20. državno tekmovanje v znanju računalništva (1996)

NALOGE ZA PRVO SKUPINO

1996.1.1 Nekega dne je programer vstal z napačno nogo in napisal naslednji program. **Kaj izpiše program** in kako bi ga popravil, da bi to izpisal hitreje?

Rešitev: str. 12

```
program SlepaKuraZrnoNajde;
const
  MaxT = 5;
var
  t: array [1..MaxT] of integer;
  i, j: integer;
begin
  for i := 1 to MaxT do t[i] := MaxT − i + 1;
  repeat
    i := Random(MaxT − 1) + 1;    { 1 ≤ i ≤ MaxT − 1 }
    j := t[i]; t[i] := t[i + 1]; t[i + 1] := j;
    i := 1;
    while (i < MaxT − 1) and (t[i] <= t[i+1]) do i := i + 1;
  until t[i] <= t[i + 1];
  for i := 1 to MaxT do WriteLn(t[i]);
end. {SlepaKuraZrnoNajde}
```

1996.1.2 **Opiši postopek**, ki preveri, če sta dve ogrlici enaki. Ogrlica je sestavljena iz N kroglic, ki so nanizane ena za drugo. Ogrlico predstavimo s tabelo znakov, kjer znaki predstavljajo barve kroglic.

Rešitev: str. 12

```
type Ogrlica = array [1..N] of char;
```

Ker so ogrlice krožne, moramo tako obravnavati tudi tabelo. Primeri ogrlic:

'1234567890'	je enaka	'7890123456'
'1234567890'	ni enaka	'1234567809'

Opišite postopek, ki na vhodu dobi dve ogrlici, na izhodu pa vrne vrednost `true`, če sta ogrlici enaki, in vrednost `false`, če nista.

1996.1.3 Računalnikar Janez Novak je na očetovem računalniku pisal domačo nalogo. Oče je nalogo našel in vanjo napisal svoje pripombe. K sreči je oče svoje pripombe dal med zlomljene oklepaje („<“ in „>“). Ker je naloge precej, se je Janez odločil, da očetovih komentarjev ne bo brisal ročno (peš), temveč bo za brisanje napisal program.

Rešitev: str. 13

Napiši program, ki bo iz poljubnega besedila pobrisal vse, kar se nahaja med znakoma „<“ in „>“, in besedilo spet izpisal.

Primer: Stavek

```
Danes je lepo, sončno vreme <kaj pa še,
danes dežuje> in ptički pojejo.
```

naj prevede v

```
Danes je lepo, sončno vreme  in ptički pojejo.
```

(Pozor: med `vreme` in `in` sta dva presledka.)

Rešitev: str. 14

1996.1.4

Hekerji so vsepovsod naokoli. Vdirajo v zasebnost, berejo in spreminjajo zaupna sporočila, zato je potrebno sporočila zakodirati. Ker želimo zagotoviti kar največjo varnost, smo se odločili, da bomo podatke kodirali s ključem, ki bo primerno velik. Ker pa računalniku ne moremo zaupati, da bo s funkcijo naključja (Random) izbral dovolj dober ključ, smo se odločili, da bomo naključni ključ vnesli sami. Pri tem nam pomaga ugotovitev, da je čas med dvema pritiskoma tipk pri tipkanju zelo naključen.

Napiši algoritem za vnos N-bitnega ključa, ki ga predstavimo z zaporednjem časov med dvema sosednjima pritiskoma na tipko. Na razpolago imaš naslednje funkcije (privzemi, da je N konstanta):

function Timer: integer; — Vrne trenutni čas v 1/1000 (tisočinkah) sekunde.

function KeyPressed: boolean; — Vrne true, če je uporabnik pritisnil kakšno tipko, sicer pa false.

function ReadKey: char; — Prebere pritisnjeno tipko. Če ni pritisnjena nobena tipka, funkcija počaka do naslednjega pritiska.

NALOGE ZA DRUGO SKUPINO

Rešitev: str. 15

1996.2.1

Zgodovinarji so v biltenu *20. državnega tekmovanja v znanju računalništva za srednješolce* iz leta 1996 med nalogami iz prve skupine odkrili naslednji program:

```
program Najden;

function xx(x: integer): integer;
const b = 10;
var y: integer;
begin
  y := 0;
  while x > 0 do begin
    y := y * b + x mod b;
    x := (x − y mod b) div b;
  end;
  xx := y;
end;

var x: integer;
begin
  for x := 1 to 1000 do
    if xx(x + xx(x)) = xx(x) + x then WriteLn(x);
end.
```

Zaradi skrajne neresnosti komisije se je besedilo naloge izgubilo, tako da zdaj nesrečni zgodovinarji ne vedo, **kakšen je** pravzaprav **problem**, ki so ga tekmovalci reševali. Jim lahko pomagaš?

Rešitev: str. 16

1996.2.2 Pri podjetju Telekom bi med mestoma Ljubljana in Amsterdam radi položili optični kabel za prenos podatkov. Ker se svetlobni signal v optičnem kablu hitro izgubi, ga je potrebno ojačati na vsake štiri kilometre, lahko pa ga ojačamo tudi pogosteje. Optični ojačevalniki so zelo dragi, zato so se odločili, da bodo kabel položili tako, da bodo ojačevalniki nameščeni na razdalji natanko štiri kilometre.

Pri tem pa so naleteli na majhen problem. Kanal, po katerem bo potekal optični kabel, prečka nekatera kratka območja, kot so na primer reke, ceste, mostovi in predori, na katerih se ojačevalnikov ne da namestiti. Podatke o ovirah so vnesli v računalnik, s programom Postavi pa bodo izbrali lokacijo prvega ojačevalnika na razdalji, manjši od štirih kilometrov, in to tako, da noben izmed preostalih ojačevalnikov ne bo pristal na oviri. Lahko se zgodi, da ojačevalnikov ni mogoče postaviti na zahtevani način. V tem primeru naj program to sporoči.

Napiši program Postavi. Pri tem si pomagaj s podatki, zbranimi v naslednjih tabelah:

- **var** Zacetek: **array** [1..StOvir] **of** integer;
  Razdalja med začetkom kabla in začetkom ovire (v metrih).

- **var** Konec: **array** [1..StOvir] **of** integer;
  Razdalja med začetkom kabla in koncem ovire (v metrih).

StOvir je torej število vseh ovir. Ovira z začetkom 100 in koncem 200 pomeni, da ojačevalnika ne smemo postaviti $100, 101, \ldots, 199$ m od začetka kabla, lahko pa ga postavimo 200 m od začetka kabla. Predpostaviš lahko, da je Zacetek[i] $<$ Konec[i] $<$ Zacetek[i] $+ 4000$.

1996.2.3 Gospa Bogataj je navdušena zbirateljica bisernih ogrlic. Za obisk računalniškega tekmovanja si je izbrala pisano ogrlico z rdečimi, modrimi in brezbarvnimi biseri.

Rešitev: str. 17

Zjutraj jo je na poti opazil ropar Dolgoprst, sicer star znanec policije, ki se mu zdi stara ženica lahek plen. Odločil se je, da bo ogrlico pretrgal, potem pa z vsakega konca ogrlice snel nekaj biserov. Ogrlico bo pretrgal tako, da mu bodo v vsaki roki ostali samo biseri ene barve in brezbarvni (modri in brezbarvni ali rdeči in brezbarvni).

Napišite program, ki bo Dolgoprstu pomagal pretrgati ogrlico tako, da mu bo v vsaki roki ostalo čimveč biserov. Ogrlica je podana kot niz znakov (`m`, `b`, `r`). Pri tem pazite na to, da se desni konec niza krožno nadaljuje od začetka naprej in obratno.

Primer ogrlice s 45 biseri:

```
 5×    2×  3×   4×     6×      5×   3×   5×   2×  4×     6×
mmmmm  bb  rrr  bbbb  rrrrrr  bbbbb rrr mmmmm rr  bbbb  mmmmmm
         ↑
```

Če ogrlico pretrga na mestu, označenem s ↑ (med sedmim in osmim biserom, torej za `bb` in pred `rrr`), mu ostane v levi roki 2`b` + 5`m` + 6`m` + 4`b` = 17 biserov, v desni pa 3`r` + 4`b` + 6`r` + 5`b` + 3`r` = 21 biserov. Ukradenih biserov je torej 38, kar je pri tej ogrlici tudi najboljši možni izkupiček.

Rešitev: str. 19

1996.2.4

Na računalniku, priključenem na Internet, opažamo povečano število poskusov vdorov. Na voljo imamo program, s katerim lahko posameznim računalnikom ali pa celi podmreži (delu omrežja) omejimo dostop do našega računalnika. Radi bi napisali še program, ki bi sproti popravljal spisek računalnikov in mrež, katerim dostop ni dovoljen.

Vsak računalnik v omrežju ima svoj naslov, ki je sestavljen iz dveh števil: prva določa podmrežo, druga pa računalnik v okviru podmreže. Odločili smo se za naslednji postopek omejevanja dostopa:

- vsakič, ko pride do poskusa vdora, si zapomnimo naslov računalnika in čas poskusa;
- če je poskus iz iste podmreže kot nek drug poskus, ki je že v tabeli, si zapomnimo kar celo podmrežo (naslov podmreže ponazorimo tako, da številko računalnika v naslovu nadomestimo z 0 — številke računalnikov v mreži se sicer začnejo z 1);
- po nekem določenem času od zadnjega poskusa vdora z nekega računalnika ali podmreže, ki ga označimo s TTL (*time-to-live* ali življenjska doba omejitve), dostop s tega računalnika oz. podmreže spet omogočimo.

Naslove in čase shranjujemo v urejen seznam. Pri upravljanju s seznamom si pomagamo z naslednjimi deklaracijami in podprogrami:

const TTL = 1800;

type Naslov = **array** [1..2] **of** integer;

function Cas: integer; — Vrne trenutni čas v sekundah.

procedure Dodaj(N: Naslov; t: integer); — V tabelo doda naslov N in čas t.

procedure Brisi(N: Naslov); — Iz tabele izbriše naslov N.

procedure Obnovi(N: Naslov; t: integer); — V tabeli spremeni čas t, ki je zapisan ob naslovu N.

function Poisci(N: Naslov): integer; — Če v tabeli obstaja naslov N, vrne pripadajoči čas, sicer vrne 0.

function Naslednji(**var** N: Naslov; **var** t: integer): boolean; — V tabeli poišče naslednji naslov po vrsti (prvega večjega od naslova N). Funkcija vrne true, če tak naslov obstaja, sicer vrne false.

Napiši podprogram Vsiljivec, ki ga bo sistem poklical vsakič, ko bo opazil poskus vdora. Naslov vdiralca dobi podprogram Vsiljivec kot parameter. Napiši tudi podprogram Sprosti, ki ga bo sistem občasno izvedel in z njim iz tabele odstranil zastarele zapise.

NALOGE ZA TRETJO SKUPINO

1996.3.1

Rešitev: str. 20

Dva računalnika si prek omrežja pošiljata in izpisujeta tekstovna sporočila, ki jih vtipkavajo uporabniki. Na obeh računalnikih se sočasno izvaja naslednji program:

```
var SprejemVkljucen: boolean;

procedure PosljiSporocilo(S: string);
var Odg: string;
begin
  Poslji('PošiljalBi');
  repeat Sprejmi(Odg) until Odg <> '';
  if Odg = 'RajeNe!' then WriteLn('Kolega ima izključen sprejem.')
  else if Odg = 'KarDaj!' then Poslji(S);
end; {PosljiSporocilo}

procedure SprejmiSporocilo;
var S: string;
begin
  if SprejemVkljucen then begin
    Poslji('KarDaj!');
    repeat Sprejmi(S) until S <> '';
    WriteLn(S);
  end
  else Poslji('RajeNe!')
end; {SprejmiSporocilo}

begin
  SprejemVkljucen := true;
  repeat
    Sprejmi(S);
    if S = 'PošiljalBi' then SprejmiSporocilo;
    if KeyPressed then begin
      WriteLn('Vtipkaj sporočilo: ');
      ReadLn(S);
      if S = '*' then SprejemVkljucen := not SprejemVkljucen
      else if S <> '' then PosljiSporocilo(s)
    end;
  until false;
end.
```

Zunanji podprogram Poslji(**var** S: string) pošlje sporočilo drugemu računalniku. Sprejeta sporočila se shranjujejo v predpomnilnik. Prebiramo jih s podprogramom Sprejmi(**var** S: string). Če je predpomnilnik prazen, Sprejmi vrne prazen niz. Funkcija KeyPressed: boolean vrne true, ko uporabnik pritisne kako tipko.

Razloži, kako deluje gornji program! V katerem primeru program ne deluje (skiciraj časovni potek „črnega scenarija“)? Kako bi ga popravil?

1996.3.2

Rešitev: str. 22

Firma PASSIVE FOOLS® je kupila superračunalnik in uporabnikom prodaja njegovo uporabo. Vsak uporabniški program Prog(Data) ima neke vhodne podatke Data in izpiše svoje rezultate v posebno datoteko.

Superračunalnik ima omejeno kapaciteto diskov, zato je pomembno, da je izpis vsakega uporabniškega programa končen, ker bi sicer izpisi enega programa zasedli celoten disk. Sistemski programer Lisjak B.™ je zato napisal program KoncenIzpis(Prog, Data), ki za poljuben program Prog in podatke Data ugotovi, če je izpis programa Prog pri vhodnih podatkih Data končen. Če je izpis programa Prog pri vhodnih podatkih Data končen, vrne vrednost true, sicer pa false (program Prog se pri podatkih Data zacikla, pri tem pa še vedno izpisuje svoje rezultate).

Uporabniki superračunalnika so praviloma strokovnjaki le na svojih področjih; pogosto se zgodi, da se njihovi programi zaciklajo ali pa sploh ničesar ne izpišejo. Taki programi zgolj kradejo računalnikov čas in jih ni smiselno izvajati.

Pomagaj sistemcu Lisjaku B.™ **napisati program**, ki zna za poljuben program Prog in podatke Data ugotoviti (dve nalogi):

1. ali program Prog pri vhodnih podatkih Data sploh kaj izpiše; in
2. ali se program Prog pri vhodnih podatkih Data ustavi ali zacikla.

Pri tem lahko tvoji programi kličejo poljubne programe, ki jih napišeš sam, pa tudi program KoncenIzpis(Prog, Data). Uporabljajo lahko vgrajeno proceduro Execute(Prog, Data), ki izvede program Prog na podatkih Data.

Primer uporabe programa KoncenIzpis: spodaj podani program Test ugotovi, če je prebrano število praštevilo. *Opomba:* predpostavimo, da je Random(n) nek generator psevdonaključnih celih števil, ki prej ali slej poljubno mnogokrat vrne vsako celo število med 1 in n.

```
program Delitelji(St);
begin
  while true do
    if St mod Random(St − 2) + 1 = 0 then
      WriteLn('Število ', St, ' ima delitelja med 2 in ', St − 1);
end. {Delitelji}

program Test;
begin
  ReadLn(Stevilo);
  if KoncenIzpis(Delitelji, Stevilo) then
    WriteLn(Stevilo, ' je praštevilo!')
  else
    WriteLn(Stevilo, ' ni praštevilo!');
end. {Test}
```

Rešitev: str. 24

1996.3.3

Napiši algoritem za usklajevanje dane skupine omejitev (relacij). V omejitvah nastopajo celoštevilske spremenljivke in na začetku poznamo za vsako spremenljivko neko začetno zalogo možnih vrednosti. Za posamezno omejitev, v kateri nastopajo recimo spremenljivke $X_1, \ldots, X_n$, pravimo, da je usklajena (konsistentna), če za vsak $i = 1, \ldots, n$ in za vsako možno vrednost x_i iz trenutne zaloge vrednosti spremenljivke X_i obstaja nek tak nabor vrednosti $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ iz trenutnih zalog vrednosti spremenljivk $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n$, da skupaj z vrednostjo $X_i = x_i$ ustrezajo opazovani omejitvi. Tvoj algoritem naj (če je to potrebno) zaloge vrednosti vseh spremenljivk zmanjša, tako da bodo vse omejitve v sistemu konsistentne. (Lahko se tudi izkaže, da to sploh ni mogoče.)

Dva primera:

$X = Y + Z$, $Y = Z + 1$, $Z > 10$

Recimo, da ima na začetku vsaka spremenljivka kot zalogo vrednosti vsa pozitivna cela števila. Po usklajevanju omejitev ostanejo zaloge vrednosti $X \in \{23, 24, \ldots\}$, $Y \in \{12, 13, \ldots\}$, $Z \in \{11, 12, \ldots\}$.[1]

$X = 3$, $X < Y$, $Z = 3$, $Y < Z$

Tega sistema omejitev ni mogoče uskladiti — zaloga vrednosti ene izmed spremenljivk se izprazni (pa kakršne koli so že bile prvotne zaloge vrednosti).

Pri pisanju algoritma si pomagajte z naslednjimi podprogrami in funkcijami:

- PreberiOmejitev — Funkcija vrne prebrano omejitev.
- UskladiOmejitev(Omejitev) — Podprogram uskladi zalogo vrednosti spremenljivkam, ki nastopajo v omejitvi Omejitev. To pomeni, da zaloge vrednosti teh spremenljivk, če je potrebno, zmanjša tako, da omejitev postane usklajena (po definiciji usklajenosti, ki smo jo navedli na začetku naloge).

  Primer 1: Če ima spremenljivka X zalogo vrednosti $X \in \{1, \ldots, 10\}$, ima po usklajevanju omejitve $X = 3$ zalogo vrednosti $X \in \{3\}$.

  Primer 2: Pred usklajevanjem imata spremenljivki X in Y zalogi vrednosti $X \in \{5, \ldots, 15\}$ in $Y \in \{1, \ldots, 10\}$. Po usklajevanju omejitve $X < Y$ sta zalogi vrednosti $X \in \{5, \ldots, 9\}$ in $Y \in \{6, \ldots, 10\}$.
- ŠteviloSpr(Omejitev) — Funkcija vrne število spremenljivk v omejitvi Omejitev.
- ImeSpr(Omejitev, Spr) — Funkcija vrne ime spremenljivke z indeksom Spr v omejitvi Omejitev.
- ZmanjšanaZalogaVrednosti(Omejitev, Spr) — Funkcija vrne true, če je bila zaloga vrednosti spremenljivke z indeksom Spr v omejitvi Omejitev zmanjšana (po uporabi podprograma UskladiOmejitev), sicer vrne false.
- PraznaZalogaVrednosti(Omejitev, Spr) — Funkcija vrne true, če ima spremenljivka z indeksom Spr v omejitvi Omejitev prazno zalogo vrednosti (po uporabi podprograma UskladiOmejitev), sicer vrne false.

1996.3.4

Rešitev: str. 25

Pogosto uporabljena metoda za tajen prenos sporočil je šifriranje sporočila s tajnim ključem (npr. z nekim dovolj dolgim skrivnim geslom ali naključnim številom). Pri tem sta algoritma za šifriranje in

[1]Tukaj tudi vidimo, da samo z usklajevanjem na nivoju posameznih omejitev (s čimer se ukvarja ta naloga) še ne pridemo nujno tudi do konkretnih rešitev celotnega sistema omejitev. V ta namen bi morali dodajati nove omejitve; če na primer dodamo omejitev $X = 23$, bi se zalogi vrednosti Y in Z zmanjšali na 12 oz. 13, kar predstavlja eno izmed konkretnih rešitev prvotnega sistema. Po drugi strani pa bi, če bi dodali omejitev $X = 30$, bi se zalogi vrednosti Y in Z zmanjšali na $Y \in \{12, \ldots, 19\}$ in $Z \in \{11, \ldots, 18\}$; za konkretno rešitev bi bila torej potrebna še ena omejitev oblike $Y = y$ ali pa $Z = z$.

Izkaže se celo, da dejstvo, da se je usklajevanje na nivoju posamičnih omejitev uspešno končalo, še ne pomeni nujno, da je sistem omejitev kot celota sploh rešljiv. Na primer: pri sistemu $X = Y$, $X = Z$, $Y \neq Z$ bi, če bi postavili na začetku vse zaloge vrednosti na $\{1, 2, \ldots\}$, ugotovili, da je vsaka omejitev sama zase že usklajena, kljub temu pa seveda nobena konkretna trojica števil (X, Y, Z) ne more ustrezati vsem trem omejitvam.

dešifriranje sicer lahko javno znana in tudi šifrirano sporočilo je dostopno nepovabljenim očem. Kljub temu pa je dešifriranje sporočila brez poznavanja ključa praktično nemogoče, saj bi preizkušanje vseh možnih ključev trajalo predolgo.

Odločili smo se, da bo ključ dolg 64 bitov.

Predpostavimo, da lahko vlomilec pri ugibanju ključa (npr. pri poskušanju dešifriranja prestreženega besedila z vsemi možnimi ključi) vedno ugotovi, ali je bil njegov poskus uspešen ali ne.

Ker nimamo na razpolago generatorja 64-bitnih naključnih števil, smo si skušali pomagati z generatorjem 16-bitnih psevdonaključnih števil:

```
procedure Random(var r: integer); external;
```

pri čemer je r neka spremenljivka, ki hrani 16-bitno število. Podprogram Random ob vsakokratnem klicu iz trenutnega števila r enolično izračuna novo 16-bitno število in ga shrani nazaj v r.

(*Opomba:* tako se obnašajo praktično vsi generatorji naključnih števil, ki jih najdemo v sistemskih knjižnicah ali vgrajene v programske jezike, le da je tam r običajno neka globalna spremenljivka in uporabnik podprograma nima neposrednega opravka z njo, pač pa bi bil Random v tem primeru funkcija, ki bi vrnila novo vrednost te globalne spremenljivke. Zaradi lažje formulacije naloge smo sintakso klica podprograma tu malo prilagodili.)

Naš algoritem za šifriranje je brez odvečnih podrobnosti približno takšen:

```
var
  j: integer;
  h, m: integer;            { trenutni čas: ura (0..23), minuta (0..59) }
  r: integer;               { naključno 16-bitno število }
  r1, r2, r3, r4: integer;  { 16-bitni deli končnega 64-bitnega ključa }
begin
  Time(h, m);               { v spremenljivkah h in m vrne trenuten čas }
  r := h * 60 + m;          { izračunamo začetno „naključno“ število }
  for j := 1 to 10000 do Random(r);    { malo pomešamo }
  Random(r); r1 := r;       { prvih 16 bitov ključa }
  Random(r); r2 := r;       { drugih... }
  Random(r); r3 := r;
  Random(r); r4 := r;

  Preberi(Besedilo);
  Sifriraj(Besedilo, r1, r2, r3, r4, SifriranoBesedilo);
  Poslji(SifriranoBesedilo);
end.
```

Vlomilec ima na razpolago računalnik z enakimi podprogrami za šifriranje in dešifriranje. Pozna tudi program, s katerim smo zašifrirali sporočilo. Ne pozna pa ključa (64-bitnega števila), s katerim je bilo prestreženo sporočilo šifrirano.

Vlomilčev računalnik zmore preizkusiti 1000 ključev na sekundo. Predpostavimo lahko, da je dešifriranje najzamudnejša operacija in da lahko ostale dele izvajanja vlomilčevega programa zanemarimo.

Izračunaj in utemelji grobo oceno, v kolikšnem času lahko vlomilec pričakuje, da bo uganil pravi ključ in dešifriral prestreženo sporočilo.

Predlagaj, kako bi lahko izboljšal zgornji program.

DRUGO ZAKLJUČNO TEKMOVANJE V ZNANJU RAČUNALNIŠTVA

1996.Z

Rešitev: str. 26

Tombola je igra, v kateri nastopa n (1000, 10000, 100000) kartic, na katerih je po 15 različnih števil med 1 in 100. V bobnu imamo 100 oštevilčenih kroglic (od 1 do 100). Zaporedoma iz bobna vlečemo kroglice, ki jih ne vračamo. Po vsaki izvlečeni kroglici poiščemo kartice, ki vsebujejo sama taka števila, ki so že bila izžrebana (t. j. vseh 15 števil na kartici je že bilo izžrebanih oz. nobeno od števil na kartici ni več v bobnu).

Napiši program Tombola, ki bo v čim krajšem času po vsaki izvlečeni kroglici izpisal število na novo zapolnjenih kartic (zadnja izvlečena kroglica je pokrila zadnje nepokrito polje na kartici).

V datoteki so zapisane kartice, ki so v igri. V vsaki vrstici datoteke je 15 različnih celih števil med 1 in 100, ki predstavljajo številke na eni kartici.

V rešitvi uporabi naslednje podprograme:

- **procedure** Pripravi — Podprogram, katerega klic mora biti prvi izvedljivi stavek v glavnem programu rešitve.
- **procedure** Pospravi — Podprogram, katerega klic mora biti zadnji izvedljivi stavek v glavnem programu rešitve.
- **function** Zrebaj(**var** St: integer): boolean — Funkcija izvleče naslednjo kroglico iz bobna in vrne njeno številko v spremenljivki St — vrednost funkcije je v tem primeru true. Ko v bobnu ni več kroglic, funkcija vrne false in vrednost spremenljivke St ni definirana.
- **procedure** Zapisi(Zadetkov: integer) — Podprogram objavi (zapiše) število zadetkov. Po vsaki izvlečeni kroglici (po klicu funkcije Zrebaj) naj program izračuna število na novo izpolnjenih kartic in pokliče Zapisi.

Čas izvajanja rešitve se začne meriti ob prvem klicu funkcije Zrebaj in neha meriti ob klicu podprograma Pospravi. Čas pred prvo izvlečeno kroglico se ne upošteva, mora pa biti krajši od minute.

Program bomo pognali trikrat:

1. `tombola 1000` — Podatki o 1000 karticah so v datoteki `k1000.txt`.
2. `tombola 10000` — Podatki o 10000 karticah so v datoteki `k10000.txt`.
3. `tombola 100000` — Podatki o 100000 karticah so v datoteki `k100000.txt`.

Ime datoteke z izvršljivim programom mora biti `tombola.exe`. Ime datoteke z izvornim programom mora biti `tombola.pas`. Iz datotek `k*.txt` si lahko pripravite drugače organizirane datoteke, ki se morajo skupaj z izvorno in izvršljivo kodo nahajati na področju `c:\finale`. Skupna velikost datotek ne sme presegati 20 MB.

V datoteki `tombola.txt` opiši svojo rešitev, ki naj obsega od 5 do 50 vrstic besedila. To besedilo je predloga za triminutni ustni zagovor pred komisijo.

Vsak tekmovalec ima na voljo disketo. Ob morebitnih nesrečah je tekmovalec sam odgovoren za svoje rešitve.

Tekmovanje bo trajalo 4 ure. Ob koncu tekmovanja mora biti delujoča rešitev v prej zahtevani obliki. Izpis rešitve mora biti identičen priloženi vzorčni rešitvi. Dodatni popravki, prevajanja, dodajanja, brisanja, preimenovanja,

... ne bodo dovoljena. Tekmovalci brez delujoče rešitve bodo diskvalificirani. Rešitve, ki bodo npr. spreminjale sistemsko uro ali se drugače grdo obnašale, bodo diskvalificirane.

Po končanem tekmovanju bo komisija prenesla vsebino področja `c:\finale` na računalnik komisije (120 MHz, 32 MB RAM, Win95), kjer bo javno preverila pravilnost in izmerila hitrost izvajanja programa. Na računalniku bo deloval diskovni predpomnilnik. Komisija bo upoštevala tudi vsebino izvorne kode in opis rešitve s triminutno predstavitvijo. Najboljše rešitve bo po tekmovanju komisija še enkrat natančno pregledala.

Naivna rešitev naloge je v datoteki `naivna.pas`. Potrudite se napisati rešitev, ki bo delovala hitreje.

```
program Tombola;
uses Zirija;
var
  fKarte: text;
  Izzrebane, Karta: set of 1..100;
  St, NovaSt: integer;
  Zadetkov: longint;
begin
  Pripravi;
  Izzrebane := [];
  if      ParamStr(1) = '1000'   then Assign(fKarte, 'k1000.txt')
  else if ParamStr(1) = '10000'  then Assign(fKarte, 'k10000.txt')
  else if ParamStr(1) = '100000' then Assign(fKarte, 'k100000.txt')
  else   Halt;
  while Zrebaj(NovaSt) do begin
    Izzrebane := Izzrebane + [NovaSt]; Zadetkov := 0;
    Reset(fKarte);
    while not Eof(fKarte) do begin
      Karta := [];
      while not Eoln(fKarte) do begin
        Read(fKarte, St); Karta := Karta + [St];
      end; {while}
      ReadLn(fKarte);
      if (NovaSt in Karta) and (Karta − Izzrebane = []) then
        Zadetkov := Zadetkov + 1;
    end; {while}
    Zapisi(Zadetkov);
  end; {while}
  Close(fKarte);
  Pospravi;
end. {Tombola}
```

Pomožni podprogrami se nahajajo v datoteki `zirija.pas`.

```
unit Zirija;

interface

  var fZreb, fRezultat: text;

  procedure Pripravi;
  procedure Pospravi;
  function Zrebaj(var St: integer): boolean;
```

```
  procedure Zapisi(Zadetkov: integer);

implementation

  uses Dos;
  var ZacetniCas, KoncniCas: longint;

  procedure Pripravi;
  begin
    Assign(fRezultat, 'rezultat.txt'); Rewrite(fRezultat);
    Assign(fZreb, 'zreb.txt'); Reset(fZreb);
    ZacetniCas := −1;
  end; {Pripravi}

  procedure Pospravi;
  var H, M, S, S100: word;
  begin
    GetTime(H, M, S, S100);
    KoncniCas := longint(H) * 360000 + longint(M) * 6000 +
                 longint(S) * 100 + longint(S100);
    WriteLn(fRezultat, 'Trajanje: ',
      (KoncniCas − ZacetniCas) / 100:0:1, ' sekund');
    Close(fZreb);
    Close(fRezultat);
  end; {Pospravi}

  function Zrebaj(var St: integer): boolean;
  var H, M, S, S100: word;
  begin
    if ZacetniCas < 0 then begin
      GetTime(H, M, S, S100);
      ZacetniCas := longint(H) * 360000 + longint(M) * 6000 +
                    longint(S) * 100 + longint(S100);
    end; {if}
    if Eof(fZreb) then
      Zrebaj := false
    else begin
      ReadLn(fZreb, St); Write(fRezultat, St);
      Zrebaj := true;
    end; {if}
  end; {Zrebaj}

  procedure Zapisi(Zadetkov: integer);
  begin
    WriteLn(fRezultat, ' ', Zadetkov);
  end; {Zapisi}

end. {Zirija}
```

Datoteke s podatki o karticah, ki so jih uporabljali na tekmovanju leta 1996, se dobijo na naslovu `http://www.brank.org/rtk/`.

Opozorimo na to, da so na tem tekmovanju programi že lahko tekli v zaščitenem načinu (DPMI), kar pomeni, da so lahko izkoriščali do 16 MB pomnilnika, le delo z bloki, večjimi od 64 KB, je bilo malo bolj nerodno.

REŠITVE NALOG ZA PRVO SKUPINO

Naloga: str. 1

R1996.1.1

Program izpiše urejene vrednosti tabele t, kar v našem primeru pomeni števila od 1 do MaxT.

V prvem stavku napolnimo tabelo t z vrednostmi od MaxT do 1. Zanka **repeat** opravlja nalogo t.i. naključnega urejanja podatkov (angl. *shuffle-sort*[2]). V vsaki ponovitvi te zanke naključno zamenjamo dva sosednja elementa tabele t, nato pa v zanki **while** preverimo, če so podatki urejeni v naraščajočem vrstnem redu (ta zanka pravzaprav to preveri le za vse elemente razen zadnjega in če je tu vse v redu, bo pogoj **until** preveril še, če je zadnji element vsaj tolikšen kot predzadnji). Ko so enkrat vsi podatki urejeni, se izvajanje zanke **repeat** konča in vrednosti tabele izpišemo.

Naloga sprašuje še, kaj bi lahko naredili, da bi program to, kar sicer izpiše že zdaj, izpisal hitreje. Po vsem, kar smo videli, lahko odgovorimo nekako takole:

```
const MaxT = 5;
var i: integer;
begin
  for i := 1 to MaxT do WriteLn(i);
end.
```

Naloga: str. 1

R1996.1.2

Ker sta ogrlici krožni, ne vemo vnaprej, katera kroglica druge ogrlice ustreza npr. prvi kroglici prve ogrlice (tudi če sta ogrlici zares enaki). Zato moramo poskusiti vse možne zamike (zunanja zanka v spodnjem programu); pri vsakem zamiku i pa moramo drugo ogrlico pred primerjavo v mislih krožno zamakniti za i mest. Spodnji program šteje kroglice od 0 do N − 1 namesto od 1 do N, ker je tako lažje uporabiti operator **mod**. Če se pri nekem zamiku ogrlici v celoti ujemata, lahko takoj končamo (spremenljivka Ok).

```
program AliStaOgrliciEnaki;
const
  N = 10;
var
  Ogrlica1, Ogrlica2: array [0..N − 1] of char;
  i, j: integer;
  Ok: boolean;
begin
  Ogrlica1 := '1234567890';
  Ogrlica2 := '7890123456';
  i := 0;
  repeat
    j := 0; Ok := true;
    while (j < N) and Ok do begin
      Ok := Ogrlica1[j] = Ogrlica2[(i + j) mod N]; j := j + 1;
    end; {while};
    i := i + 1;
```

[2] Izraz shuffle-sort se res uporablja za algoritem, ki naključno premeša podatke, preveri, če so urejeni, in to ponavlja, dokler ne dobi urejenega zaporedja; vendar pa se uporablja ta izraz tudi za nek algoritem za urejanje podatkov z več paralelno delujočimi enotami (Harold S. Stone: *Parallel processing with the perfect shuffle*, IEEE Trans. on Computers, C-20(2):153–61, Feb. 1971; Jai Menon: *A study of sort algorithms for multiprocessor database machines*, VLDB 1986, pp. 197–206).

```
  until Ok or (i = N);
  if Ok then
    WriteLn('Ogrlici sta enaki.')
  else
    WriteLn('Ogrlici nista enaki.');
end. {AliStaOgrliciEnaki}
```

Nalogo bi lahko rešili tudi tako, da bi enega od nizov podvojili (Ogrlica1 := Ogrlica1 + Ogrlica1; pravzaprav bi bilo dovolj že Ogrlica1 := Ogrlica1 + Copy(Ogrlica1, 1, Length(Ogrlica1) − 1)) in nato na poljuben način preverili, če se drugi (Ogrlica2) pojavlja v njem kot podniz.

R1996.1.3

Naloga: str. 1

Ko beremo vhodne podatke, hranimo v spremenljivki Oklepaj podatek o tem, ali se trenutno nahajamo znotraj oklepajev <...> ali ne. Če se ne, prebrane vhodne znake tudi sproti izpisujemo. Ko naletimo na oklepaj ali zaklepaj, pa vrednost spremenljivke Oklepaj primerno popravimo. Da nam bo lažje, smo predpostavili, da oklepaji in zaklepaji niso gnezdeni (npr. <...<...>...>); če bi bili gnezdeni, bi morala biti spremenljivka Oklepaj tipa integer in bi štela, koliko oklepajev < je trenutno odprtih; ob vsakem znaku < bi jo povečali za ena, ob vsakem > pa zmanjšali za ena. Ostale znake bi izpisovali le, če bi bila Oklepaj = 0.

```
program OckaNehaj(Input, Output);
var ch: char;
    Oklepaj: boolean;
begin
  Oklepaj := false;
  while not Eof(Input) do begin
    while not Eoln(Input) do begin
      Read(Input, ch);
      if ch = '<' then Oklepaj := true
      else if ch = '>' then Oklepaj := false
      else if not Oklepaj then Write(Output, ch);
    end; {while};
    ReadLn(Input);
    if not Oklepaj then WriteLn(Output);
  end; {while}
end. {OckaNehaj}
```

S primernim jezikom, kot je na primer `awk`, je lahko rešitev še veliko krajša:

```
BEGIN { RS = "<[^>]*>" }
{ printf "%s", $0 }
```

Program v `awku` je sestavljen iz pravil (*rules*), vsako pravilo pa iz vzorca (*pattern*) in akcije (*action*). Vzorec določa, kdaj se pravilo izvede, akcija pa, kaj se takrat zgodi. Tu imamo dve pravili; pri prvem je vzorec BEGIN, kar pomeni, naj se to pravilo izvede na samem začetku. `awk` razdeli svoj vhodni tok na zapise (*records*) in pri tem kot ločilo uporabi niz ali regularni izraz RS (privzeta vrednost RS je znak za konec vrstice). Prvo pravilo postavi RS na regularni izraz, ki se ujema z vsakim nizom oblike <...> (med oklepajema je lahko poljuben niz znakov, le nobenega > ne sme vsebovati — brez te dodatne zahteve bi se

ta izraz ujel z vsem besedilom od prvega < do zadnjega > v celi datoteki!). Zapisi, ki tako nastanejo, so torej ravno koščki besedila med komentarji; če vse izpišemo, smo dobili ravno celotno besedilo brez komentarjev. To naredimo z drugim pravilom; to vzorca sploh nima, zato se izvede po enkrat pri vsakem zapisu. Vsebino vzorca dobimo v nizu $0, izpišemo pa ga s stavkom printf, ki deluje podobno kot istoimenska funkcija v C/C++.

Naloga: str. 2

R1996.1.4

Naredimo podprogram, ki bo meril čas med pritisnjenimi tipkami, jemal spodnji bit teh časov in te bite vtikal posamezno v ključ. Pazimo na to, če se časi oziroma tipke ponavljajo (autorepeat, avtomatski keyboard stufferji). Torej, če je med dvema zaporednima pritiskoma minilo premalo časa, se za drugega raje ne zmenimo; podobno, če je med enim in drugim pritiskom minilo skoraj čisto enako časa kot med drugim in tretjim, pa gre ves čas za isto tipko, si mislimo, da uporabnik najbrž tišči tipko pritisnjeno in se za te pritiske raje ne zmenimo. Ko je ključ poln, podprogram konča delo.

```
const
  MinMed = 20;        { Minimalni sprejemljivi čas med dvema pritiskoma
                        (drugače drugi pritisk ignoriramo). }
  Natancnost = 2;     { Koliko niha čas med dvema zaporednima „pritiskoma“,
                        če uporabnik v resnici tišči tisto tipko ves čas pritisnjeno. }
  Dolzina = . . . ;   { Dolžina ključa v bitih. }
type
  KljucT = array [0..(Dolzina + 7) div 8 − 1] of byte;

procedure PripraviKljuc(var Kljuc: KljucT);
var Bit, Cas, PrejCas, Med, PrejMed: integer;
    Tipka, PrejTipka: char;
begin
  WriteLn('Tipkajte...');
  Bit := 0; PrejTipka := ReadKey; PrejCas := Timer; PrejMed := 0;
  while Bit < Dolzina do begin
    Tipka := ReadKey; Cas := Timer; Med := Cas − PrejCas;
    if (Med > MinMed) and ((Tipka <> PrejTipka) or
                           (Abs(Med − PrejMed) > Natancnost)) then begin
      if (Bit mod 8) = 0 then Kljuc[Bit div 8] := 0;
      Kljuc[Bit div 8] := Kljuc[Bit div 8] or ((Med and 1) shl (Bit mod 8));
      PrejMed := Med; PrejTipka := Tipka; Bit := Bit + 1;
      Write('.'); { Obvestimo uporabnika. }
    end; {if};
    PrejCas := Cas;
  end; {while}
  WriteLn; WriteLn('Ključ uspešno prebran.');
end; {PripraviKljuc}
```

Ena od slabosti tega pristopa je, če na primer Timer od nekod dobi čas v stotinkah sekunde, pa ga potem pomnoži z 10, da bi dobil tisočinke: potem bo Med vedno sod in naš ključ bo sestavljen iz samih ničel. Malo robustnejša rešitev bi bila, če bi gledali za tri zaporedne pritiske na tipko, ali je minilo več časa med prvim in drugim ali med drugim in tretjim (v prvem primeru bi dodali svojemu naključnemu zaporedju recimo bit 0, v drugem pa bit 1). To različico

uporabljajo na primer na `http://www.fourmilab.ch/hotbits/`, kjer namesto pritiskov na tipke spremljajo razpad neke radioaktivne snovi.

Glej tudi 4. nalogo za tretjo skupino (str. 7).

REŠITVE NALOG ZA DRUGO SKUPINO

R1996.2.1 Besedo ali stavek, ki se nazaj bere enako kot naprej, imenujemo *palindrom.* Podobno so *številski palindromi* števila, katerih zapis v desetiškem sistemu je z desne proti levi enak kot z leve proti desni (taki števili sta na primer 48584 ali 232).

Naloga: str. 2

Podprogram xx(x) vrne število, ki ga dobimo, če x zapišemo v desetiškem sistemu in ga preberemo v nasprotni smeri. Pri b = 10 je namreč x **mod** b ravno najbolj desna števka v desetiškem zapisu števila x, tako da vrstica y := y * b + x **mod** b v bistvu pripiše številu y na desni še x-ovo skrajno desno števko. Po tem prirejanju imata tako y kot x isto skrajno desno števko, zato x − y **mod** b v naslednji vrstici pravzaprav postavi skrajno desno števko števila x na 0, deljenje z b v isti vrstici pa skrajno desno x-ovo števko še pobriše. (Ker operator **div** tako ali tako zaokroži rezultat deljenja navzdol, bi lahko vzeli tudi preprosto x := x **div** b).

Glavni blok programa izpiše vsa cela števila x od 1 do 1000, za katera velja: če temu x prištejemo število (recimo mu $\hat{x}$), ki ga dobimo, če x preberemo z desne proti levi, je vsota neko palindromno število. Izkaže se, da ima to lastnost 291 števil. Primer takega števila je recimo 65, saj je $65 + 56 = 121$, torej palindrom.

Do tega, kateri so ugodni x, lahko pridemo tudi z razmislekom. Od števil $x < 10$ so ugodna 1, 2, 3 in 4, ostala pa ne. Oglejmo si zdaj dvomestna števila: $x = 10a + b$, torej $\hat{x} = 10b + a$. Kdaj je vsota $S := x + \hat{x} = 10(a + b) + (a + b)$ palindrom? No, če je $a + b < 10$, sta obe števki S-ja enaki, tako da je v redu. Temu pogoju ustreza 45 števil: 10..18, 20..27, 30..36, 40..45, 50..54, 60..63, 70..72, 80..81 in 90. Sicer pa je $a + b = 10 + c$ za nek $c \in 0..8$ in $S = 100 + 10(1 + c) + c$, kar je lahko palindrom le, če je $c = 1$, torej če se a in b seštejeta v 11. Ugodni x so torej 29, 38, 47, 56, 65, 74, 83 in 92.

Oglejmo si še tromestna števila: $x = 100a+10b+c$, $S = 100(a+c)+10(2b)+(a + c)$. Ločili bomo primere glede na to, ali pride pri seštevanju do prenosa naprej. (*i*) Ena možnost je na primer ta, da je $a+c < 10$ in $b < 5$; hitro vidimo, da je takih x-ov $45 \cdot 5 = 225$. (*ii*) Če $a + c < 10$ in $b \geq 5$, bo $2b = 10 + d$ za nek d in $S = 100(a + c + 1) + 10d + (a + c)$, kar ne more biti palindrom, če je res tromestno število; štirimestno pa je v primeru, ko je $a + c = 9$, S pa je oblike $1000 + 10d + 9$, kar tudi ne more biti palindrom. (*iii*) Če $a + c = 10 + d$ in $b < 5$, bo $1 + 2b < 10$ in S bo oblike $1000 + 100d + 10(1 + 2b) + d$. To je torej palindrom natanko tedaj, ko je $d = 1$ in $1 + 2b = d = 1$, torej $b = 0$, za a in c pa je potem 8 načinov, da se seštejeta v 11 $(2 + 9, 3 + 8, \ldots, 9 + 2)$. (*iv*) Če $a + c = 10 + d$ (za nek $d \in 0..8$) in $2b = 10 + e$ (za nek $e \in \{0, 2, 4, 6, 8\}$), ima S obliko $1000 + 100(1 + d) + 10(1 + e) + d$, torej je palindrom le, če je $e = d = 1$, kar pa ni mogoče, saj bi $e = 1$ pomenilo $2b = 11$.

Skupaj smo torej našteli 4 enomestne, $45+8$ dvomestnih in $45\cdot5+8$ tromestnih števil z iskano lastnostjo; to lastnost pa ima tudi število 1000, kar nam da vsega skupaj 291 števil.

Naloga: str. 3

R1996.2.2

Naj bo x položaj prvega ojačevalnika, na intervalu od z do k pa naj bo neka ovira. Kateri x so zaradi te ovire neugodni? Tisti, pri katerih pademo nekoč v interval med z in k, če začnemo pri x in prištevamo 4000. Naj bo $z = 4000q + z'$ za nek $0 \leq z' < 4000$ in $k = 4000r + k'$ za nek $0 \leq k' < 4000$. (Pri tem vemo, da je $r = q$ ali pa $r = q + 1$, ker bi bila ovira drugače zanesljivo dolga vsaj 4000 m, kar bi bilo brezupno.) Očitno velja, da na ojačevalnike $x + 4000s$ za $s < q$ in $s > r$ ta ovira ne more vplivati. Če je $r = q$, bo ovira mogla vplivati le na ojačevalnik $x + 4000q$, ki torej ne sme ležati na intervalu od z do k. Tako imamo pogoja, da ovira ni v nadlego: veljati mora $x + 4000q < z$ ali pa $x + 4000q \geq k$, kar pa je isto kot $x < z'$ oz. $x \geq k'$. Druga možnost, $r = q + 1$, pomeni, da lahko vpliva ta ovira na ojačevalnika $x + 4000q$ in $x + 4000r$; no, zanesljivo vedno velja $x + 4000q < k$ in $x + 4000r > z$, tako da bo moral prvi od omenjenih dveh ojačevalnikov ležati pred oviro, drugi pa za njo. To nam da pogoja $x + 4000q < z$ in $x + 4000r \leq k$ oziroma $x < z'$ in $x \geq k'$. (Primere, ko je $r = q + 1$, lahko enostavno ločimo od tistih, ko je $r = q$, takole: če je $r = q$, sledi iz $z < k$ tudi $z' < k'$; če pa je $r = q + 1$, mora biti $k' \leq z'$, saj bi bila sicer ovira daljša od 4000 m.)

Spodnji program torej lahko ravna takole: za vsako oviro $[z, k]$ izračuna $z' = z \bmod 4000$ in $k' = k \bmod 4000$. Vsaka ovira, pri kateri je $k' \leq z'$, nam dá pogoj, da mora biti $x \geq k'$, torej lahko za začetni x vzamemo največjo k' po vseh teh ovirah. Od tu naprej postopoma povečujemo x, če najdemo kakšno oviro, ki ji trenutni x ne ustreza; vsakič ga povečamo kar najmanj, vendar toliko, da ga tista ovira ne omejuje več: tako vemo, da ne bomo nobenega spregledali (v vsakem trenutku vemo, da bi kateri koli x, manjši od trenutnega, gotovo imel težave pri vsaj eni oviri). Torej: če najdemo oviro, za katero je $z' \leq x < k'$, moramo postaviti x na k', če pa najdemo táko, za katero je $k' \leq z' \leq x$, smo v težavah, saj vemo, da bi moral biti pri taki oviri x manjši od z', obenem pa očitno noben manjši x ni ustrezal vsem dosedanjim oviram.

```
program Postavi;
var
  Zacetek: array [1..StOvir] of integer;
    { razdalja med začetkom kabla in začetkom ovire (v metrih) }
  Konec: array [1..StOvir] of integer;
    { razdalja med začetkom kabla in koncem ovire (v metrih) }
  Ovira: integer; { števec ovir }
  Kje: longint;   { možna lokacija prvega ojačevalnika}
begin
  Kje := 0;
  for Ovira := 1 to StOvir do begin
    Zacetek[Ovira] := Zacetek[Ovira] mod 4000;
    Konec[Ovira] := Konec[Ovira] mod 4000;
    if (Zacetek[Ovira] >= Konec[Ovira]) and (Konec[Ovira] > Kje) then
      Kje := Konec[Ovira];
  end; {for}
  Ovira := 1;
  repeat
    if Zacetek[Ovira] <= Kje then begin
      if Konec[Ovira] <= Zacetek[Ovira] then begin
        Kje := 4000; Ovira := StOvir
      end else if Konec[Ovira] > Kje then begin
        Kje := Konec[Ovira]; Ovira := 0
```

```
      end; {if}
    end; {if}
    Ovira := Ovira + 1;
  until Ovira > StOvir;
  if Kje < 4000 then
    WriteLn('Prvi ojačevalnik naj bo na razdalji ', Kje, ' metrov.')
  else WriteLn('Problem je nerešljiv.');
end. {Postavi}
```

Ta program bi se dalo še izboljšati, da ne bi vsakič, ko popravi trenutni položaj prvega ojačevalnika (spremenljivka Kje), šel pregledovat vseh ovir znova od začetka. Zadostovalo bi že, če bi ovire pred izvajanjem glavne zanke **repeat** uredili po naraščajoči k' (pri tistih s $k' \leq z'$ pa bi si mislili, da imajo $k' = 4000$). Tako bi vedeli, da, ko se Kje recimo pri i-ti oviri poveča, postane enak njenemu k' in s tem $\geq$ od k' vseh prejšnjih ovir, torej mu one zdaj prav gotovo niso v napoto in jih ni treba ponovno pregledovati. Zanka **for** bi ostala nespremenjena, v zanki **repeat** pa bi se znebili stavka Ovira := 0.

R1996.2.3

Naloga: str. 3

Med dvema biseroma iste barve (ali dvema brezbarvnima) ogrlice nima smisla prestriči; lahko prestrižemo eno mesto bolj levo ali desno in še vedno dobimo vse, kar bi dobili tudi prej.

Prav tako nima smisla prestriči med barvnim in brezbarvnim, če je prvi barvni za tistim brezbarvnim iste barve. Na primer: `rbbrm` nima smisla prestriči med prvim in drugim, saj bomo tako pobrali le `rbbr`; če bi prestrigli desno od drugega `r`, bi vse to še vedno dobili, za povrhu pa tudi `m`. Zato se bomo v nadaljevanju tega razmisleka pretvarjali, da so brezbarvni biseri, ki jih na obeh straneh obdajata bisera iste barve, v resnici tudi sami take barve.

Če bi radi prestrigli med rdečim in modrim, pa je vmes še nekaj brezbarvnih, je vseeno, pri katerem prerežemo.

Zato je dovolj, če se pri razmišljanju o tem, kje bi prestrigli, omejimo na primer na takšna mesta: takoj za biserom ene barve, ki mu sledi 0 ali več brezbarvnih in nato vsaj en biser druge barve. Okolico takšnega mesta si lahko predstavljamo takole (barvni biseri so podčrtani):

```
...m bb...b rr...r bb...b mm...m bb...b rr...r bb...b mm...m bb...b r...
            ↑      ↑      ↑      ↑      ↑      ↑      ↑      ↑      ↑
            i      j      i1     j1     i2     j2     i3     j3     i4
```

Če prestrižemo takoj levo od bisera j_1, bomo lahko obdržali vse bisere od j do $i_3 - 1$. Naslednje mesto, o katerem bi razmišljali, je takoj levo od bisera j_2; če bi prestrigli tam, bi obdržali vse od j_1 do $i_4 - 1$, pri čemer je i_4 indeks prvega rdečega bisera desno od i_3. Tako nadaljujemo po ogrlici, dokler ne pridemo naokoli: če je bil j_1 prvi biser, pri katerem smo poskušali prestriči, bomo končali po tistem, ko bomo razmislili še o striženju levo od j.

Podprogram Nasl v spodnji rešitvi poišče, če mu damo indeks nekega bisera i, indeks prvega naslednjega bisera nasprotne barve (i_1) in še indeks (j) prvega brezbarvnega bisera v strnjenem zaporedju brezbarvnih biserov tik pred i_1.

Podprogram KjeRezati začne tako, da poišče prvi barvasti biser (h), vendar pa ta ni nujno na začetku zaporedja biserov svoje barve (lahko je to biser v s[0], pa so še na drugem koncu tabele biseri iste barve). Zato si nato pomagamo s podprogramom Nasl, da poiščemo primeren i. V nadaljevanju se lahko zgodi, da je $i_1 = i$, kar pomeni, da biserov druge barve sploh ni (in bomo ne glede na

mesto reza dobili celo ogrlico), ali pa, da je $i_2 = i$, kar pomeni ogrlico oblike `rrr bbb mmm bbb`, ki jo bomo tudi lahko dobili celo, če le ne prerežemo med biseroma iste barve. Drugače pa poiščemo še j_2 in i_3 in že imamo vse, kar potrebujemo, da ocenimo možnost reza pri j_1. Nato se v zanki pomikamo naprej, dokler ne pridemo spet naokoli: zadnje mesto reza, ki ga moramo oceniti, je prvotna vrednost j (ki si jo podprogram zapomni kot j_0).

Pri delu z indeksi moramo ves čas upoštevati, da je ogrlica krožna. Zaradi lažjega naslavljanja štejemo indekse v nizu od 0 do $n-1$ namesto od 1 do n. Za indeksom i pride $(i+1) \bmod n$, pred njim pa $(i-1) \bmod n$, vendar slednje raje pišimo kot $(i+n-1) \bmod n$, da ne bi imel mod kakšnih težav zaradi negativnega operanda.[3] Če vrne KjeRezati vrednost i, to pomeni rezanje med biseroma s[i − 1] in s[i] (če vrne 0, pa seveda rezanje med s[n − 1] in s[0].

```
const MaxDolzina = ... ;
type Niz = array [0..MaxDolzina − 1] of char;

function KjeRezati(s: Niz; n: integer): integer;

  procedure Nasl(i: integer; var j, i1: integer);
  begin
    i1 := (i + 1) mod n;
    while i1 <> i do
      if (s[i1] <> 'b') and (s[i1] <> s[i]) then break
      else i1 := (i1 + 1) mod n;
    j := (i1 + n − 1) mod n;
    while j <> i do
      if s[j] <> 'b' then break
      else j := (j + n − 1) mod n;
    j := (j + 1) mod n;
  end; {Nasl}

var h, h1, i, j, j0, i1, j1, i2, j2, i3, b, bi, c: integer;
begin
  h := 0;
  while h <> n do      { poiščimo prvi barvni biser }
    if s[h] <> 'b' then break
    else h := h + 1;
  if h = n then        { vse brezbarvno }
    begin KjeRezati := 0; exit end;
  Nasl(h, h1, i); Nasl(i, j, i1);
  if i1 = i then       { nista prisotni in modra in rdeča }
    begin KjeRezati := i; exit end;
  Nasl(i1, j1, i2);
  if i2 = i then       { en kos modre in en kos rdeče }
    begin KjeRezati := i; exit end;
  b := 0; bi := 0; j0 := j;
  repeat
    Nasl(i2, j2, i3);
    c := i3 − j;       { izplen, če prerežemo pred j1 }
    if c <= 0 then c := c + n;
```

[3] V resnici je operator `a mod n` v standardnem pascalu definiran tako, da vedno vrača vrednosti z intervala $0, \ldots, n-1$, vendar se nekateri prevajalniki tega ne držijo in za `a mod n` raje vzamejo vrednost `a − (a div n) * n`, ki je pri negativnem `a` lahko negativna. Za več o različnih definicijah operacij div in mod glej rešitev naloge 1997.2.1.

```
    if c > b then begin b := c; bi := j1 end;
    i := i1; j := j1; i1 := i2; j1 := j2; i2 := i3;
  until j = j0;
  KjeRezati := bi;      { položaj najboljšega izplena }
end;
```

R1996.2.4

Naloga: str. 4

Podprogram Vsiljivec(N) lahko najprej preveri, če že obstaja omejitev za celo N-jevo podmrežo; če obstaja, je treba samo osvežiti njen čas in že lahko končamo. Drugače pa preveri, če že obstaja omejitev za kak drug (od N-ja različen) naslov iz N-jeve podmreže (pri tem omejitve, ki so že prestare, ignorira oz. jih kar sproti pobriše iz seznama). Če najdemo kakšno tako omejitev (zastavica Nasel), moramo uvesti omejitev za celo N-jevo podmrežo, dotedanje omejitve za posamezne naslove iz N-jeve podmreže pa lahko pobrišemo. Če pa ni bilo nobene take omejitve, je mogoče v seznamu vsaj omejitev za sam naslov N; če je, obnovimo njen čas, drugače pa jo dodajmo kot novo omejitev.

Podprogram Sprosti je še preprostejši, saj mora le prečesati vse naslove v seznamu (začne lahko pri $\langle 0, 0\rangle$) in pobrisati tiste, ki so že prestari.

```
procedure Vsiljivec(N: Naslov);
var M: Naslov; t, tM: integer; Nasel: boolean;
begin
  M[1] := N[1]; M[2] := 0; t := Cas;
  { Preverimo, če že obstaja omejitev za N-jevo podmrežo. }
  if Poisci(M) > 0 then begin Obnovi(M, t); exit end;   {⋆}
  { Preverimo, če obstaja omejitev za kak drug naslov iz te podmreže. }
  Nasel := false;
  while Naslednji(M, tM) and not Nasel do begin
    if M[1] <> N[1] then break;
    if (M[2] <> N[2]) then
        if t − tM < TTL then Nasel := true
                        else Brisi(M);
  end; {while}
  if Nasel then begin
    { Nadomestimo omejitve za naslove te podmreže z omejitvijo za celo podmrežo. }
    M[1] := N[1]; M[2] := 0;
    while Naslednji(M, tM) do
      if M[1] <> N[1] then break else Brisi(M);
    M[1] := N[1]; M[2] := 0; Dodaj(M, t);
  end else if Poisci(N) > 0 then Obnovi(N, t)
  else Dodaj(N, t);
end; {Vsiljivec};

procedure Sprosti;
var N: Naslov; t, tN: integer;
begin
  N[1] := 0; N[2] := 0; t := Cas;
  while Naslednji(N, tN) do
    if t − tN > TTL then Brisi(N);
end; {Sprosti}
```

Zgornji podprogram Vsiljivec ima še eno morebitno slabost: na začetku preveri, če že obstaja omejitev za podmrežo, in če obstaja, jo v vsakem primeru

obnovi. Toda ta omejitev je mogoče že prestara (starejša od TTL sekund) — natančneje, prestara je lahko za največ toliko, kolikor časa mine med dvema zaporednima klicema podprograma Sposti. Podprogram Vsiljivec bi na osnovi take zastarele omejitve spet prepovedal dostop celi podmreži namesto le naslovu N. Bolj pošteno bi bilo v takem primeru omejitev za podmrežo pobrisati in uvesti le omejitev za naslov N. Ker podprogrami, ki jih imamo na voljo za delo z omejitvami, ne omogočajo bolj elegantnega načina za ugotavljanje starosti omejitve, si bomo morali pomagati s podprogramom Naslednji. Vrstico {⋆} podprograma Vsiljivec bi morali zamenjati z nečim takšnim:

```
M[1] := N[1] − 1; M[2] := 9999;
if Naslednji(M, tM) then if (M[1] = N[1]) and (M[2] = 0) then begin
  if t − tM < TTL then Obnovi(M, t)
  else begin Brisi(M); Dodaj(N, t) end;
  exit;
end; {if}
M[1] := N[1]; M[2] := 0;
```

REŠITVE NALOG ZA TRETJO SKUPINO

Naloga: str. 5

R1996.3.1

Komunikacija poteka v dveh fazah — prva faza je vzpostavljanje zveze in druga pošiljanje sporočila.

Zveza se vzpostavi tako, da računalnik, ki želi oddati sporočilo, pošlje drugemu računalniku sporočilo „PošiljalBi“. Drugi lahko na to odgovori s „KarDaj!“ (zveza je vzpostavljena, sporočilo se pošlje in izpiše na drugem računalniku) ali z „RajeNe!“ (drugi uporabnik ne želi sprejemati sporočil; računalnik, ki oddaja sporočilo, o tem obvesti svojega uporabnika).

Črni scenarij je naslednji:

računalnik A	računalnik B
uporabnik začne tipkati sporočilo	
	uporabnik začne tipkati sporočilo
uporabnik konča tipkanje računalnik pošlje „PošiljalBi“ računalnik čaka v stavku Sprejmi(s)	
	uporabnik konča tipkanje računalnik pošlje „PošiljalBi“ računalnik čaka v stavku Sprejmi(s)
računalnik sprejme „PošiljalBi“	
	računalnik sprejme „PošiljalBi“

Nobeden od računalnikov ne pošlje sporočila, oba se iz podprograma PosljiSporocilo vrneta v glavni program, uporabnik pa pri tem ni obveščen, da njegovo sporočilo ni bilo poslano.

Izkaže se, da je to pravzaprav edina tovrstna resna težava našega protokola. Izognemo se ji lahko tako, da dovolimo „prepleteno prejemanje“:

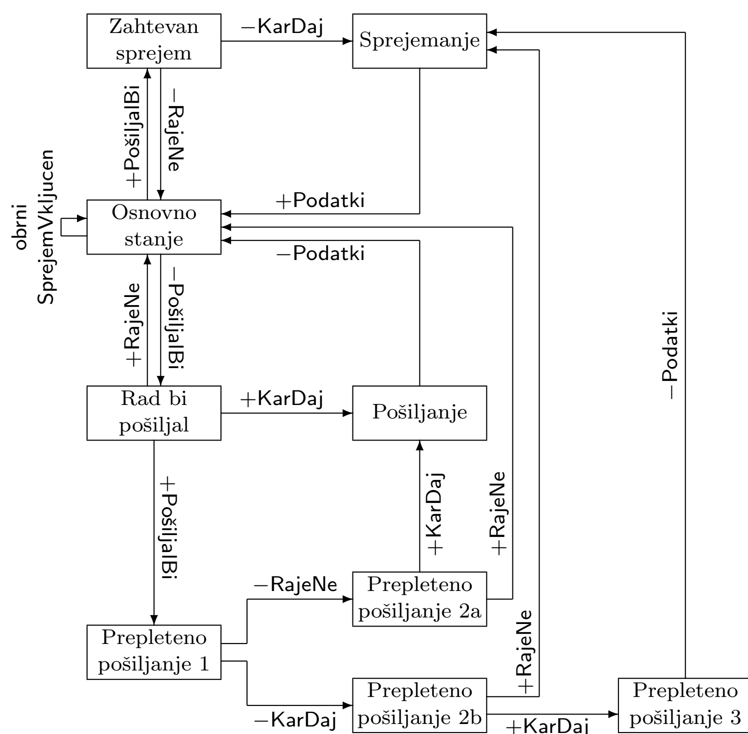
```
procedure PosljiSporocilo(S: string);
var Odg: string;
begin
  Poslji('PošiljalBi');
```

```
    repeat Sprejmi(Odg) until Odg <> '';
    if Odg = 'RajeNe!' then WriteLn('Kolega ima izključen sprejem.')
    else if Odg = 'KarDaj!' then Poslji(S)
    else if Odg = 'PošiljalBi' then begin
      { Prepleteno prejemanje. }
      { Najprej mu povejmo, kaj si mislimo o njegovi zahtevi. }
      if SprejemVkljucen then Poslji('KarDaj!')
      else Poslji('RajeNe!');
      { Nato poglejmo, kaj si misli on o naši. }
      repeat Sprejmi(Odg) until Odg <> '';
      { Naj pošljemo svoje podatke? }
      if Odg = 'KarDaj!' then Poslji(S);
      { Naj čakamo na njegove? }
      if SprejemVkljucen then begin
        repeat Sprejmi(S) until S <> '';
        WriteLn(S);
      end; {if}
    end; {if}
  end; {PosljiSporocilo}
```

Drugih delov programa ni treba spreminjati. Ta rešitev še vedno predpostavlja, da se paketi ne izgubljajo (lahko pa se zakasnijo) in da se njihov vrstni red ne spreminja (prejemnik jih dobi v takem vrstnem redu, v kakršnem jih je pošiljatelj oddal). Izkaže pa se, da prejemnika pri tem protokolu nikoli ne bodo čakala več kot tri sporočila hkrati (npr. če je drugi računalnik pri prepletenem prejemanju zavrnil našo željo po prenosu sporočila, mi pa smo njegovo sprejeli, se lahko zgodi, da se nam v vrsti naberejo njegova sporočila `RajeNe`, uporabnikovi podatki in še (ker je z njegovega vidika prepleteno prejemanje končano in si lahko zaželi pošiljati naslednje sporočilo) `PošiljalBi`; dlje kot to pa ne more iti, ker pošiljatelj zdaj čaka na naš odgovor na ta `PošiljalBi`, dobil pa ga ne bo prej, preden ne bomo mi pobrali vseh treh sporočil iz vrste).

O tem, da je dobljeni protokol zdaj res brez napak, se lahko prepričamo tudi z avtomatskim testiranjem protokola.[4] Program lahko predstavimo z grafom; točkam pravimo stanja, povezave pa so prehodi med njimi (glej sliko na str. 22). Ob vsakem prehodu lahko računalnik pošlje ali sprejme eno sporočilo ter spreminja vrednosti svojih spremenljivk. Stanje celotnega sistema je zdaj sestavljeno iz tega, v katerem stanju grafa se nahaja prvi in v katerem drugi računalnik, kakšne so vrednosti njunih spremenljivk (v našem primeru je pomembna le spremenljivka SprejemVkljucen) in kakšna sporočila so v njunih vrstah (za vsak računalnik imamo medpomnilnik oz. vrsto sporočil, ki mu jih je drugi računalnik že poslal, on jih pa še ni prevzel). Začetno stanje sistema poznamo (programa na začetku izvajanja, prazne vrste ipd.), nato pa lahko za vsako stanje ugotovimo, katera stanja bi mu utegnila slediti (odvisno od tega, kateri računalnik izvede prehod in kakšen prehod naredi). Če omejimo dolžino vrst, je možnih stanj sistema le končno mnogo, zato lahko s takšnim sistematičnim preiskovanjem prej ali slej pridemo do vseh; če pri tem ne opazimo nobenih težav, je z našim protokolom vse v redu. Pri razmeroma preprostem protokolu, kot je naš, možnih stanj sistema običajno sploh ni tako zelo veliko (npr. okoli sto), razen če ni s protokolom kaj narobe in se potem nakopiči ogromno „čudnih“ oz. problematičnih stanj.

[4] Gl. npr. Tone Vidmar, *Računalniška omrežja in storitve*, 1997, razd. 4.3.



Graf stanj, s katerim lahko opišemo naš popravljeni protokol. Prehodi, ki jih sproži prevzem sporočila x, so označeni s „+x“; prehodi, pri katerih odpošljemo sporočilo x, pa so označeni z „−x“. Sporočilo Podatki predstavlja podatke, ki jih je vtipkal uporabnik in jih hoče poslati na drugi računalnik. Če sta iz nekega stanja narisani puščici tako za −RajeNe kot za −KarDaj, to pomeni, da se pri SprejemVkljucen = true izvede vedno prehod −KarDaj, sicer pa −RajeNe.

Naloga: str. 5

R1996.3.2

Podprogram KoncenIzpis nam pomaga ločiti med primeri, ko ima nek program končen izpis, in primeri, ko ima neskončen izpis. Če pa bi mi v resnici radi ločili med tem, da nek program Prog sploh ničesar ne izpiše, in tem, da ima nek izpis (ki je lahko končen ali neskončen), bomo morali Prog oviti v nek nov program (recimo mu Prog2) in to tako, da bo imel Prog2 končen izpis, ko bo imel Prog prazen izpis, in neskončen izpis, ko bo imel Prog neprazen izpis.[5] Potem bo odgovor, ki nam ga bo dal KoncenIzpis, ko mu bomo pokazali program Prog2, povedal ravno to, kar nas zanima: ali ima Prog prazen izpis ali ne. Primer takega programa Prog2 je program, ki v neskončni zanki poganja program Prog vedno na enih in istih vhodnih podatkih: če Prog sploh kaj izpiše, bo to izpisal v vsaki ponovitvi te zanke in izpis programa Prog2 bo zato gotovo neskončen; če pa Prog ne izpiše ničesar, tudi Prog2 ne bo izpisal

[5] Lahko bi zastavili tudi obratno, torej da bi imel Prog2 končen izpis, ko bo imel Prog neprazen izpis, in neskončen izpis, ko bo imel Prog prazen izpis. To bi lahko dosegli tako, da bi Prog2 simuliral delovanje programa Prog (npr. kot interpreter ali emulator) in pri vsakem koraku nekaj izpisal. Če bi Prog v nekem trenutku nekaj izpisal, bi se Prog2 takoj ustavil (v tem primeru bi imel Prog2 očitno končen izpis); če pa bi se Prog ustavil, bi šel Prog2 v neskončno zanko, v kateri bi kar naprej nekaj izpisoval. Tretja možnost je še, da se Prog že sam zacikla in pri tem nikoli nič ne izpiše, to pa tudi že zagotavlja, da bo imel Prog2 v tem primeru neskončen izpis.

ničesar.

```
program Prog2(Prog, Data);
begin
  while true do
    Execute(Prog, Data);
end. {Prog2}
```

Tukaj torej velja:

$$\mathsf{Prog(Data)}\text{ ne izpiše ničesar} \Longleftrightarrow \mathsf{KoncenIzpis(Prog2, [Prog, Data])}$$

Če pa nas zanima, ali se dani program Prog ustavi ali zacikla, moramo pripraviti nek nov program (recimo Prog3), ki bo imel končen izpis v primeru, ko se bo Prog zaciklal, in neskončen izpis v primeru, ko se bo Prog ustavil. To lahko naredimo tako, da Prog poženemo in če oz. ko se ustavi, nekaj izpišemo; to ponavljamo v nedogled. Če se Prog ustavi, bo Prog3 v vsaki ponovitvi te zanke nekaj izpisal in bo imel zato neskončen izpis. Če se Prog zacikla s končnim izpisom, ne bo Prog3 sploh nikoli prišel do tega, da bi tudi sam nekaj izpisal, tako da bo skupni izpis še vedno končen. Nerodna je le tretja možnost, namreč da se Prog zacikla in ima pri tem neskončen izpis; v tem primeru je tudi izpis celotnega programa Prog3 neskončen. Na srečo pa lahko to možnost preverimo neposredno, če programu KoncenIzpis pokažemo program Prog.[6]

```
program Prog3(Prog, Data);
begin
  while true do begin
    Execute(Prog, Data);
    WriteLn('puf, puf');
  end; {while}
end. {Prog3}
```

Program Prog3(Prog, Data) da končen izpis $\Longleftrightarrow$
program Prog(Data) se zacikla in da končen izpis.
Prog(Data) se zacikla $\Longleftrightarrow$ KoncenIzpis(Prog3, [Prog, Data])
or not KoncenIzpis(Prog, Data)

Mimogrede, ni se težko prepričati, da programa KoncenIzpis sploh ni mogoče napisati. Recimo namreč, da bi tak program res imeli. Potem lahko napišemo program:

```
program Prog4(Prog);
begin
  if KoncenIzpis(Prog, [Prog]) then
    while true do WriteLn('tralala');
end. {Prog4}
```

[6] Lahko pripravimo tudi tak Prog3, ki bo imel končen izpis, če se Prog konča, in neskončen izpis, če se Prog zacikla. To bi lahko naredili tako, da bi Prog3 simuliral delovanje programa Prog na danih vhodnih podatkih Data in pri tem po vsakem izvršenem koraku nekaj izpisal. Če se Prog zacikla, bo imel Prog3 tako neskončen izpis, če se Prog ustavi, pa končnega. Zdaj bo odgovor na vprašanje, ali ima Prog3 končen izpis, ravno enak kot na vprašanje, ali se Prog ustavi.

Ali ima Prog4, če mu damo opis samega sebe kot vhodni podatek, končen izpis ali ne? Recimo, da ima končen izpis; torej klic KoncenIzpis v tretji vrstici vrne true; toda v tem primeru pade Prog4 v neskončno zanko in ima neskončen izpis. Torej je ta predpostavka napačna — imeti mora neskončen izpis; toda v tem primeru vrne KoncenIzpis vrednost false in Prog4 se takoj konča, ne da bi sploh kaj izpisal, tako da ima Prog4 končen izpis. Tako smo ugotovili, da ne more imeti Prog4, če mu pokažemo samega sebe kot vhodni podatek, niti končnega niti neskončnega izpisa, to pa je nemogoče, torej tudi ni mogoče, da bi obstajal nek program KoncenIzpis z lastnostmi, kot jih omenja naloga.[7]

Naloga: str. 6

R1996.3.3

Ko preberemo vse omejitve v pomnilnik (množica Omejitve v spodnjem programu), si za vsako spremenljivko S tudi zapomnimo, v katerih omejitvah se pojavlja (množica Kje[S]). V nadaljevanju bomo vzdrževali množico omejitev, ki jih je še treba uskladiti (Neusklajene); na začetku so to kar vse omejitve. Nato v vsakem koraku uskladimo eno od teh omejitev; če se pri tem kakšni od njenih spremenljivk zaloga vrednosti popolnoma izprazni, vemo, da vseh omejitev ne bo mogoče uskladiti, in lahko takoj nehamo. Drugače pa se lahko kakšni od spremenljivk zaloga vrednosti vsaj zmanjša in v tem primeru moramo ponovno pregledati omejitve, v katerih se ta spremenljivka pojavlja (kajti zaradi zmanjšanja njene zaloge vrednosti se zdaj lahko na novo zmanjša zaloga vrednosti še kakšni drugi spremenljivki, ki se pojavlja skupaj z njo v kakšni omejitvi), zato vse te dodamo v množico neusklajenih omejitev. Če se nam množica Neusklajene izprazni, to pomeni, da smo uspeli uskladiti vse omejitve, ne da bi se kakšni spremenljivki zaloga vrednosti zmanjšala.[8]

```
function Usklajenost: boolean;
begin
  Omejitve := {};
  za vsako spremenljivko S: Kje[S] := {};
  while not Eof(Input) do begin
    O := PreberiOmejitev;
    dodaj O v množico Omejitve;
    for i := 1 to ŠteviloSpr(O) do
      dodaj O v množico Kje[ImeSpr(O, i)];
  end; {while}
  Neusklajene := Omejitve;
  while Neusklajene ≠ {} do begin
    vzemi neko O iz množice Neusklajene;
    UskladiOmejitev(O);
    for i := 1 to ŠteviloSpr(O) do begin
      if PraznaZalogaVrednosti(O, i) then return false;
      if ZmanjšanaZalogaVrednosti(O, i) then
        Neusklajene := Neusklajene ∪ Kje[ImeSpr(O, i)];
    end; {for}
  end; {while}
```

[7]Več o stvareh, o katerih smo razmišljali pri tej nalogi, najdemo v učbenikih teorije izračunov (*theory of computation*), npr. M. Sipser: *Introduction to the Theory of Computation*, 1996; J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, 1979.

[8]Opisani postopek za usklajevanje množice omejitev in klestenje zalog vrednosti spremenljivk izvira s področja logičnega programiranja z omejitvami logičnega programiranja z omejitvami (Constraint Logic Programming — CLP). Za več o tem glej nalogo 1995.3.2 in literaturo, navedno v opombi pri njeni rešitvi.

```
  return true;
end; {Usklajenost}
```

V konkretni implementaciji bi množico Omejitve verjetno izvedli s seznamom, v vsakem od elementov tega seznama pa bi bila neka omejitev in še podatek o tem, ali je ta omejitev trenutno v množici Neusklajene ali ne. Množico Kje[S] pa bi lahko predstavili s seznamom, v katerem bi bil vsak element kazalec na enega od elementov seznama Omejitve. Potem se je zelo poceni zapeljati po vseh omejitvah iz Kje[S], za vsako pa takoj vidimo, če je že v množici Neusklajene in če ni, vemo, da jo moramo tja dodati. Množica Neusklajene naj bo tudi seznam kazalcev na elemente seznama Omejitve, tako da lahko, ko vzamemo neko omejitev iz množice Neusklajene, hitro označimo (v seznamu Omejitve), da te omejitve ni več v tej množici.

Konkreten primer programa, ki vsebuje tudi postopek za usklajevanje omejitev, kakršen je bil opisan tule, si lahko ogledamo pri rešitvi naloge 1995.3.2.

R1996.3.4

Naloga: str. 7

Če bi bilo vseh 64 bitov ključa zares naključno izbranih, bi vlomilec potreboval $2^{64}/(1000 \cdot 60 \cdot 60 \cdot 24 \cdot 365) \approx 585$ milijonov let za preizkus vseh ključev. V resnici vseh možnih ključev, ki jih lahko izbere naš program za šifriranje, ni toliko, saj program izbere začetno „naključno“ vrednost le iz trenutnega časa in vseh možnih minut v dnevu je le 1440. Ker funkcija Random vedno predvidljivo iz nekega števila izračuna novo število, 10000-kratno klicanje funkcije Random prav nič ne pripomore k naključnosti, saj vlomilec pozna naš program in lahko enak postopek ponovi. Prav tako zlaganje 64-bitnega števila iz štirih psevdonaključnih, ki so izpeljana eno iz drugega, nič ne pripomore k povečanju števila možnih ključev, saj lahko vlomilec isti postopek ponovi in dobi isti rezultat.

Kljub temu torej, da je na prvi pogled videti 64 bitov ključa precej naključnih, je vseh možnih različnih ključev, ki jih lahko izbere naš program, le 1440. Za preizkus vseh teh ključev bi vlomilec potreboval $1440/1000 = 1{,}44$ sekunde, pričakovani čas pa je polovico tega: 0,72 sekunde. Če vlomilec ve, kdaj približno smo besedilo šifrirali, pa se pričakovani čas, potreben za ugibanje, še zmanjša.

Prvo izboljšavo bi lahko naredili pri izbiri začetnega „naključnega“ števila. Tega bi morali dobiti iz opazovanja zunanjih dogodkov, ki niso tako predvidljivi kot ura, npr. poslušanja šuma iz avdio kartice, merjenja časov med pritiski na tipke pri tipkanju nekega daljšega besedila, merjenju časov med prekinitvami (interrupts) diskovnega vmesnika med večjo aktivnostjo diska in podobno — najbolje kar iz več virov.[9] Če bi tako izbrali prvo 16-bitno število, ostala štiri pa še vedno izračunali eno iz drugega, bi bilo vseh možnih ključev $2^{16} = 65536$ in za preizkus potrebnih 65,5 sekund, kar ni poseben napredek. Torej se je treba v celoti odpovedati uporabi generatorja psevdonaključnih števil in z opisanim postopkom zbiranja šuma iz računalnikovega okolja zbrati vseh 64 bitov ključa.

Pripomnimo še, da veljajo podobna časovna razmerja tudi v primeru, če vlomilec našega programa ne bi poznal, le več truda in računalniškega časa bi moral žrtvovati, pomagalo pa bi tudi, če bi uspel prestreči več šifriranih sporočil. Temelj sodobnega šifriranja ni v tajnosti šifrirnih algoritmov, ampak v tajnosti in neuganljivosti ključev.

[9] Glej tudi 4. nalogo za prvo skupino (str. 2).

REŠITEV NALOGE DRUGEGA ZAKLJUČNEGA TEKMOVANJA IZ ZNANJA RAČUNALNIŠTVA

Naloga: str. 9

R1996.Z

Za začetek je pametno podatke o karticah v celoti prebrati v glavni pomnilnik, saj ga je dovolj. Če bi predelali kar naivno rešitev, bi morali za vsako kartico hraniti množico tipa **set of** 1..100, ki bi zasedala v pomnilniku $\lceil 100/8 \rceil = 13$ bajtov, tako da bi vseh 100 000 kartic porabilo slabega 1,3 MB (spomnimo se, da imamo na voljo skoraj 16 MB pomnilnika). S tem bi prihranili že veliko časa, ker nam ne bi bilo treba po vsakem izžrebanem številu brati podatkov o karticah z diska. Poleg tega lahko to naredimo v času pred prvim žrebom, tako da se nam sploh ne bo štelo v čas, ki ga meri žirija.

Naivna rešitev primerja vsebino vsake kartice kar z množico doslej izžrebanih števil; če je razlika prazna množica, vemo, da je bila kartica izžrebana. (Obenem še preveri, da je številka, ki smo jo nazadnje izžrebali, napisana na kartici, kajti nas zanimajo le kartice, ki so se dopolnile šele ob zadnji izžrebani številki.) Naš program torej kar naprej (po vsaki izžrebani številki) računa razlike Karta − Izzrebane in si daje pri tem vsakič opraviti z vsemi stotimi biti (oz. vsemi trinajstimi bajti) v teh dveh spremenljivkah, čeprav se bo rezultat v primerjavi s tistim pred zadnjim žrebanjem razlikoval le po tem, da v njem ne bo več pravkar izžrebane karte. Stavek

```
if (NovaSt in Karta) and (Karta − Izzrebane = []) then
   Zadetkov := Zadetkov + 1;
```

bi torej lahko zamenjali z

```
var Karta: array [1..StKartic] of set of 1..100;
{ ... }
if NovaSt in Karta[i] then begin
   Exclude(Karta[i], NovaSt);
   if Karta[i] = [] then Zadetkov := Zadetkov + 1;
end; {if}
```

To nam že prihrani nekaj časa, ker Exclude le izključi bit, ki predstavlja element, ki ga brišemo iz množice (torej ne gre skozi vseh sto bitov). Preverjanje, če je Karta[i] po novem prazna množica, pa je še vedno tako zahtevno kot prej, ker mora program za vseh trinajst bajtov, ki jih ta spremenljivka zaseda v pomnilniku, preveriti, če v njih res ni noben bit prižgan. Zato je koristno nekje za vsako kartico hraniti podatek o tem, koliko številk z nje je še neizžrebanih. Vsakič, ko zbrišemo neko število iz množice, bi ta števec zmanjšali in nato zelo poceni preverili, če je po novem enak 0. Na začetku bi bil ta števec enak 15 za vse kartice.

Še vedno pa po nepotrebnem tratimo čas, ko po vsakem izžrebanem številu pregledujemo *vse* kartice in za vsako preverjamo, če mogoče vsebuje pravkar izžrebano število; saj je vendar že vnaprej jasno, da nastopa posamezno število v povprečju le na $100\,000 \cdot 15/100 = 15\,000$ karticah, torej ostalih 85 000 pregledujemo brez koristi. Zato si bomo raje za vsako številko (od 1 do 100) zapomnili, na katerih karticah se pojavlja; potem se bomo morali po vsakem žrebu ubadati le s tistimi karticami, ki to številko zares vsebujejo. Zdaj pa lahko tudi ugotovimo, da množice Karta[i], ki naj bi za vsako karto povedala, katere številke so na njej še neizžrebane, sploh ne potrebujemo več (doslej smo jo namreč uporabljali le za preverjanje, ali neka kartica vsebuje pravkar izžrebano številko ali ne,

po novem pa nam tega ne bo treba početi, saj bomo imeli za vsako številko seznam kartic, na katerih se pojavlja).

Kartice bi lahko preprosto oštevilčili (recimo od 0 do 99 999) in seznam kartic, na katerih se pojavlja določena številka, predstavili kar s tabelo 32-bitnih številk kartic. Vendar pa bi bila v tem primeru ta tabela lahko večja od 64 KB, saj nam nihče ne zagotavlja, da se ne pojavlja neka številka na zelo veliko karticah. Spodnji program se skuša temu izogniti tako, da kartice v mislih razdeli na skupine po tisoč kartic (odvisno od tega, koliko je res vseh kartic, imamo potem eno, deset ali pa sto takšnih skupin). Dodatna prednost tega je, da so zdaj številke kartic vedno od 0 do 999, tako da nam zanje zadostujejo 16-bitna števila.

Najprej zapišimo program za pripravo indeksnih datotek, v kateri bo za vsako številko pisalo, na katerih karticah se pojavlja. Naloga pravi, da si lahko pripravimo za največ 20 MB datotek, kar je za nas več kot dovolj: 111 000 kartic s po 15 številkami in še knjigovodski podatki (na koliko karticah se pojavlja vsaka številka; to je 111-krat po sto števil); ker uporabljamo 16-bitna števila, bo dovolj že 3 352 200 bytov. Dobljene indeksne datoteke bo kasneje uporabljal naš glavni program Tombola.

```
program PripraviIndeks;
type
  { V tabeli tipa T1000Words bo element 0 hranil število kartic
    (iz neke skupine 1000 kartic), na katerih je neka številka,
    naslednjih toliko elementov pa vsebuje indekse teh kartic. }
  P1000Words = ↑T1000Words;
  T1000Words = array [0..1000] of word;
var
  fIn: text;
  fOut: file;
  Buf: array [0..16383] of byte; { za hitrejše branje datoteke s karticami }
  Kartice: array [1..100] of P1000Words; { kartice trenutne skupine }
  i, j, k, N, StSkupin: integer;
begin
  if      ParamStr(1) = '1000' then StSkupin := 1
  else if ParamStr(1) = '10000' then StSkupin := 10
  else if ParamStr(1) = '100000' then StSkupin := 100
  else    Halt;
  Assign(fIn, 'k' + ParamStr(1) + '.txt');
  Reset(fIn); SetTextBuf(fIn, Buf, SizeOf(Buf));
  Assign(fOut, 'k' + ParamStr(1) + '.idx');
  Rewrite(fOut, 1);
  for i := 1 to 100 do New(Kartice[i]);
  for i := 0 to StSkupin − 1 do begin
    { Obdelajmo naslednjih tisoč kartic. }
    for j := 1 to 100 do Kartice[j]↑[0] := 0;
    for j := 0 to 999 do begin
      for k := 1 to 15 do begin
        Read(fIn, N);
        Kartice[N]↑[0] := Kartice[N]↑[0] + 1;
        Kartice[N]↑[Kartice[N]↑[0]] := j;
      end;
      ReadLn(fIn);
    end; {for j}
```

```
    { Shranimo podatke o teh tisoč karticah. }
    for j := 1 to 100 do BlockWrite(fOut,
      Kartice[j]↑, SizeOf(word) * (Kartice[j]↑[0] + 1));
  end; {for i}
  Close(fIn); Close(fOut);
  for i := 1 to 100 do Dispose(Kartice[i]);
end. {PripraviIndeks}
```

Tu pa je še program Tombola:

```
program Tombola;

uses Zirija;

type
  P1000Words = ↑T1000Words;
  T1000Words = array [0..1000] of word;
  P1000Bytes = ↑T1000Bytes;
  T1000Bytes = array [0..999] of byte;
var
  i, j, StSkupin, NovaSt: integer; W: word; N: byte;
  NStevil: array [0..99] of P1000Bytes; { št. neizžrebanih števil na karticah }
  { Katere kartice vsebujejo določeno število? Element Kartice[X, Y]↑[Z] = Q
    nam pove, da kartica št. 1000 * Y + Q vsebuje številko X.
    Pri tem lahko Z zavzame vrednosti od 1 do Kartice[X, Y]↑[0]. }
  Kartice: array [1..100, 0..99] of P1000Words;
  fIdx: file;
  Zadetkov: longint; P: P1000Words; PN: P1000Bytes;
begin
  Pripravi;
  { Ugotovi število kartic. }
  if      ParamStr(1) = '1000' then StSkupin := 1
  else if ParamStr(1) = '10000' then StSkupin := 10
  else if ParamStr(1) = '100000' then StSkupin := 100
  else    Halt;
  { Preberi podatke o karticah. }
  Assign(fIdx, 'k' + ParamStr(1) + '.idx');
  Reset(fIdx, 1);
  for i := 0 to StSkupin − 1 do begin
    for j := 1 to 100 do begin
      BlockRead(fIdx, W, SizeOf(W)); { Število kartic, na katerih je številka j. }
      GetMem(Kartice[j, i], SizeOf(Word) * (W + 1));
      Kartice[j, i]↑[0] := W;
      BlockRead(fIdx, Kartice[j, i]↑[1], W * SizeOf(Word));
    end; {for j}
    New(NStevil[i]);
    { Na vseh karticah je še 15 neizžrebanih števil. }
    for j := 0 to 999 do NStevil[i]↑[j] := 15;
  end; {for i}
  Close(fIdx);
  { Žrebaj. }
  while Zrebaj(NovaSt) do begin
    Zadetkov := 0;
    for i := 0 to StSkupin − 1 do begin
      P := Kartice[NovaSt, i]; PN := NStevil[i];
```

```
      for j := 1 to P↑[0] do begin
        W := P↑[j]; { Številka kartice. }
        N := PN↑[W] − 1; { Novo št. še neizžrebanih številk na tej kartici. }
        PN↑[W] := N;
        if (N = 0) then Zadetkov := Zadetkov + 1;
      end; {for j};
    end; {for i};
    Zapisi(Zadetkov);
  end; {while}
  { Pospravi. }
  for i := 0 to StSkupin − 1 do begin
    for j := 1 to 100 do
      FreeMem(Kartice[j, i], SizeOf(Word) * (Kartice[j, i]↑[0] + 1));
    Dispose(NStevil[i]);
  end; {for}
  Pospravi;
end. {Tombola}
```