečno le 12,37.[6] Dodatna nerodnost je tudi ta, da jih bomo morali brati v več kosih, kajti za nekatere (n, m) je skupna dolžina vseh besed iz $W(n, m)$ daljša od 2^{16} in jih torej naš 16-bitni program ne bo morel prebrati v enem kosu (najdaljša je $W(8, 6)$, ki ima 8 668 besed in torej skupno 69 344 črk).

Ena možna izboljšava je ta, da bi besede predstavili v kakšni bolj jedrnati obliki, ki bi nam omogočila za čim več nebližnjih besed ugotoviti, da niso bližnje, tako da se nam z njimi ne bi bilo treba več ukvarjati. Lahko bi na primer za vsako besedo izračunali „bitno karto“ — 32-bitno število, v katerem vsak bit pove, ali je neka črka v tej besedi prisotna ali ne (potrebujemo torej le 29 bitov, trije pa bi pač ostali neizkoriščeni). Beseda vsekakor ne more biti bližnja, če v njej manjka kakšna črka, ki jo vzorec ima, take primere pa lahko preprosto ugotovimo s primerjanjem bitnih kart: če ima vzorec bitno karto p, beseda pa w, mora veljati (p **and** w) = p, sicer beseda gotovo ni bližnja. Ta pogoj pa seveda še ni zadosten, saj na primer ne preverja tega, ali se vsaka črka vzorca pojavlja v besedi vsaj tolikokrat kot v vzorcu.

Količino podatkov, ki jih bo treba brati, lahko še zmanjšamo. Bitne karte za posamezno skupino $W(n, m)$ (saj bomo vedno brali celo skupino naenkrat) so čisto navadna cela števila; lahko jih uredimo in si zapomnimo pri vsakem le razliko med prejšnjim številom in tem. Lepo pri tem je, da so te razlike pogosto najbrž razmeroma majhne; lahko bi recimo razlike, manjše od 128, shranili kot en byte, ostale pa kot štiri. Pri tem moramo paziti le še na to, da jih bomo lahko tudi nedvoumno prebrali (na primer tako: pri razlikah, večjih od 128, premaknimo bite od bita 7 naprej za eno mesto proti levi in bit 7 prižgimo; ko

[6]Ko primerjamo različne algoritme in razmišljamo o tem, kateri je boljši, je seveda malce nerodno to, da ne vemo, na kakšni množici vzorcev bodo naš program na koncu preizkušali. Tu smo vzeli povprečja kar po množici vseh besed v slovarju, torej kot da bi za vzorce vzeli vse te besede; pri tem se zanašamo na upanje, da bo imela množica vzorcev, ki jih bodo na koncu res uporabili, v povprečju vsaj podobne lastnosti. Na primerno izbrani množici vzorcev pa bi znala biti razmerja med hitrostmi različnih algoritmov tudi precej drugačna. Več o tem gl. D. LaLoudouana, M. B. Tarare: *Data set selection*, NIPS 2002.

bomo to število zapisali na disk, bo imel tako prvi byte, ki hrani najnižjih osem bitov, na najvišjem bitu enico, tako da ga bo pri branju lahko ločiti od byta, ki sam po sebi predstavlja razliko, manjšo od 128; pri tistem pomiku v levo smo sicer izgubili najvišji bit, ampak saj vemo, da so najvišji trije biti pri nas tako ali tako neizkoriščeni). Izkaže se, da ta preprosta oblika kompresije bitnih kart skrajša zapis za približno 36 % in za toliko se zmanjša tudi količina podatkov, ki jih moramo pri branju bitnih kart prenašati z diska.

Naš program naj bi torej najprej prebral bitne karte za ustrezni $W(n,m)$, jih primerjal z bitno karto vzorca in tako dobil seznam besed-kandidatk, ki bi utegnile biti bližnje vzorcu. Nato mora prebrati vse te kandidatke in za vsako dokončno preveriti, ali je bližnja ali ni. Tu imamo sicer načeloma pri vsaki kandidatki po en naključen dostop do diska, vendar se izkaže, da gre to branje vendarle presenetljivo hitro, če beremo ves čas po naraščajočih naslovih. Zato je koristno besede $W(n,m)$ zapisati v enakem vrstnem redu, v kakršnem smo uredili njihove bitne karte. Tako bo že iz položaja bitne karte znotraj zaporedja bitnih kart jasno tudi, na katero besedo se ta bitna karta nanaša. V nasprotnem primeru bi morali ob vsaki bitni karti hraniti še indeks besede, ki ji ta karta pripada, to pa je spet potrata prostora in bi nam povečalo količino podatkov, ki jih moramo ob poizvedbi prebrati.

Z zgoraj opisano kompresijo podatkov smo pridobili še nekaj: izkaže se, da zdaj bitne karte za (n,m), $(n,m+1)$ in $(n,m+2)$ tudi v najslabšem primeru ne zasedejo vse skupaj več kot 45 530 bytov prostora. Torej se nam, če imamo skupine bitnih kart na disku razporejene po naraščajočih n in pri vsakem n še po naraščajočih m, ne bo treba ukvarjati z vsakim od šestih parov (n,m) posebej. Lahko bomo prebrali najprej bitne karte za (n_0,m_0); nato z enim samim branjem še vse tiste za (n_0+1,m_0) in (n_0+1,m_0+1); in nato spet z enim samim branjem še vse tiste za (n_0+2,m_0), (n_0+2,m_0+1) in (n_0+2,m_0+2). Tako si prihranimo še nekaj naključnih dostopov do diska.

Opisana rešitev z bitnimi kartami je že zelo dobra, vendar pa se da najti še boljše. Označimo z $W(n,m,a)$ množico vseh besed iz $W(n,m)$, ki vsebujejo tudi vsaj en primerek črke a. Če vsebuje tudi naš vzorec črko a, je jasno, da so možne bližnje besede tiste iz $W(n,m,a)$, ne pa tudi tiste iz množice $W(n,m) \setminus W(n,m,a)$. Če bi torej podatke o besedah na disku uredili po trojicah (n,m,a), bi bilo dovolj, če bi pri vsakem (n,m) delali le z besedami $W(n,m,a)$ za eno od črk a iz našega vzorca. (Seveda bi bilo pametno vzeti tisto črko, pri kateri bi morali z diska prebrati najmanj podatkov. To je še zlasti lepo, ker ima vzorec skoraj zagotovo vsaj kakšno razmeroma redko črko, pri kateri bo množica $W(n,m,a)$ prijetno majhna. Če smo prej kot povprečje prek poizvedb z vsemi besedami iz slovarja videli, da bi se morali ukvarjati pri vsaki poizvedbi s povprečno 22 837,54 besedami, pade zdaj to število na 3 382,85.)

Nerodno pa je to, da zdaj spada skoraj vsaka beseda v več množic $W(n,m,a)$, medtem ko je prej pripadala samo eni $W(n,m)$. Če hočemo hraniti podatke o vseh $W(n,m,a)$, bo tu skoraj vsaka beseda predstavljena na več mestih in količina podatkov na disku se nam zato precej napihne. Za branje to ni slabo, saj jih bomo morali brati manj kot prej; paziti pa moramo, da ne presežemo omejitve na 5 MB.

Če bi recimo hoteli hraniti celotne sezname besed za vse $W(n,m,a)$, tako kot smo jih prej za vse $W(n,m)$, bi videli, da so vsi skupaj zdaj dolgi 6 267 552 črk namesto dosedanjih 883 193; mi pa moramo v tistih 5 MB stlačiti še vse bitne karte! Lahko bi se odločili hraniti le sezname $W(n,m)$ kot doslej, bitne karte pa bi hranili za vsako $W(n,m,a)$ posebej. Vendar pa bi morali zdaj ob vsaki bitni karti hraniti še indeks, ki pove, katera po vrsti je v seznamu $W(n,m)$ beseda, na katero se nanaša ta bitna karta. Ker nima nobena množica $W(n,m)$ več kot 8 668 besed, bi tu zadostovala 16-bitna števila. Bitne karte bi nam tako zasedle 3 576 150 bytov prostora, tako da ga ostane dovolj še za sezname besed $W(n,m)$ (883 193 bytov) in še za knjigovodstvo. (Lahko se tudi pri indeksih odločimo za kompresijo — če se indeks pri neki bitni karti razlikuje za manj kot 128 od tistega pri prejšnji, ga shranimo le kot en byte. Tu se zanašamo na to, da bo tudi seznam $W(n,m)$, enako kot seznami bitnih kart, urejen po naraščajoči vrednosti bitne karte, zato indeksi, upajmo, ne bodo delali prevelikih skokov. Vendar se izkaže, da je ta prihranek majhen; poraba prostora za bitne karte se tako zmanjša na 3 177 443 bytov, podobno pa tudi količina podatkov, ki jih moramo v povprečju prebrati pri iskanju bližnjih besed.)

S tem, ko beremo bitne karte za manj besed, smo precej pridobili, s tem, da vsebujejo te bitne karte zdaj tudi indeks v zaporedje $W(n,m)$, pa smo nekaj izgubili, vendar je končni učinek še vedno zelo ugoden: v povprečju (čez poizvedbe z vsemi besedami iz slovarja) moramo za branje bitnih kart prebrati z diska okoli 75 % podatkov manj kot prej (pri vsaki poizvedbi povprečno 15 022 bytov namesto 58 188).

Nekaj smo izgubili tudi s tem, ker moramo pri vsakem (n,m) izbrati nek a in imamo potem en naključni dostop do diska za ta $W(n,m,a)$. Druga možnost bi bila ta, da bi bitne karte na disku zložili po naraščajočem n in nato pri vsakem n najprej po a, nato pa za vsak (n,a) še po naraščajočem m. Če nas v nekem trenutku zanimajo besede iz (n,m), $(n,m{+}1)$ in $(n,m{+}2)$, bi pogledali, kateri a ima najmanjšo vsoto $|W(n,m,a)|+|W(n,m+1,a)|+|W(n,m+2,a)|$ in potem zanj v enem zamahu prebrali bitne karte za vse tri skupine. (Poraba prostora na disku se pri tem ni nič spremenila, saj smo le drugače razmestili skupine z bitnimi kartami.) Res je sicer, da moramo brati zdaj nekaj več podatkov kot prej, ko smo si lahko za vsak m posebej izbrali najugodnejši a; vendar pa je črka, ki je redka pri enem m, najbrž kolikor toliko redka tudi pri kakšnem drugem; namesto prihranka 75 %, ki smo ga ugotovili na koncu prejšnjega odstavka, bi imeli zdaj malo manjši prihranek 65 % (povprečno bi morali za branje bitnih kart prebrati po 20 222 bytov pri vsakem vzorcu), kar še vedno ni slabo; in poleg tega imamo zdaj za branje bitnih kart le po en naključni dostop za vsak n, ne pa več po enega za vsak par (n,m).

Vendar pa smo si, odkrito povedano, vse to delo z bitnimi kartami nakopali le zato, da bi morali z diska prebrati čim manj podatkov. (Varčevanje s procesorjevim časom je precej nepomembno, ker naš program levji delež časa čaka na disk.) Najprej prebiramo bitne karte, izluščimo kup besed-kandidatk in nato vsako od njih preberemo še posebej, da vidimo, če je res bližnja. Mar ne bi bilo bolje, če bi prebrali že takoj kar besede same? Poskusimo torej raje zmanjšati količino potrebnega branja z bolj zgoščenim zapisom besed, ne pa z bitnimi kartami.

Za začetek lahko opazimo, da je zelo potratno, če namenimo vsaki črki cel byte, saj imamo le 29 možnih črk (črke slovenske abecede in še q, w, x in y) in je torej vsaki dovolj že 5 bitov, ne pa 8. Druga koristna zamisel je, da besede uredimo po abecedi (pravzaprav smo jih dobili tako urejene že v slovarju) in pri vsaki napišemo, v koliko prvih črkah se ujema s prejšnjo, nato pa zapišemo le črke od prvega neujemanja naprej (za število skupnih črk bi zadostovali štirje biti, saj imajo besede največ 12 črk, vendar bomo zaradi preprostosti tudi tu uporabili pet bitov). Ker so urejene po abecedi, se dve sosednji gotovo pogosto začneta na nekaj skupnih črk. Izkaže se, da nam vsaka od teh dveh izboljšav posebej prihrani okoli 30 % prostora, obe skupaj pa slabih 60 %. Če bi recimo hoteli po starem prebirati vse besede iz vseh šestih možnih $W(n, m)$, bi zneslo to povprečno po 188 098 bytov pri vsakem vzorcu, zdaj pa le še 77 740.

Še bolj koristno pa je to, da lahko zdaj shranimo v indeksno datoteko vse $W(n, m, a)$; namesto prej omenjenih 6 267 552 zasedejo le še 2 808 460 bytov. Če torej pri vsakem (n, m), ki nas zanima, izberemo za a tisto črko vzorca, ki bo zahtevala branje najmanj podatkov, in potem preberemo $W(n, m, a)$, bomo pri povprečni poizvedbi za branje teh besed porabili skupno le 12 907 bytov. Če pa pri vsakem n izberemo le en a, ki bo dal najkrajšo skupno dolžino zaporedij $W(n, m, a)$, $W(n, m + 1, a)$ in mogoče še $W(n, m + 2, a)$ (kolikor jih pač potrebujemo), bomo morali pri povprečni poizvedbi prebrati 17 175 bytov (imamo pa zato manj naključnih dostopov, namreč po enega za vsak n namesto po enega za vsak (n, m)). Vidimo lahko, da je zdaj količina podatkov, ki jih moramo prebrati, že manjša kot tista pri branju bitnih kart, pri čemer smo morali tam kasneje prebirati še besede kandidatke, česar zdaj ni več.

Razmislimo malo še o knjigovodskih podatkih. Pri rešitvi, do katere smo prišli na koncu, zadostuje, če na začetku indeksne datoteke zapišemo, koliko 5-bitnih znakov potrebuje posamezno zaporedje $W(n, m, a)$. To bi bila lahko tabela $12 \times 12 \times 29$ šestnajstbitnih števil. To pomeni 8 352 bytov, ki jih moramo prebrati na začetku vsake poizvedbe; mogoče bi lahko še kaj prihranili, če bi jih hranili kako bolj zgoščeno, ker je veliko seznamov praznih in so v tej tabeli zato tam ničle (na primer pri $m > n$, pa tudi pri kakšnih redkejših črkah). Rešitev z bitnimi kartami bi zahtevala podatke o velikosti množic $W(n, m, a)$ in še množic $W(n, m)$, saj imamo besede shranjene le v slednjih. Če ne delamo z množicami $W(n, m, a)$, je dovolj že tabela z velikostmi množic $W(n, m)$.

Za preverjanje, ali je neka beseda res bližnja danemu vzorcu, lahko uporabimo naslednji pogoj: beseda mora pripadati eni od šestih skupin $W(n, m)$ (torej za $n_0 \leq n \leq n_0 + 2$ in $m_0 \leq m \leq m_0 + (n - n_0)$) in vsake črke, ki se pojavlja v vzorcu, mora vsebovati vsaj toliko izvodov kot vzorec, vendar največ dva izvoda več kot vzorec. Ta pogoj je potreben: če je neka beseda bližnja, gotovo spada v eno od tistih šestih skupin $W(n, m)$ (o tem smo razmislili že zgoraj) in gotovo tudi ne vsebuje nobene črke manjkrat kot vzorec, niti ne v treh ali več izvodih več kot vzorec, saj vse to sledi že takoj iz definicije bližnjih besed. Naš pogoj pa je tudi zadosten: če neka beseda ustreza temu pogoju, očitno vsebuje vse črke vzorca v vsaj toliko izvodih kot vzorec, obenem pa že iz omejitve na dolžino ($n \leq n_0 + 2$) sledi, da ne more vsebovati več kot dveh dodatnih črk, torej je gotovo bližnja vzorcu.

Za konec lahko opozorimo še na en pomemben vir dostopov do diska, o katerem doslej nismo razmišljali. To je izvršljiva (`.exe`) datoteka našega programa. Ker se požene naš program za vsak vzorec posebej, računalnik pa nima diskovnega predpomnilnika, mora pred obdelavo vsakega vzorca ponovno prebrati to datoteko; torej, če je ta predolga, je lahko to že kar pomemben strošek, saj naš program vendarle ne bo bral tako veliko podatkov (recimo 20 KB za besede in še prej 10 KB za knjigovodske podatke). Po naših opažanjih se Turbo Pascal tu odreže precej bolje kot Turbo C, saj nam je slednji vztrajno pripravljal izvršljive datoteke, dolge čez 40 KB, Turbo Pascal pa je ostal pri dobrih 10 KB (oz. 6 KB, če izrežemo dele programa za tvorbo indeksa).

```
program Isci;

{ Skupne deklaracije }

const
   MaxDolz = 12;
   StCrk = 29;
   Crke: string = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ^[@';
   ImeIndeksa = 'index.dat';

type
   Beseda = string[MaxDolz];
   Glava = array [1..MaxDolz, 1..MaxDolz, 1..StCrk] of integer;
   SeznamCrk = array [1..MaxDolz] of 1..StCrk;

{ Vrne število različnih črk v besedi W. V sc vrne seznam teh črk. }
function RazlicneCrke(const W: Beseda; var sc: SeznamCrk): integer;
var i, j, M: integer;
begin
   M := 0;
   for i := 1 to Length(W) do begin
      j := 1; while (j < i) and (W[j] <> W[i]) do j := j + 1;
      if j = i then begin M := M + 1; sc[M] := Pos(W[i], Crke) end;
   end; {for}
   RazlicneCrke := M;
end; {RazlicneCrke}

{ Priprava indeksa }

type
   TextBuf = array [0..4095] of byte;
   BufText = record { za hitrejše delo s tekstovnimi datotekami }
      T: text; Buf: ↑TextBuf;
   end;

procedure Odpri(var ob: BufText; const ImeDat: string; Novo: boolean);
begin
   Assign(ob.T, ImeDat); New(ob.Buf);
   SetTextBuf(ob.T, ob.Buf↑);
   if Novo then Rewrite(ob.T) else Reset(ob.T);
end; {Odpri}

procedure Zapri(var ob: BufText);
   begin Close(ob.T); Dispose(ob.Buf) end;

function ToStr(N: integer): string;
   var S: string; begin Str(N, S); ToStr := S end;

{ Za vsak (n, m) pripravi po eno datoteko z vsemi besedami iz W(n, m). }
procedure RazdeliWnm;
var T: BufText; W: Beseda; N, M: integer; sc: SeznamCrk;
    Tnm: array [1..MaxDolz] of BufText;
begin
   for N := 1 to MaxDolz do begin
      for M := 1 to N do
         Odpri(Tnm[M], 'Wnm' + ToStr(N) + '-' + ToStr(M) + '.txt', true);
      Odpri(T, 'besede.txt', false);
```

```
    while not Eof(T.T) do begin
      ReadLn(T.T, W);
      if Length(W) = N then
        WriteLn(Tnm[RazlicneCrke(W, sc)].T, W);
    end; {while}
    Zapri(T); for M := 1 to N do Zapri(Tnm[M]);
  end; {for N}
end; {RazdeliWnm}

{ Pripravi po eno datoteko za vsak (n, a); v njej so besede iz W(n, m, a) za vse m,
  urejene po naraščajočih m. V G[n, m, a] vpiše število besed v množici W(n, m, a).
  Datoteke s seznami W(n, m) na koncu pobriše. }
procedure PripraviWna(N: integer; var G: Glava);
var M, A: integer; T, Ta: BufText; W: Beseda;
begin
  for M := 1 to N do begin
    for A := 1 to StCrk do begin
      G[N, M, A] := 0;
      Odpri(Ta, 'Wna' + ToStr(N) + '-' + ToStr(A) + '.txt', M = 1);
      if M > 1 then Append(Ta.T);
      Odpri(T, 'Wnm' + ToStr(N) + '-' + ToStr(M) + '.txt', false);
      while not Eof(T.T) do begin
        ReadLn(T.T, W);
        if Pos(Crke[A], W) > 0 then
          begin WriteLn(Ta.T, W); G[N, M, A] := G[N, M, A] + 1 end;
      end; {while}
      Zapri(T); Zapri(Ta);
    end; {for A}
    Erase(T.T);
  end; {for M}
end; {PripraviWna}

{ Doda vsebino vseh W(n, m, a) v indeksno datoteko. Vsaka od teh skupin je
  komprimirana posebej. V G[N, M, A] vpiše število 5-bitnih znakov v
  komprimiranem zapisu seznama W(n, m, a). Pomožne datoteke na koncu pobriše. }
procedure DodajWna(var F: file; N, A: integer; var G: Glava);
const BufLen = 4095;
var Buf: array [0..BufLen − 1] of byte; BufPos, Bit, D: integer;

  procedure Zapisi; { Zapiše vsebino bloka Buf na disk. }
  begin
    if (Bit > 0) and (BufPos < BufLen) then BufPos := BufPos + 1;
    BlockWrite(F, Buf, BufPos); BufPos := 0;
  end; {Zapisi}

  procedure Dodaj5Bitov(X: integer);
  begin
    Buf[BufPos] := (Buf[BufPos] or (X shl Bit)) and 255; Bit := Bit + 5; D := D + 1;
    if Bit >= 8 then begin { Nekaj bitov zapišimo v naslednji byte. }
      Bit := Bit − 8; BufPos := BufPos + 1;
      if BufPos = BufLen then Zapisi;
      Buf[BufPos] := X shr (5 − Bit);
    end; {if}
  end; {Dodaj5Bitov}
```

```
var M, i, j: integer; T: BufText; W, Wp: Beseda;
begin {DodajWna}
  Odpri(T, 'Wna' + ToStr(N) + '-' + ToStr(A) + '.txt', false);
  for M := 1 to N do begin
    BufPos := 0; Bit := 0; D := 0; Buf[BufPos] := 0; W := '';
    for i := 1 to G[N, M, A] do begin
      Wp := W; ReadLn(T.T, W);
      j := 1; if i > 1 then while (j < N) and (Wp[j] = W[j]) do j := j + 1;
      Dodaj5Bitov(j − 1); { Skupni začetek s prejšnjo besedo. }
      { Shrani še preostanek besede. }
      while j <= N do begin Dodaj5Bitov(Pos(W[j], Crke)); j := j + 1 end;
    end; {for i}
    Zapisi; G[N, M, A] := D;
  end; {for M}
  Zapri(T); Erase(T.T);
end; {DodajWna}

{ Na podlagi datoteke besede.txt pripravi indeksno datoteko,
  ki jo bomo uporabljali pri odgovarjanju na poizvedbe. }
procedure PripraviIndeks;
var N, A: integer; G: Glava; F: file;
begin
  RazdeliWnm;
  for N := 1 to MaxDolz do PripraviWna(N, G);
  Assign(F, ImeIndeksa); Rewrite(F, 1);
  BlockWrite(F, G, SizeOf(G));
  for N := 1 to MaxDolz do for A := 1 to StCrk do DodajWna(F, N, A, G);
  Seek(F, 0); BlockWrite(F, G, SizeOf(G));
  Close(F);
end; {PripraviIndeks}

{ Odgovarjanje na poizvedbe }

type
  Pojavitve = array [1..StCrk] of integer;
  Buffer = array [0..31807] of byte;
  Odmiki = array [1..MaxDolz, 1..MaxDolz, 1..StCrk] of longint;

{ Koliko bytov potrebujemo za zapis N 5-bitnih simbolov? }
function PakDolz(N: longint): integer;
  begin PakDolz := (N * 5 + 7) div 8 end;

{ Za vsako trojico (n, m, a) izračuna, na katerem odmiku v
  datoteki se zacnejo podatki za W(n, m, a). }
procedure IzracunajOdmike(const G: Glava; var O: Odmiki);
var N, M, A: integer; OO: longint;
begin
  OO := SizeOf(G);
  for N := 1 to MaxDolz do for A := 1 to StCrk do for M := 1 to N do
    begin O[N, M, A] := OO; OO := OO + PakDolz(G[N, M, A]) end;
end; {IzracunajOdmike}

procedure PoisciBliznje(const P: Beseda);
var Buf: ↑Buffer; BufPos, Bit: integer; { Blok z besedami v komprimirani obliki. }
   Znakov: integer; { Število še neprebranih 5-bitnih znakov v bloku Buf. }
```

```
  { Vrne naslednjih 5 bitov iz bloka Buf. Poveča Bit in BufPos, zmanjša Znakov. }
  function Beri5Bitov: integer;
  var X: integer;
  begin
    X := (Buf↑[BufPos] shr Bit) and 31; Bit := Bit + 5;
    if Bit >= 8 then begin { Nekaj bitov vzemimo še iz naslednjega byta. }
      Bit := Bit − 8; BufPos := BufPos + 1;
      X := (X or (Buf↑[BufPos] shl (5 − Bit))) and 31;
    end; {if}
    Beri5Bitov := X; Znakov := Znakov − 1;
  end; {Beri5Bitov}

var
  F: file; G: ↑Glava; O: ↑Odmiki;
  M: integer; PojP: Pojavitve; CrkeP: SeznamCrk; { Lastnosti vzorca P. }

  { Pregleda besede iz množice W(NN, MM, A), ki se morajo nahajati
    v bloku Buf od indeksa BufPos naprej. Če je LeCePrva = true,
    se vsaka beseda izpiše le, če ne vsebuje nobene črke, manjše od A.
    To se uporablja, če je vzorec prazen niz in hočemo v resnici
    izpisati vse besede iz W(NN, MM), vsako natanko enkrat. }
  procedure PreglejWnma(NN, MM, A: integer; LeCePrva: boolean);
  var i: integer; PojW: Pojavitve; Bliznja: boolean; CrkeW: SeznamCrk; W: Beseda;
  begin
    Znakov := G↑[NN, MM, A]; Bit := 0;
    while Znakov > 0 do begin
      i := Beri5Bitov + 1; { Skupni začetek s prejšnjo besedo. }
      Bliznja := true;
      while i <= NN do begin { Preberimo preostanek besede. }
        CrkeW[i] := Beri5Bitov;
        if LeCePrva and (CrkeW[i] < A) then Bliznja := false;
        i := i + 1;
      end; {while}
      if not Bliznja then continue; { Vsebuje črko, manjšo od A. }

      { Preverimo, če je res bližnja. }
      for i := 1 to M do PojW[CrkeP[i]] := PojP[CrkeP[i]];
      for i := 1 to NN do PojW[CrkeW[i]] := PojW[CrkeW[i]] − 1;
      i := 1; while (i <= M) and Bliznja do
        if (PojW[CrkeP[i]] < −2) or (PojW[CrkeP[i]] > 0)
        then Bliznja := false else i := i + 1;

      if Bliznja then begin { Če je bližnja, jo izpišimo. }
        W[0] := Chr(NN);
        for i := 1 to NN do W[i] := Crke[CrkeW[i]];
        WriteLn(W);
      end; {if Bliznja}
    end; {while}
    if Bit > 0 then BufPos := BufPos + 1;
  end; {PreglejWnma}

var N, NN, A, i, j, KA: integer; D, KD: longint;
begin
  Assign(F, ImeIndeksa); Reset(F, 1); New(G); New(O);
  BlockRead(F, G↑, SizeOf(G↑)); IzracunajOdmike(G↑, O↑);
  N := Length(P); M := RazlicneCrke(P, CrkeP); New(Buf);
```

```
  if N = 0 then begin { Prazen vzorec obravnavajmo posebej. }
    { Izpisati bomo morali vse besede dolžine 1 in 2. }
    D := 0; for A := 1 to StCrk do D := D + PakDolz(G↑[1, 1, A]) +
      PakDolz(G↑[2, 1, A]) + PakDolz(G↑[2, 2, A]);
    Seek(F, O↑[1, 1, 1]); BlockRead(F, Buf↑, D); BufPos := 0;
    for j := 1 to 2 do for A := 1 to StCrk do for i := 1 to j do
      PreglejWnma(j, i, A, True);
    exit;
  end; {if prazen vzorec}

  for A := 1 to StCrk do PojP[A] := 0;
  for i := 1 to N do PojP[Pos(P[i], Crke)] := PojP[Pos(P[i], Crke)] + 1;
  for j := 0 to 2 do if N + j <= MaxDolz then begin
    NN := N + j;
    { Zdaj se bomo ukvarjali z besedami iz W(NN, MM) za NN = N + j,
      M ≤ MM ≤ M + j. Naj bo A tista črka, pri kateri bomo morali
      prebrati najmanj podatkov. }
    for i := 1 to M do begin
      KA := CrkeP[i]; KD := PakDolz(G↑[NN, M, KA]);
      if j > 0 then KD := KD + PakDolz(G↑[NN, M + 1, KA]);
      if j > 1 then KD := KD + PakDolz(G↑[NN, M + 2, KA]);
      if (i = 1) or (KD < D) then begin A := KA; D := KD end
    end; {for i}
    if D <= 0 then continue;
    Seek(F, O↑[NN, M, A]); BlockRead(F, Buf↑, D); BufPos := 0;
    for i := 0 to j do PreglejWnma(NN, M + i, A, false);
  end; {for, if}
  Dispose(Buf); Close(F); Dispose(G); Dispose(O);
end; {PoisciBliznje}

begin {Isci}
  if ParamCount < 1 then PripraviIndeks
  else if Length(ParamStr(1)) <= MaxDolz then PoisciBliznje(ParamStr(1));
end. {Isci}
```

Program smo preizkusili na množici skupno 56 vzorcev, ki so jih uporabljali tudi na tekmovanju leta 1995 (ali pri testiranju ali pa kot primer za pomoč tekmovalcem); le računalnik je imel malo hitrejši procesor (P166). Zgoraj prikazani program potrebuje povprečno okoli 6,9 s, da odgovori na vse poizvedbe. Različica, ki zloži množice $W(n, m, a)$ po n, nato po m in šele nato po a (in lahko zato za vsak (n, m) izbere drug a ter zato prebere z diska malo manj podatkov, vendar pa ima zato po en dostop do diska za vsak (n, m) in ne le po enega za vsak n), porabi na isti množici vzorcev povprečno 8,1 s. Malo preprostejša rešitev, ki prebere komprimirane bitne karte za vse besede iz primernih $W(n, m)$, se ob njih odloči, katere besede bi bilo res treba pregledati, in nato prebere te (vsako z enim naključnim dostopom; celoten slovar ima v nekomprimirani obliki) ter izpiše zadetke, porabi 9,5 s. Naivna rešitev iz besedila naloge pa porabi 133 s časa. Izkaže se tudi, da ima na čas izvajanja teh rešitev zelo velik vpliv fragmentiranost diska. Da bi se pri merjenju časa izvajanja izognili vplivu pisanja rezultatov na disk, kar sicer zahteva naša naloga, so vsi ti programi pisali rezultate kar na zaslon.