12. republiško tekmovanje v znanju računalništva (1988)

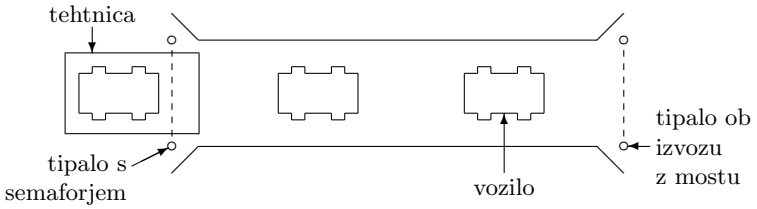
	Naloge	Rešitve
I	1 2 3 4	1 2 3 4
II	1 2 3 4	1 2 3 4
III	1 2 3 4	1 2 3 4

NALOGE ZA PRVO SKUPINO

1988.1.1

Rešitev: str. 6

Dolg ozek most, po katerem teče promet le v eno smer, zdrži skupno obremenitev 20 ton. Na mostu je lahko hkrati več vozil, vendar prehitevanje ni možno. Tik pred mostom je tehtnica, ki meri težo naslednjega vozila, ki bo zapeljalo na most. Ob tehtnici je tudi semafor, s katerim lahko prepovemo ali dovolimo, da bi vozilo s tehtnice zapeljalo na most. Na obeh straneh mostu sta tipali, ki povesta, kdaj kakšno vozilo zapelje z mostu oziroma s tehtnice nanj. Predpostaviš lahko, da je most na začetku prazen.



Za upravljanje semaforja želimo uporabiti računalnik. **Napiši algoritem**, po katerem bo računalnik upravljal semafor. **Opiši podatkovno strukturo**, ki jo potrebujemo za opis stanja na mostu.

1988.1.2

Rešitev: str. 7

Napiši program, ki prebere vrstico z nekaj besedami, ki so ločene z enim ali več presledki, in besede izpiše v obratnem vrstnem redu.[1] Primer:

```
Prijatlji! odrodile so trte vince nam sladkó
```

naj program prepiše v

```
sladkó nam vince trte so odrodile Prijatlji!
```

1988.1.3

Rešitev: str. 9

Iz majhnega računalnika, številčnega zaslona in tipke želimo sestaviti štoparico. Na zaslonu je prostor za prikaz ur, minut, sekund in stotink sekunde. S tipko upravljamo delovanje štoparice takole:

- dolg pritisk na tipko (daljši od pol sekunde) postavi čas na 0 in ustavi štoparico, ne glede na to, ali je prej tekla ali ni;
- krajši pritisk na tipko požene ustavljeno ali pa ustavi tekočo štoparico.

Napiši program, ki bo krmilil štoparico. Na razpolago imaš naslednje podprograme:

function Tipka vrne true, če je tipka pritisnjena, sicer vrne false;

function Impulz vrne true, če je bila od prejšnjega klica te funkcije dopolnjena nova stotinka sekunde;

[1] Podobna naloga je tudi 2000.U.1.

procedure Zaslon(h, m, s, cs) zapiše novo vrednost na zaslon; h, m, s, cs so cela števila — ure, minute, sekunde in stotinke sekunde.

Predpostaviš lahko, da je računalnik dovolj hiter, da ob rednem klicanju podprograma Impulz ne spregleda nobene stotinke sekunde.

Rešitev: str. 9

1988.1.4 Imamo naslednji program:

```
program Naloga(Input, Output);
var a, d, k, p, q: integer;
begin
  d := 0; ReadLn(a);
  for k := 2 to a do begin
    p := 2; q := k div p;
    while p < q do begin p := p + 1; q := k div p end;
    if (p = q) and ((k mod p) = 0) then d := d + 1;
  end; {for}
  WriteLn(d);
end. {Naloga}
```

(a) Kaj navedeni program izpiše, če za a podamo vrednost 99? Kaj, če podamo 100?

(b) Kaj navedeni program dela? Utemelji!

NALOGE ZA DRUGO SKUPINO

Rešitev: str. 10

1988.2.1 Datoteka na disku je organizirana kot seznam blokov, ki so poljubno razmetani po disku. Zato vsak blok poleg vsebine datoteke vsebuje še naslednja podatka:

- naslov naslednjega bloka datoteke na disku; ter
- naslov bloka z dvakratno zaporedno številko.

Do nekega bloka datoteke lahko tako pridemo po več poteh. Primer:

$$1 \to 2 \to 3 \to 4 \to \cdots \to 16 \to 17 \text{ (zaporedoma) ali pa}$$
$$1 \to 2 \to 3 \to 6 \to 7 \to 8 \to 16 \to 17$$

in še kako drugače.

Vsaka puščica seveda zahteva dostop na disk in branje bloka podatkov. **Napiši algoritem**, ki za dano številko bloka poišče najkrajšo možno pot od začetka datoteke do zahtevanega bloka v njej! **Kako** bi to pot čim krajše opisal? **Najmanj koliko** bitov je potrebnih za opis poti do blokov s številkami, manjšimi ali enakimi 2^n?

Rešitev: str. 12

1988.2.2

Uporabniki nekega programa so se pritožili, da so ukazne besede predolge. Zato želimo vpeljati možnost okrajševanja ukazov. Ukaz `dodaj` naj bi bilo na primer mogoče okrajšati na `dod`, `doda` ali `dodaj`; ne pa na `do` (prekratka okrajšava) ali `dodamo` (napačni znaki v ukazu). Okrajšavo opišemo tako, da z zvezdico v modelu ukazne besede označimo, do kam sega obvezni del ukaza; v našem primeru `dod*aj`. Okrajšani ukaz mora biti dolg najmanj en znak (v modelu ukazne besede zvezdica nikoli ne stoji na prvem mestu).

Napiši podprogram, ki ugotovi, ali nek niz ustreza na zgornji način podanemu modelu ukazne besede. Model ukazne besede in preverjani niz sta v ustreznih tabelah znakov in ju zaključuje vsaj en presledek.

Rešitev: str. 12

1988.2.3

Dve ločeni osi, med katerima je sklopka, se vrtita z različnima hitrostma v isto smer. Pri tem lahko krmilimo prvo os s pomočjo podprograma:

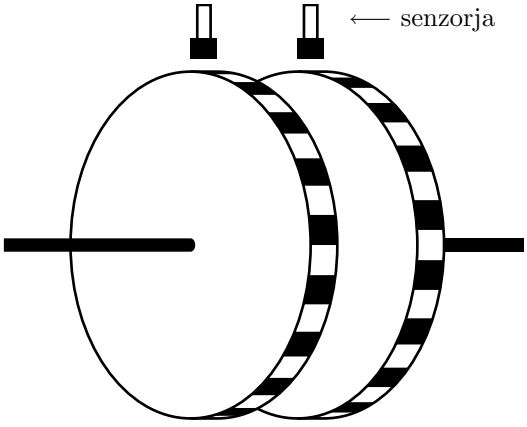
```
procedure Pogon(Pospesek: real);
```

ki določa pospešek (Pospesek > 0) oziroma pojemek (Pospesek < 0) vrtenja. Upoštevaj, da motor, ki ga upravlja program, ne zmore večjega pospeška kot $+1$ in ne pojemka, večjega od -1. Hitrost vrtenja druge osi, ki se lahko počasi spreminja, lahko le opazujemo, nanjo pa ne moremo vplivati.

Vsaka os je po obodu označena z enakomerno debelimi črnimi in belimi pasovi. Za odčitavanje oznak imamo na voljo funkcijo:

```
function Oznaka(Os: integer): boolean;
```

ki pove za zahtevano os (število 1 ali 2), ali je ob senzorju črn pas ali ne (glej sliko).



Za svoje delo imamo na voljo še funkcijo

```
function Cas: integer;
```

ki vrne trenutni čas v milisekundah od zagona programa; pri tem zanemarimo možnost prekoračitve obsega celih števil.

Napiši program, ki bo krmilil vrtenje prve osi tako, da se bo vrtela enako hitro kot druga os in bo na koncu s pomočjo podprograma Sklopi spojil obe

osi, da se bosta vrteli skupaj. Osi je dovoljeno sklopiti šele pri dovolj majhni razliki hitrosti vrtenja obeh osi. Pri tem upoštevaj, da je računalnik zelo hiter v primerjavi z vrtenjem osi ter da so spremembe hitrosti majhne v primerjavi s hitrostjo vrtenja.

Rešitev: str. 14

1988.2.4

Imamo naslednji program:

```
program Naloga(Output);
const
  mCif = 4;
  mStv = 9999;
  pStv = 2;
var
  cif, stv, rbo, obr, vst: integer;
begin
  vst := 0;
  for stv := pStv to mStv do begin
    rbo := stv; obr := 0;
    for cif := 1 to mCif do
      begin obr := 10 * obr + (rbo mod 10); rbo := rbo div 10 end;
    vst := vst + (stv - obr);
  end; {for}
  WriteLn(vst);
end. {Naloga}
```

(a) **Kaj izpiše** podani program? Kaj bi program izpisal, če bi bila konstanta pStv enaka 1 namesto 2?

(b) **Kaj dela** podani program? Utemelji!

NALOGE ZA TRETJO SKUPINO

Rešitev: str. 14

1988.3.1

Na voljo imamo računalniški sistem z 10 enakimi procesorji. Vsak procesor lahko uporablja skupen pomnilnik na varen način, ne da bi ga pri tem motili ostali procesorji. **Napiši algoritem**, ki bo na tem sistemu kar najhitreje poiskal vsa praštevila med 1 in 100000! **Koliko hitreje** se izvede tvoj program (na vsakem procesorju teče svoja kopija) na tem sistemu kot na enoprocesorskem sistemu? Za usklajevanje dela med procesorji imaš na razpolago naslednje podprograme:

MojaOznaka(Oznaka) vrne v spremenljivki Oznaka številko procesorja (med 0 in 9), na katerem teče program;

Zaseden označi, da je procesor, ki izvaja ta klic, zaseden;

Prost označi, da je procesor, ki izvaja ta klic, prost;

VsiProsti je funkcijski podprogram, ki vrne vrednost true, ko so vsi procesorji prosti (nobeden še ni s klicem podprograma Zaseden označil, da je zaseden, ali pa so vsi, ki so bili zasedeni, s klicem podprograma Prost označili, da so prosti); če je vsaj en procesor zaseden, je vrednost funkcije false. Ob zagonu sistema ima funkcija vrednost true.

VsiZasedeni je funkcijski podprogram, ki vrne vrednost true, ko so vsi procesorji zasedeni (vsak je že vsaj enkrat klical Zaseden in po svojem zadnjem klicu podprograma Zaseden še ni poklical podprograma Prost), sicer pa vrne False.[2]

1988.3.2

Sestavi program, ki prešteje, kolikokrat se v nekem nizu znakov pojavi nek drug niz. Pri tem ni nujno, da znaki drugega niza v prvem stoje zaporedoma, ujemati se mora le vrstni red.

Rešitev: str. 18

```
Primer:  MATI   v  MATEMATIKA     ali     SENO   v  SOSEDNOST
     1           MAT    I                        S  E NO
     2           MA    TI                          SE NO
     3           M    ATI
     4               MATI
```

1988.3.3

Na laserski gramofonski plošči je glasbena informacija zapisana v digitalni obliki tako, da je za vsako sekundo podanih 44100 vzorcev zvočne amplitude (vzorec je celo število med 0 in 65535). Na ta način je možno digitalizirati poljuben zvok iz človekovega slušnega območja. Pri zapisovanju in kasnejšem branju lahko pride do napak v zapisu, ki jih mora ustrezno vezje čimbolje zakriti (seveda čudeži niso možni, če je izgubljene preveč informacije). To dela takole:

Rešitev: str. 19

(a) Če je poškodovanih do največ 9 zaporednih vzorcev, jih nadomesti z navadno linearno funkcijo od zadnjega pravilnega vzorca pred napako do prvega pravilnega po napaki.

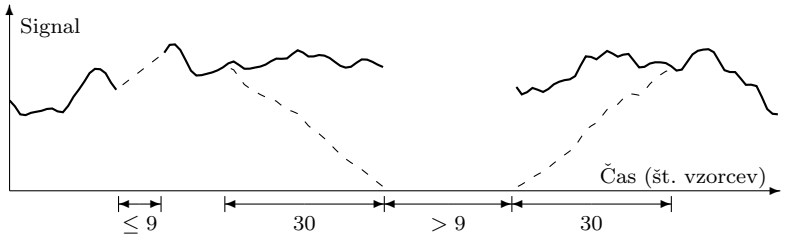
(b) Če je poškodovanih več kot 9 zaporednih vzorcev, se ti nadomestijo z ničlami, razen tega se zadnjih trideset vzorcev pred začetkom napak linearno utiša in prvih trideset po napaki primerno ojača.

Napiši program, ki je skrit v opisanem vezju. Na voljo imaš podprograma:

DobiVzorec(Vzorec, Pravilen), ki dobi s plošče vzorec in podatek o pravilnosti vzorca (je/ni pravilen);

PosljiVzorec(Vzorec), ki pošlje vzorec digitalno/analognemu pretvorniku.

Predpostavi, da je delovanje programa dovolj hitro za sprotno branje in obdelavo vzorcev.



[2] V besedilu te naloge, kakršno se nam je ohranilo v biltenu tekmovanja za leto 1988, funkcije VsiZasedeni ni (pač pa le MojaOznaka, Zaseden, Prost in VsiProsti). Izkaže se, da si je samo s temi ostalimi funkcijami težko pomagati pri sinhronizaciji naših paralelno delujočih procesorjev (tudi rešitev v biltenu za leto 1988 ima pri sinhronizaciji napake). Zato smo zdaj dodali še funkcijo VsiZasedeni, bralec pa lahko poskusi razmisliti tudi o tem, kako bi skrbel za sinhronizacijo, če ne bi imel na voljo tu opisanih funkcij (razen MojaOznaka).

Rešitev: str. 20

1988.3.4

Pri prenosu sporočil prek javnih sredstev (pošta, telegram, elektronska pošta itd.) želimo zagotoviti tajnost. To storimo tako, da pošiljatelj sporočilo zakodira (predela svoje besedilo x v drugo besedilo $y = f(x)$, pri tem pa funkciji f pravimo *kodirna funkcija*), prejemnik pa nad zakodiranim besedilom y uporabi obratno funkcijo g (*razkodirna funkcija*), ki predela prejeto besedilo y v prvotno besedilo $x = g(y) = g(f(x))$.

Da ima skupina ljudi zagotovljeno tajnost sporočil pri njihovi medsebojni izmenjavi, vsakemu človeku določimo njemu lasten par funkcij f in g, za kateri velja:

$$f(g(x)) = g(f(x)) = x.$$

Funkciji f in g morata biti seveda izbrani tako, da je kljub poznavanju ene od funkcij ter nekaj primerkov nekodiranega in zakodiranega besedila v sprejemljivem času nemogoče izračunati njej obratno funkcijo.

Vsakdo eno od svojih funkcij zadrži kot svojo skrivnost (denimo f), drugo pa (denimo g) uvrsti v javni kodirni imenik. Če torej želi Aleš poslati sporočilo Bojanu, poišče v kodirnem imeniku Bojanovo javno kodirno funkcijo in z njo zakodira sporočilo. Razkodira ga lahko le Bojan s svojo skrivno funkcijo.

Javnost kodirnih funkcij odpira nov problem: vsakdo lahko napiše sporočilo Bojanu in se v njem izdaja za Aleša (vsakdo lahko zakodira sporočilo z Bojanovo javno kodirno funkcijo ter mu ga pošlje). Ta se ne more na noben zanesljiv način prepričati, da sporočilo, zakodirano po zgornjem postopku, res prihaja od Aleša in ne od koga drugega, ki se šali na račun obeh.

Ali lahko samo z uporabo opisanih parov kodirnih funkcij posameznih članov skupine (po ene javne in ene skrivne) Aleš sporočilo zakodira tako, da bo Bojan povsem prepričan, da sporočilo prihaja zares od njega? **Razloži** rešitev!

REŠITVE NALOG ZA PRVO SKUPINO

Naloga: str. 1

R1988.1.1

Ker je tehtnica postavljena le pri vstopu na most, si moramo težo vozila zapomniti, da jo bomo lahko, ko bo vozilo odpeljalo z mostu, odšteli od skupne obremenitve mostu. Ker je na mostu lahko hkrati več vozil, med seboj pa se ne morejo prehitevati, bo primerna podatkovna struktura vrsta, katere elementi so teže posameznih vozil. Vrsto lahko realiziramo s tabelo, indeksoma najstarejšega in najnovejšega elementa v njej in s spremenljivko, ki hrani trenutno število elementov v vrsti. Če pa nimamo nobene pametne zgornje meje za število vozil na mostu in zato ne vemo, kolikšno tabelo pripraviti, bi lahko vrsto realizirali tudi s seznamom, v katerem so posamezni elementi povezani s kazalci.

```
SkupnaTeza := 0; IzprazniVrsto;
while true do begin
  Tehtaj(TezaVozila); { odčitamo težo na tehtnici; lahko je tudi 0 }
  if SkupnaTeza + TezaVozila <= DovoljenaObremenitev
    then prižgi zeleno luč else prižgi rdečo luč;
  if vozilo zapeljalo na most then begin
    SkupnaTeza := SkupnaTeza + TezaVozila
    DodajVVrsto(TezaVozila);
  end;
  if vozilo zapeljalo z mostu then begin
```

```
      TezaVozila := najstarejši element v vrsti; SkrajsajVrsto;
      SkupnaTeza := SkupnaTeza − TezaVozila;
    end; {if}
end; {while}
```

Ta rešitev predpostavlja, da tehtnica v trenutku, ko tipalo na mostu zazna vozilo, še vedno odčituje celotno težo vozila (tako kaže tudi slika pri besedilu naloge). Predpostavlja tudi, da tipali sporočita mimovozeče vozilo le trenutno, torej je za vsako vozilo odčitana vrednost true le ob prvem odčitavanju, ne pa ob vsakokratnem odčitavanju, dokler se vozilo pelje mimo tipala.

R1988.1.2

Naloga: str. 1

Ko beremo niz, si lahko v neki globalni spremenljivki (vBesedi v spodnjem programu) zapomnimo, ali smo trenutno znotraj besede ali ne. Tako lahko opazimo začetek besede (če doslej nismo bili v besedi, trenutni znak pa je nekaj drugega kot presledek). Začetke besed si zapomnimo v neki tabeli (Zacetek). Potem se lahko z zanko **for** sprehodimo po tej tabeli od konca proti začetku in tako izpišemo obrnjeno zaporedje besed.

```
program ObrniVrstniRed(Input, Output);
const
  mVrsta = 200;          { največje število znakov v vrsti }
  mBesed = 100;          { največje možno število besed v vrsti }
type
  VrstaT = array [1..mVrsta + 1] of char;
  ZacetekT = array [1..mBesed] of integer;
var
  Zacetek: ZacetekT;     { indeksi začetkov besed }
  BesL: integer;         { število shranjenih začetkov besed }
  Vrsta: VrstaT;         { vrstica z besedami }
  VrstaL: integer;       { dolžina vrstice z besedami }
  vBesedi: boolean;      { pove, ali smo pri branju v besedi }
  Znak: char;
  j, Bes: integer;
begin
  VrstaL := 0; BesL := 0; vBesedi := false;
  { Preberemo vrstico z besedami in shranimo indekse začetkov besed. }
  while not Eoln do begin
    Read(Znak); VrstaL := VrstaL + 1; Vrsta[VrstaL] := Znak;
    if Znak = ' ' then vBesedi := false
    else if not vBesedi then
      begin vBesedi := true; BesL := BesL + 1; Zacetek[BesL] := VrstaL end;
  end; {while}
  { Na konec vrste dodamo presledek za stražarja. }
  Vrsta[VrstaL + 1] := ' ';
  { Izpišemo besede v obratnem vrstnem redu. }
  for Bes := BesL downto 1 do begin
    j := Zacetek[Bes]; if Bes < BesL then Write(' ');    {⋆}
    while Vrsta[j] <> ' ' do begin Write(Vrsta[j]); j := j + 1 end;
  end; {for}
  WriteLn;
end. {ObrniVrstniRed}
```

Da bi lahko gornji program varno uporabljali v praksi, bi mu morali dodati še preverjanje prekoračitve največje dolžine vrstice in največjega števila besed v vrstici.

Gornji program med dvema zaporednima besedama vedno napiše po en presledek, čeprav je bilo v vhodnih podatkih tam mogoče več zaporednih presledkov. Če bi hoteli ohranjati vse presledke, bi lahko vrstico {⋆} zamenjali z nečim takšnim:

```
if Bes < BesL then begin { Izpišemo presledke med besedama Bes in Bes + 1. }
  j := Zacetek[Bes + 1]; while Vrsta[j − 1] = ' ' do j := j − 1;
  while Vrsta[j] = ' ' do begin Write(Vrsta[j]); j := j + 1 end;
end; {if}
j := Zacetek[Bes];
```

Pa še primer rešitve v pythonu, ki kaže, da lahko problem včasih rešimo lažje, če se ga lotimo s primernim orodjem:

```
import sys
besede = sys.stdin.readline().split()  # preberemo vrstico in jo razbijemo na besede
besede.reverse()                       # obrnemo seznam besed
print " ".join(besede)  # staknemo jih skupaj (vmes postavimo presledke) in izpišemo
```

Šlo bi celo z enim samim stavkom, čeprav to verjetno že ni več primer lepega programiranja:

```
print " ".join((lambda x: x.reverse() or x)(__import__("sys").stdin.readline().split()))
```

Operator **or** v pythonu izračuna najprej svoj levi operand; če je njegova vrednost logično resnična, jo vrne, sicer pa izračuna in vrne vrednost desnega operanda. V gornjem primeru `x.reverse()` ne vrne ničesar (bo pa obrnil seznam `x`), zato je učinek izraza `x.reverse() or x` ta, da obrne seznam `x` in vrne referenco nanj. Funkcija `__import__` naloži zahtevani modul in vrne referenco nanj. Izraz **lambda** `x: tralala` pa vrne funkcijo, ki `x` preslika v `tralala`. Gornji stavek torej skonstruira funkcijo, ki obrne dani seznam, in jo kar takoj pokliče na seznamu besed iz prve vrstice standardnega vhoda; besede v obrnjenem vrstnem redu potem stakne (vmes postavi presledke) in izpiše.

Uvedba „extended slices“ v Pythonu 2.3 omogoča še za odtenek krajšo rešitev:

```
print " ".join(__import__("sys").stdin.readline().split()[::−1])
```

Izraz `x[a:b:k]` sestavi (če malo poenostavimo) seznam, v katerega uvrsti elemente seznama `x` z indeksov od `a` do `b` s korakom `k`. Če vrednosti `a` in `b` izpustimo, to pomeni, naj se predela cel seznam. Korak je lahko tudi negativen, v našem primeru -1; zato z `x[::−1]` dobimo seznam, ki vsebuje vse elemente seznama `x`, le da v obratnem vrstnem redu.

Od Pythona 2.4 naprej pa lahko za obračanje seznamov (in drugih zaporedij) uporabljamo tudi standardni podprogram `reversed`:

```
print " ".join(reversed(__import__("sys").stdin.readline().split()))
```

Majhna slabost te in prejšnje rešitve (z „`[::−1]`“) je tudi to, da pri obračanju ustvarita novo kopijo seznama `besede`; je pa res, da tako stara kot nova kopija kažeta na iste elemente (v našem primeru nize, ki predstavljajo posamezne besede iz vhodne vrstice), tako da potrata prostora vendarle ni prehuda.

R1988.1.3

Naloga: str. 1

Prejšnje stanje tipke si zapomnimo v spremenljivki tPrej; z njeno pomočjo lahko ugotovimo, kdaj je uporabnik pritisnil ali spustil tipko. Ko uporabnik pritisne tipko, spremenimo stanje ure (spremenljivka Tece) in začnemo odštevati čas (spremenljivka Brisi): če ostane pritisnjena dovolj dolgo, bomo postavili (ko bo padla Brisi na 0) čas na 0. Če pa tipko prej spusti, postavimo Brisi takoj na 0, kar nam zagotavlja, da se do naslednjega pritiska na tipko s to spremenljivko ne bomo več ukvarjali.

```
program Stoparica;
var
  Tece: boolean;          { ura teče }
  Brisi: integer;         { 0 ali čas v stotinkah do brisanja časa }
  t, tPrej: boolean;      { trenutno in prejšnje stanje tipke }
  h, m, s, cs: integer;   { ure, minute, sekunde, stotinke }

  function Tipka: boolean; external;
  function Impulz: boolean; external;
  procedure Zaslon(h, m, s, cs: integer); external;

begin {Stoparica}
  Tece := false; Brisi := 0; tPrej := false;
  h := 0; m := 0; s := 0; cs := 0; Zaslon(h, m, s, cs);
  repeat
    t := Tipka;
    if t and not tPrej then begin Tece := not Tece; Brisi := 50 end
    else if tPrej and not t then Brisi := 0;
    tPrej := t;
    if Impulz then begin
      if Brisi > 0 then begin
        Brisi := Brisi - 1;          { merimo čas do brisanja časa }
        if Brisi = 0 then begin      { čas je za brisanje zaslona }
          h := 0; m := 0; s := 0; cs := 0; Zaslon(h, m, s, cs);
          Tece := false;             { po brisanju naj ura stoji }
        end; {if}
      end; {if}
      if Tece then begin             { če ura teče, popravimo zaslon }
        cs := cs + 1;
        if cs >= 100 then begin
          s := s + 1; cs := cs - 100;
          if s >= 60 then begin m := m + 1; s := s - 60 end;
          if m >= 60 then begin h := h + 1; m := m - 60 end;
          if h >= 24 then h := h - 24;
        end; {if}
        Zaslon(h, m, s, cs);
      end; {if Tece}
    end; {if Impulz}
  until false;
end. {Stoparica}
```

R1988.1.4

Naloga: str. 2

Program prešteje, koliko je popolnih kvadratov od 2 do prebranega števila a. Za štetje uporablja spremenljivko d. Ta se poveča za ena natančno tedaj, ko veljata hkrati pogoja:

$$p = q \qquad \text{in} \qquad (k \bmod p) = 0.$$

Ker je $q = k \operatorname{div} p$ in je ostanek tega deljenja takrat 0, velja $p \cdot q = k$; in ker je $p = q$, velja takrat tudi $p \cdot p = k$, torej se d poveča natanko tedaj, ko je k popoln kvadrat.

Popolnih kvadratov med 2 in vključno 99 je osem, med 2 in 100 pa devet. V prvem primeru program torej izpiše 8, v drugem pa 9. V splošnem torej program izpiše $\lfloor \sqrt{a} \rfloor - 1$.

REŠITVE NALOG ZA DRUGO SKUPINO

Naloga: str. 2

R1988.2.1

Pot lahko opišemo s tem, da pri vsakem prebranem bloku povemo, katerega od nadaljevalnih naslovov bomo uporabili: tistega, ki kaže na naslednji blok, ali tistega, ki kaže na blok z dvakrat večjo številko. Za to nam za vsak korak na poti zadostuje po en bit.

Najkrajša pot je tista, pri kateri kar največkrat uporabimo skok na blok z dvakratno številko, saj z njim pridemo najdlje.[3] Najlaže to pot določimo v smeri od zadnjega bloka proti začetku datoteke. Če je številka bloka sodo število, do tega bloka najhitreje pridemo iz bloka s pol manjšo številko; do bloka z liho številko pa ne moremo drugače kot iz predhodnega bloka. Ostale podrobnosti so razvidne iz programa.

```
const MaxPot = 32;                                  { največja dolžina poti }
type SmerT = (Naprej, Skok);                        { tip koraka do naslednjega bloka }
     PotT = packed array [1..MaxPot] of SmerT;      { opis poti }

procedure PotDoBloka(StZap: integer; var Pot: PotT);
{ Določi najkrajšo pot do bloka datoteke na disku. }
var
  Korak: integer;      { števec korakov na poti }
  Blok: integer;       { števec blokov, skozi katere gre pot }
  Zadnji: integer;     { zadnji korak }
  t: SmerT;
begin
  { Najprej določimo pot v smeri od bloka nazaj proti začetku datoteke. }
  Korak := 0; Blok := StZap;
  while Blok > 1 do begin
    if Odd(Blok) then
      begin Korak := Korak + 1; Pot[Korak] := Naprej; Blok := Blok - 1 end;
    Korak := Korak + 1; Pot[Korak] := Skok; Blok := Blok div 2;
  end; {while}
  { sedaj pot obrnemo }
  Zadnji := Korak;
  for Korak := 1 to Zadnji div 2 do begin
```

[3] Natančneje povedano: nobena rešitev nima strogo več skokov kot najkrajša. O tem se bomo prepričali kasneje, ko bomo videli, da ima najkrajša pot do bloka B natanko $\lfloor \lg B \rfloor$ skokov. Če ima neka druga pot do tega bloka recimo s skokov, velja $2^s \leq B$, saj nam vsak skok podvoji številko trenutnega bloka, ta pa ne sme preseči B; torej je $s \leq \lg B$, ker pa je s celo število, pomeni to tudi $s \leq \lfloor \lg B \rfloor$.

Ni pa nujno, da je vsaka rešitev s tem maksimalnim številom skokov že tudi najkrajša — skoke moramo izvesti ob pravem času. Na primer, najkrajša pot do 15 je $1 \Rightarrow 2 \to 3 \Rightarrow 6 \to 7 \Rightarrow 14 \to 15$ (puščice $\to$ ponazarjajo korake na naslednji blok, $\Rightarrow$ pa skoke na dvakratno številko) in ima tri skoke (ali dva, odvisno od tega, kako štejemo korak od 1 do 2). Toda poti s tremi skoki je še več, niso pa vse najkrajše; $1 \Rightarrow 2 \Rightarrow 4 \Rightarrow 8 \to 9 \to 10 \to \cdots \to 15$ ima na primer tudi tri skoke, pa je precej daljša od najkrajše poti.

```
      t := Pot[Korak];
      Pot[Korak] := Pot[Zadnji − Korak];
      Pot[Zadnji − Korak] := t;
    end; {for}
end; {PotDoBloka}
```

Do bloka s številko 2^n pridemo zelo hitro: v n korakih, pri čemer vedno uporabimo naslov dvakrat bolj oddaljenega bloka. Največ dela imamo, da preberemo blok s številko $2^n - 1$. To vidimo takole: da do tega bloka pridemo iz bloka s številko $2^{n-1}-1$, porabimo dva koraka. Od začetka datoteke tako opravimo $2n-2$ korakov, kar je mogoče enostavno dokazati z matematično indukcijo. Podobno dokažemo tudi, da do tega bloka ni krajše poti. Za zapis vseh poti do blokov s številkami, manjšimi ali enakimi 2^n, torej potrebujemo najmanj $2n-2$ bitov.

Prepričajmo se z indukcijo, da naš postopek res vedno daje najboljšo rešitev. Naj bo $\#_0(B)$ število ničel v dvojiškem zapisu števila B, $\#_1(B)$ pa število enic. Iz glavne zanke **while** našega programa se takoj vidi, da je pot, ki jo ta program najde do bloka B, dolga $f(B) := \#_0(B) + 2(\#_1(B) - 1)$ korakov. Označimo z $g(B)$ dolžino najkrajše poti do B; očitno je $g(1) = 0$ (da pridemo do bloka 1, nam ni treba narediti ničesar, ker že vnaprej vemo, kje se nahaja), $g(2B+1) = g(2B) + 1$ (ker se vse poti do bloka z liho številko $2B+1$ končajo s korakom od $2B$ do $2B+1$ in je zato tudi najkrajša pot do $2B+1$ lahko le podaljšek najkrajše poti do $2B$) in $g(2B) = \min\{g(B), g(2B-1)\} + 1$ (vse poti do bloka s sodo številko $2B$ se končajo ali s korakom od $2B-1$ do $2B$ ali pa s skokom od B do $2B$, najkrajša pa bo pač tista med njimi, ki je podaljšek najkrajše poti do $2B-1$ ali pa do B).

O tem, da vrača naš program vedno najboljšo rešitev, se lahko prepričamo, če pokažemo, da je $g(B) = f(B)$ pri vseh B. Za $B = 1$ je to očitno res, saj je $f(B) = g(B) = 0$. Recimo zdaj, da velja $f(b) = g(b)$ za vse $b = 1, 2, \ldots, B-1$. Če je B lih, se njegov dvojiški zapis od $B-1$ razlikuje le v tem, da ima $B-1$ v najnižjem bitu ničlo, B pa enico; zato je $f(B) = f(B-1) + 1$; po induktivni predpostavki je $f(B-1) = g(B-1)$, po definiciji g pa je $g(B) = g(B-1) + 1$ (saj je B lih), tako da sledi $f(B) = g(B)$.

Recimo zdaj, da je B sod; v dvojiškem zapisu ima na najnižjih nekaj mestih ničle, prej ali slej pa tudi neko enico: recimo, da ima na najnižjih k bitih ničle, na naslednjem pa enico. Ker je B sod, se števili B in $B/2$ v dvojiškem zapisu razlikujeta le po tem, da ima $B/2$ na koncu eno ničlo manj, tako da je $f(B/2) = f(B) - 1$ (po definiciji f); število $B-1$ pa se od B razlikuje po tem, da ima v dvojiškem zapisu na spodnjih k mestih enice, na naslednjem pa ničlo (le tako je namreč mogoče, da s prištevanjem 1 dobimo v vsoti na spodnjih k mestih same ničle: drugače bi se prenos pri seštevanju že prej ustavil) in iz definicije f sledi $f(B-1) = f(B) + k - 1$ ($+k$ zato, ker ima $B-1$ enice na spodnjih k mestih, B pa ima tam ničle, -1 pa zato, ker ima $B-1$ na naslednjem mestu ničlo, B pa enico). Izjema je le, če je dvojiški zapis števila $B-1$ za eno mesto krajši od zapisa števila B, torej če je $B-1$ sestavljen iz samih enic ($B-1 = 2^{k+1}-1$ je iz k enic); tedaj $B-1$ nima tiste ničle levo od svojih enic in velja $f(B-1) = f(B) + k - 2$. Ker nas zanimajo $B > 1$, je $k \geq 1$, tako da iz $f(B-1) \geq f(B) + k - 2$ sledi $f(B-1) \geq f(B) - 1 = f(B/2)$.

Po induktivni predpostavki je $f(B/2) = g(B/2)$ in $f(B-1) = g(B-1)$ in po definiciji g je $g(B) = \min\{g(B/2), g(B-1)\} + 1$, kar je naprej enako

$\min\{f(B/2), f(B-1)\}+1$, to pa je zaradi $f(B-1) \geq f(B/2)$ enako $f(B/2)+1$, za kar smo zgoraj ugotovili, da je enako $f(B)$. Torej velja $f(B) = g(B)$ tudi v tem primeru.

Naloga: str. 3

R1988.2.2

Spodnji podprogram v zanki preverja istoležne znake modela in niza. Čim odkrijemo kakšno neujemanje med trenutnima črkama, lahko takoj nehamo in že vemo, da se niz ne ujema z modelom. Zapomniti si moramo tudi, če smo v modelu že prišli do zvezdice (spremenljivka VseObv); tako lahko preverimo, če ni bil ukaz preveč okrajšan.

```
const MaxDolz = 20;
type BesedaT = packed array [1..MaxDolz] of char;

function AliJeUkaz(Model, Niz: BesedaT): boolean;
var
  iNiz: integer;       { števec znakov v nizu }
  iModel: integer;     { števec znakov v modelu ključne besede }
  VseObv: boolean;     { ali smo obdelali obvezni del ukaza }
  SeUjema: boolean;    { ali se obdelani del niza ujema z modelom }
  Konec: boolean;      { ali smo prišli do konca niza }
begin
  iNiz := 1; iModel := 1; VseObv := false; SeUjema := true; Konec := false;
  while SeUjema and not Konec do begin
    if Model[iModel] = '*' then
      begin VseObv := true; iModel := iModel + 1 end
    else if Niz[iNiz] = ' ' then Konec := true
    else if Niz[iNiz] <> Model[iModel] then SeUjema := false
    else begin iNiz := iNiz + 1; iModel := iModel + 1 end;
  end; {while}
  AliJeUkaz := VseObv and SeUjema;
end; {AliJeUkaz}
```

Če bi prišli do konca modela prej kot do konca niza, bi podprogram še vedno pravilno deloval, ker bi primerjal trenutni znak niza s presledkom, na katerega bi naletel na koncu modela, in ugotovil, da se pač ne ujemata.

Naloga: str. 3

R1988.2.3

Če v zanki opazujemo stanje senzorjev (Ozn1, Ozn2) in ga primerjamo s stanjem ob prejšnji meritvi (PrejOzn1, PrejOzn2), lahko opazimo, kdaj smo prišli do začetka naslednje oznake. Če si zapomnimo še čas, ko smo opazili začetek prejšnje oznake (t1, t2), lahko izračunamo hitrost vrtenja: ena oznaka v dt1 oz. dt2 milisekundah. S primerjavo teh dveh hitrosti lahko usmerjamo pogon (v nasprotno smer): če je $1/\mathsf{dt1} > 1/\mathsf{dt2}$, moramo vrtenje prve osi malo upočasniti, sicer pa pospešiti. Ker naloga pravi, da zmore motor le pospeške med -1 in 1, delimo razliko $1/\mathsf{dt1} - 1/\mathsf{dt2}$ s hitrostjo druge osi, tako da pospeški ali pojemki, ki jih bomo zahtevali, ne bodo preveliki (razen če se ne vrti prva os res zelo hitro). Ko na plošči prvič zagledamo začetek oznake, hitrosti še ne moremo oceniti, ker ne vemo, kdaj se je začela prejšnja oznaka; ta primer si zapomnimo s spremenljivkama Uvod1 in Uvod2.

```
program Sklopka;
type
  CasT = integer;
  OsT = 1..2;
```

```
  function Cas: CasT; external;
  function Oznaka(Os: OsT): boolean; external;
  procedure Pogon(Pospesek: real); external;
  procedure Sklopi; external;

var
  Uvod1, Uvod2: integer;        { koraki, potrebni za začetek meritve }
  Ozn1, Ozn2, OznPrej1, OznPrej2: boolean;    { za iskanje začetka oznake }
  t, t1, t2: CasT;              { čas zadnje oznake }
  dt1, dt2: CasT;               { čas med dvema oznakama }
  RelNapaka: real;              { relativna razlika hitrosti osi }
  Izenaceno: boolean;           { je razlika hitrosti osi dovolj majhna? }
begin {Sklopka}
  Uvod1 := 2; Uvod2 := 2; Izenaceno := false;
  OznPrej1 := Oznaka(1); OznPrej2 := Oznaka(2); t1 := 0; t2 := 0;
  repeat
    Ozn1 := Oznaka(1); Ozn2 := Oznaka(2); t := Cas;
    if Ozn1 and not OznPrej1 then begin
      dt1 := t - t1; t1 := t; if Uvod1 > 0 then Uvod1 := Uvod1 - 1;
    end; {if}
    if Ozn2 and not OznPrej2 then begin
      dt2 := t - t2; t2 := t; if Uvod2 > 0 then Uvod2 := Uvod2 - 1;
    end; {if}
    OznPrej1 := Ozn1; OznPrej2 := Ozn2;
    if (Uvod1 = 0) and (Uvod2 = 0) then begin
      RelNapaka := (1/dt1 - 1/dt2) / (1/dt2); Pogon(-RelNapaka);
      Izenaceno := Abs(RelNapaka) < 0.01;
    end; {if}
  until Izenaceno;
  Sklopi;
end. {Sklopka}
```

Naloga pravi, da funkcija Cas šteje milisekunde; če se plošči vrtita dovolj hitro in je na njiju dovolj veliko število belih in črnih oznak, zna biti takšno merjenje časa premalo natančno. Pri n oznakah vsake barve in k obratih na sekundo bi morala naša spremenljivka dt izmeriti $1000/(nk)$ milisekund, v resnici pa bo namerila $\lfloor 1000/(nk) \rfloor$ ali pa $\lceil 1000/(nk) \rceil$. Če torej namestimo d milisekund, utegne biti prava vrednost karkoli med $d-1$ in $d+1$, pravo število obratov n pa zato karkoli med $(d-1)n/1000$ in $(d+1)n/1000$. Krajše ko so oznake (večji n) in hitreje ko se vrti opazovana os (manjši d), bolj je ta naša meritev nenatančna, zato pa je tudi razlika v hitrosti osi, ko ju na koncu poskušamo sklopiti, lahko večja. Natančnejše meritve bi lahko dobili, če bi merili čas v manjših enotah (s tem bi v zadnjih dveh formulah namesto 1000 dobili nek večji imenovalec) ali pa bi namesto časa od začetka ene do začetka naslednje črne oznake gledali po več oznak zaporedoma (kar ima tak učinek, kot če bi se zmanjšal n); pri slednjem moramo seveda upati, da se hitrost vrtenja ne spreminja prehitro, da bi to preveč pokvarilo naše meritve.

Krmiljenje pogona v tej rešitvi je tudi sicer le zasilno; potreben čas za izenačitev hitrosti ni najmanjši. Za optimalno krmiljenje bi bilo potrebno poznavanje odziva mehanizma in upoštevanje metod krmiljenja procesov s povratnimi zankami.

Naloga: str. 4

R1988.2.4

Program v notranji zanki **for** določi obrat obr štirimestnega števila stv; obrat 1234 je 4321, obrat 1 je 1000. V zunanji zanki zato uravnoteženo računa razliko naslednjih dveh vsot:

$$\begin{aligned} S_1 &= 2 + 3 + \ldots + 9999 \\ &= (1 + 2 + \ldots + 9999) - 1 \\ &= S - 1 \\ \text{in } S_2 &= o(2) + o(3) + \ldots + o(9999) \\ &= (o(1) + o(2) + \ldots + o(9999)) - o(1) \\ &= S - 1000. \end{aligned}$$

Tu smo z $o(n)$ označili obrat štirimestnega števila n. V vsoti S_1 nastopajo vsa štirimestna števila razen 1 (med štirimestna števila štejemo vsa števila, ki jih lahko zapišemo z največ štirimi števkami), v drugi vsoti pa, čeprav v premešanem vrstnem redu, ravno tako vsa štirimestna števila, razen obrata 1, ki je enak 1000. Program izpiše razliko obeh vsot, to je

$$S_1 - S_2 = (S - 1) - (S - 1000) = 999.$$

Če bi vrednost pStv spremenili v 1, bi program izračunal razliko $S - S$, zato bi izpisal 0.

REŠITVE NALOG ZA TRETJO SKUPINO

Naloga: str. 4

R1988.3.1

Eden od najhitrejših postopkov za iskanje praštevil na enoprocesorskem sistemu je Eratostenovo sito. Vzemimo ga tudi za osnovo postopka na večprocesorskem sistemu. Da bo postopek učinkovit, mora biti delo enakomerno porazdeljeno med vse procesorje (vsak bo rešetal svoj približno enako velik del tabele števil).

Ob zagonu vsi procesorji opravijo inicializacijo: kot prvo praštevilo izberejo 2. Vsi procesorji potem delajo po istem postopku: vsak si najprej določi del sita (tabele), v katerem rešeta in iz svojega dela tabele odstrani vse mnogokratnike trenutnega praštevila. Nato vsak procesor v svojem delu sita poišče najmanjše še ne odstranjeno število. To število je kandidat za novo praštevilo. Vsak procesor nato počaka, da tudi ostali izberejo vsak svojega kandidata; kot novo praštevilo vsi opisani procesorji izberejo najmanjšega med kandidati. Nato vsi ponove opisani postopek.

Ker je v vsakem delu tabele približno enako mnogo mnogokratnikov vsakokratnega praštevila, vsi procesorji končajo z delom v približno enakem času. Zato lahko ocenimo, da z vzporednim postopkom praštevila določimo približno desetkrat hitreje kot z navadnim. Nekaj časa več je potrebnega le zaradi delitve dela.

```
program VzporednoSejanje;
const
  MaxStev = 100000;          { Gornja meja območja iskanih praštevil. }
  StProc = 10;               { Število procesorjev. }
var
  { Skupne spremenljivke. V tabeli Kandidat vsak procesor predlaga,
    s katerim praštevilom bi se v nadaljevanju ukvarjali (obvelja najmanjše). }
  Kandidat: shared array [1..StProc] of integer;
```

```
  JePrast: shared packed array [1..MaxStev] of boolean;    { Sito. }

  { Lokalne spremenljivke. }
  Stev: integer;                { Trenutno število. }
  Prast: integer;               { Trenutno praštevilo, čigar večkratnike črtamo. }
  SpMeja, ZgMeja: integer;      { Meji obdelovanega dela sita. }
  Oznaka: 0..StProc - 1;        { Oznaka procesorja }

  procedure MojaOznaka(var Oznaka: integer); external;
  procedure Zaseden; external;
  procedure Prost; external;
  function VsiProsti: boolean; external;
  function VsiZasedeni: boolean; external;

begin {VzporednoSejanje}
  { Inicializacija. }
  MojaOznaka(Oznaka); Zaseden; Prast := 2;
  { Vsak procesor počisti svoj del sita. }
  for Stev := 1 + MaxStev * Oznaka div StProc to
              MaxStev * (Oznaka + 1) div StProc do JePrast[Stev] := true;
  { Počakajmo na ostale. }
  repeat until VsiZasedeni; Prost; repeat until VsiProsti;

  { Začnimo z rešetanjem. }
  while Prast <= MaxStev do begin
    Zaseden;

    { Pregledali bomo števila od Prast do MaxStev in prečrtali večkratnike
      števila Prast. Razdelimo si ta kos tabele na StProc delov; naš
      procesor bo pregledal indekse SpMeja..ZgMeja. }
    SpMeja := Prast + (MaxStev - Prast + 1) * Oznaka div StProc;
    ZgMeja := Prast + ((MaxStev - Prast + 1) * (Oznaka + 1) div StProc) - 1;
    { Poiščimo najmanjši večkratnik števila Prast v svojem delu tabele. }
    Stev := ((SpMeja + Prast - 1) div Prast) * Prast;
    if Stev <= Prast then Stev := Stev + Prast;

    { Presejmo svoj del tabele. }
    while Stev <= ZgMeja do
      begin JePrast[Stev] := false; Stev := Stev + Prast end;

    { Poiščimo kandidata za naslednje praštevilo. }
    Stev := SpMeja; if Stev <= Prast then Stev := Prast + 1;
    if Stev <= SpMeja then
      while (Stev < ZgMeja) and not JePrast[Stev] do Stev := Stev + 1;
    if Stev >= SpMeja then Kandidat[Oznaka] := MaxStev + 1
    else Kandidat[Oznaka] := Stev;
    { Počakajmo na ostale. }
    repeat until VsiZasedeni; Prost; repeat until VsiProsti;

    { Določimo naslednje praštevilo }
    Zaseden;
    Prast := MaxStev + 1;
    for Stev := 0 to StProc - 1 do
      if Kandidat[Stev] < Prast then Prast := Kandidat[Stev];
    { Počakajmo na ostale. }
    repeat until VsiZasedeni; Prost; repeat until VsiProsti;

  end; {while}
```

```
end. {VzporednoSejanje}
```

Tehnika, ki jo gornji program uporablja za usklajevanje (sinhronizacijo) dela posameznih procesorjev, se imenuje „sinhronizacija z oviro“ (*barrier synchronization*): ko pride nek procesor do ovire, mora počakati, dokler ne pridejo do nje še vsi ostali.[4] Preden označi procesor sebe za prostega, mora počakati, da so vsaj za hip vsi označeni kot zasedeni — to je znak, da so že zapustili prejšnjo oviro. Zato je dobro, da imamo na razpolago podprogram VsiZasedeni. Brez tega bi se lahko zgodilo naslednje: ko procesorji čakajo na oviri in jo doseže še zadnji med njimi, bi zdaj eden pač prvi opazil, da so vsi prosti, in bi oddrvel naprej; in zdaj bi se lahko zgodilo, da bi se ta razglasil za zasedenega, še preden bi drugi uspeli opaziti, da so bili nekaj časa vsi hkrati prosti. Zato bi drugi še naprej čakali na prejšnji oviri, medtem pa bi naš procesor tekel naprej in se ustavil šele na naslednji oviri; ko bi se takrat razglasil za prostega, bi se lahko ponovila ista težava. Spet bi lahko eden od procesorjev oddrvel naprej, mogoče celo isti kot prej. Naš program bi po vsem tem lahko dajal čisto napačne rezultate.

Razmislimo še o tem, kako bi lahko naredili oviro, če ne bi imeli funkcij, kot sta VsiZasedeni in VsiProsti. Če bi imeli na voljo kakšen zanesljiv mehanizem za zaklepanje (na primer inštrukcijo, ki prebere in nato zapiše neko pomnilniško celico in pri tem zagotavlja, da je med branjem in pisanjem ne bo prekinil noben drug procesor — spodaj tako operacijo predstavlja funkcija BeriInPisi), bi lahko oviro izvedli s števcem. Prvotna vrednost števca naj bo 0; vsak procesor, ki pride do ovire, mora števec povečati za 1, nato pa v zanki opazuje vrednost števca in ko ta doseže StProc, zapusti oviro. Z zaklepanjem lahko zagotovimo, da pri povečevanju števca procesorji ne bodo hodili drug drugemu v zelje.

```
  var Kljucavnica: shared integer value 0; { za dostop do števca }
      Stevec: shared integer value 0;  { koliko jih čaka na trenutni oviri }
      TrenutnaOvira: integer value 0;  { ovira, ki jo je naš procesor nazadnje zapustil }
      KoncanaOvira: shared integer value 0; { ovira, ki se je nazadnje sprostila
                                              (ker so jo dosegli vsi procesorji) }

  { Vpiše N v S in vrne staro vrednost S-ja. To opravi kot atomarno (nedeljivo)
    operacijo, torej med našim branjem in pisanjem ne more nihče drug do S. }
  function BeriInPisi(var S: integer; N: integer): integer; external;

procedure Ovira;
var St: integer;
begin
  TrenutnaOvira := 1 - TrenutnaOvira;          { Gremo na naslednjo oviro. }
  repeat until BeriInPisi(Kljucavnica, 1) = 0; { Zasezimo števec. }
  St := Stevec + 1; Stevec := St;   { Povečajmo števec procesorjev na tej oviri. }
  Kljucavnica := 0;                            { Sprostimo števec. }
  if St = StProc then    { Mi smo zadnji na tej oviri — dajmo znak za konec. }
    begin St := 0; KoncanaOvira := TrenutnaOvira end
  else    { Počakajmo, da pridejo še drugi do ovire in jo eden od njih sprosti. }
    repeat until KoncanaOvira = TrenutnaOvira;
end; {Ovira}
```

Ker naša naloga ne omenja, da bi imeli na voljo zaklepanje, lahko oviro izvedemo takole:

[4] Gl. npr. J. L. Hennessy, D. A. Patterson: *Computer Architecture: A Quantitative Approach*, 2. izd., 1996, str. 700–703.

```
var NaOviri: shared array [1..StProc] of integer;

procedure Ovira;
var i, TaOvira, NaslednjaOvira: integer; Ok: boolean;
begin
  TaOvira := (NaOviri[Oznaka] + 1) mod 3;
  NaOviri[Oznaka] := TaOvira;
  NaslednjaOvira := (TaOvira + 1) mod 3;
  repeat
    { Preverimo, če so že vsi prišli do trenutne ovire. }
    i := 0; Ok := true;
    while Ok and (i <= StProc) do begin
      Ok := (NaOviri[i] = TaOvira) or (NaOviri[i] = NaslednjaOvira);
      i := i + 1;
    end; {while}
  until Ok;
end; {Ovira}
```

Vrednost NaOviri[i] torej pove, katera je zadnja ovira, ki jo je procesor i že dosegel (in mogoče tudi zapustil). Na oviri moramo čakati, dokler ne ugotovimo, da so vsi že na trenutni ali pa na naslednji oviri. Možnost, da je kakšen procesor A že na naslednji oviri, se lahko zgodi le v primeru, da je A že prej opazil, da so vsi na trenutni oviri, in oddrvel naprej do naslednje ovire, še preden smo mi sploh ponovno preverili, ali so vsi na trenutni oviri ali ne. Pomembno pa je, da A ne bo mogel oddrveti dlje kot do naslednje ovire, saj on zdaj tam čaka, da bodo tudi vsi ostali prišli vsaj do tiste ovire — dokler je kdo še na trenutni oviri, se A ne more premakniti naprej. Zato tudi štejemo ovire s števili 0, 1, 2 in ne le 0, 1: v slednjem primeru ne bi mogli ločiti procesorjev, ki čakajo še na prejšnji oviri, od tistih, ki čakajo že na naslednji.

Priznati je treba, da bi bil podprogram Ovira verjetno razmeroma neučinkovit: ko nek procesor spremeni svojo celico NaOviri[Oznaka], jo bodo tisti, ki trenutno še čakajo v zanki **repeat**, kmalu spet prebrali; ker se je spremenila, jo bodo morali na novo potegniti iz skupnega pomnilnika. To bi utegnilo povzročiti veliko konfliktov na vodilu.

Pri inicializaciji moramo zagotoviti, da dobijo vsi elementi tabele NaOviri vrednosti 0, še preden bo kakšen od procesorjev prvič poklical podprogram Ovira. V nasprotnem primeru bi se namreč lahko zgodilo, da bi Ovira že poskušala postaviti nek element tabele NaOviri na neko novo vrednost, ki bi označevala, da je ta procesor prišel do te ovire, nato pa bi nek drug procesor v okviru inicializacije povozil celo tabelo z ničlami. To, da bi vsak procesor postavil na nič le svojo celico tabele NaOviri, ni dovolj, ker potem pri prvem klicu podprograma Ovira ne vemo, katere celice že vsebujejo smiselne vrednosti, katere pa še ne. S podprogramom VsiProsti si ne moremo pomagati, kajti tudi če se med inicializacijo vsi procesorji razglašajo za zasedene, to, da so vsi prosti, še ne pomeni, da so res že opravili z inicializacijo — mogoče je sploh še niso začeli. Res zanesljiva rešitev bi bila, da bi nek kontrolni program postavil tabelo NaOviri na 0, še preden se naši paralelni programi sploh začnejo izvajati.

Mimogrede, Eratostenovo sito lahko z nekaj preprostimi prijemi še pospešimo, tako v enoprocesorski kot v tu opisani paralelni različici. Na primer: prva stvar, ki jo bo program naredil, je to, da bo prečrtal v situ vsa soda števila (razen 2); torej lahko 2 obravnavamo kot poseben primer, v situ pa hranimo le liha števila; tako prihranimo pol pomnilnika. Ko brišemo iz rešeta večkratnike

praštevila p, nima smisla začeti pri p ali $2p$. Vsi p-jevi večkratniki do vključno $(p-1)p$ imajo očitno tudi nek faktor, manjši od p; in ker obravnavamo praštevila v naraščajočem vrstnem redu, smo jih morali prečrtati že prej. Torej lahko začnemo črtati p-jeve večkratnike šele od p^2 naprej, postopek pa lahko ustavimo, čim velja $p^2 >$ MaxStev (saj odtlej ne bi prečrtali ničesar več). Poleg tega tudi ni treba gledati vseh p-jevih večkratnikov: eno od števil kp in $(k+1)p$ je gotovo sodo in ga torej v tabeli ni; tako je dovolj, če pri praštevilu p gledamo večkratnike p^2, $(p+2)p$, $(p+4)p$ in tako naprej.

Možne so tudi bolj zapletene izboljšave Eratostenovega sita; glej opombo na koncu rešitve naloge 2001.1.4.

Tudi na `online-judge.uva.es` je nekaj nalog, kjer pride prav Eratostenovo sito (npr. #367, #10311).

Naloga: str. 5

R1988.3.2

Nalogo najlaže rešimo z rekurzivnim sestopom;[5] v nizu poiščemo prvi znak, ki se ujema s prvim znakom podniza, nato pa iskanje ponovimo s preostankom podniza v preostanku niza. Če pri tem sestopu podniz izpraznimo, smo našli pojavitev; če ne najdemo poti naprej, pa se vrnemo na prejšnji nivo. Ta metoda je eden od osnovnih načinov za reševanje problemov v umetni inteligenci.

```
program IsciPodniz(Input, Output); { Ugotovi, kolikokrat podniz nastopa v nizu. }

  const MaxDolz = 20;
  type NizT = packed array [1..MaxDolz] of char;
  var Niz, Podniz: NizT;
      Stevilo: integer;    { število pojavitev niza v podnizu }

  { Primerja del niza od ZacNiz dalje s podnizom od ZacPodniz dalje. }
  procedure Primerjaj(ZacNiz, ZacPodniz: integer; var Stevilo: integer);
  begin
    if (ZacNiz > MaxDolz) or (ZacPodniz > MaxDolz) then begin end
    else if Podniz[ZacPodniz] = ' ' then Stevilo := Stevilo + 1
    else begin
      while ZacNiz <= MaxDolz do begin
        if Niz[ZacNiz] = Podniz[ZacPodniz] then
          Primerjaj(ZacNiz + 1, ZacPodniz + 1, Stevilo);
        ZacNiz := ZacNiz + 1;
      end; {while}
    end; {if}
  end; {Primerjaj}

begin
  Write('Vnesi niz: '); ReadLn(Niz);
  Write('Vnesi podniz: '); ReadLn(Podniz);
  Stevilo := 0; Primerjaj(1, 1, Stevilo);
  WriteLn;
  WriteLn('Podniz "', Podniz, '" se ', Stevilo, '-krat ',
          'pojavi v nizu "', Niz, '".');
end. {IsciPodniz}
```

Če bi na začetku izmerili dolžino niza in podniza, bi lahko pogoj v zanki **while** zamenjali z malo strožjim ZacNiz <= DolzinaNiza − DolzinaPodniza + ZacPodniz. S

[5] Obstajajo pa tudi precej učinkovitejše rešitve od tu opisane, npr. z dinamičnim programiranjem; gl. nalogo 1997.2.3, ki je praktično povsem enaka tej nalogi.

tem se izognemo brezplodnemu pregledovanju primerov, ko je v nizu ostalo manj znakov kot v podnizu. Vendar je postopek kljub temu še vedno zelo neučinkovit.

R1988.3.3

Naloga: str. 5

Spodnji program hrani v tabeli Vrsta zadnjih 40 vzorcev. Toliko jih potrebujemo zaradi možnosti, da pride do daljše skupine napačnih vzorcev: ko opazimo, da je trenutni vzorec in še prejšnjih devet napačnih, moramo trideset pravilnih pred njimi utišati, tako da potrebujemo v vrsti vsaj zadnjih 40 vzorcev. Z drugimi besedami, šele ko pride za nekim vzorcem še 40 drugih, smo lahko prepričani, da že poznamo dokončno vrednost tistega vzorca. V spremenljivki ZadnjiPrav si bomo zapomnili, kateri je zadnji pravilni vzorec pred trenutnim; tako lahko opazimo, če pride do krajše napake in moramo vmesne vzorce interpolirati, opazimo pa lahko tudi, kdaj dobimo deseto zaporedno napako in moramo vzorce pred tem utišati. V slednjem primeru tudi postavimo Utisaj na 30; to vrednost ohrani, dokler opažamo napačne vzorce, nato pa jo začnemo zmanjševati in s tem nadziramo postopno linearno ojačanje zvoka. Naloga ne predpisuje, kaj storiti v primeru, če med dvema dolgima zaporedjema napak nastopi peščica (59 ali manj) dobrih vzorcev: pri nekem konkretnem dobrem vzorcu lahko zahteva prejšnja skupina napak oslabitev s faktorjem $a/31$, naslednja skupina napak pa z $b/31$. Ena možnost bi bila, da bi ga pomnožili kar z $\min\{a,b\}/31$, naš spodnji program pa ga v takem primeru oslabi z vsakim faktorjem posebej, kar ustreza množenju vzorca z $(a/31)(b/31)$.

```
program CD;

  const m = 40;                { dolžina zakasnilne vrste }
  type VzorecT = 0..65535;     { vzorčena amplituda signala }

  procedure PosljiVzorec(Vzorec: VzorecT); external;
  procedure DobiVzorec(var Vzorec: VzorecT; var Pravilen: boolean); external;

var
  Vrsta: array [1..m] of VzorecT;   { zakasnilna vrsta }
  Pravilen: boolean;                { ali je vzorec pravilen }
  ZadnjiPrav: integer;              { indeks zadnjega dobrega vzorca v vrsti }
  Utisaj: integer;                  { utišati je treba toliko naslednjih vzorcev }
  i: integer;
  k: real;
begin
  ZadnjiPrav := m; Utisaj := 0;
  for i := 1 to m do Vrsta[i] := 0;
  repeat
    PosljiVzorec(Vrsta[1]);                    { pošlji najstarejši vzorec }
    for i := 2 to m do Vrsta[i - 1] := Vrsta[i];   { pomakni vrsto }
    DobiVzorec(Vrsta[m], Pravilen);
    if ZadnjiPrav > 1 then ZadnjiPrav := ZadnjiPrav - 1;
    if Pravilen then begin
      if Utisaj > 0 then begin
        { zaradi prejšnje daljše napake postopno povečujemo jakost }
        Vrsta[m] := Vrsta[m] * (30 - Utisaj + 1) div 31;
        Utisaj := Utisaj - 1;
      end; {if}
      if m - ZadnjiPrav > 1 then begin          { interpoliraj }
        k := (Vrsta[m] - Vrsta[ZadnjiPrav]) / (m - ZadnjiPrav);
```

```
        for i := ZadnjiPrav + 1 to m − 1 do
          Vrsta[i] := Vrsta[ZadnjiPrav] + Round(k * (i − ZadnjiPrav));
      end; {if}
      ZadnjiPrav := m;
    end else if Utisaj = 30 then begin          { daljša napaka še vedno traja }
      Vrsta[m] := 0; ZadnjiPrav := m;
    end else if m − ZadnjiPrav > 9 then begin   { zadnjih 10 vzorcev je napačnih }
      { postopno utišaj 30 vzorcev pred napačnimi }
      for i := ZadnjiPrav − 30 + 1 to ZadnjiPrav do
        Vrsta[i] := Vrsta[i] * (ZadnjiPrav − i + 1) div 31;
      for i := ZadnjiPrav + 1 to m do Vrsta[i] := 0;
      { treba bo postopno dvigniti jakost 30-tim pravilnim vzorcem }
      Utisaj := 30; ZadnjiPrav := m;
    end; {if}
  until false;
end. {CD}
```

Naloga: str. 6

R1988.3.4

Postopek kodiranja je naslednji: Aleš najprej zakodira vsebino sporočila s svojo skrivno funkcijo f_A. Potem sporočilu doda svoj podpis (da Bojanu olajša iskanje pošiljatelja) in celotno sporočilo zakodira še z Bojanovo javno funkcijo g_B. Tako zakodirano sporočilo lahko Bojan (in le Bojan) razkodira najprej z uporabo svoje skrivne funkcije f_B. Tako zve za podpisnika sporočila (vendar v avtentičnost še ne more biti prepričan). Preostali del sporočila sedaj razkodira z Aleševo javno funkcijo g_A. Če je sporočilo čitljivo, je s tem preveril, da mu je sporočilo res poslal Aleš.

Zapišimo postopek še drugače (funkcija f je skrivna, g pa javna):

$$x \xrightarrow[f_A]{\text{Aleš}} f_A(x) \xrightarrow[g_B]{\text{Aleš}} g_B(f_A(x)) \xrightarrow[f_B]{\text{Bojan}} f_B(g_B(f_A(x))) \quad \| \quad f_A(x) \xrightarrow[g_A]{\text{Bojan}} g_A(f_A(x)) = x.$$

Problem nastopi le tedaj, ko sta naslovnik in pošiljatelj ista oseba, saj tedaj vsebina sporočila potuje nezakodirana. Možna rešitev bi bila, da bi imel vsakdo dva različna para ključev: en par bi se uporabljal, ko je ta človek v vlogi pošiljatelja, drugi par pa, ko je ta človek prejemnik. Prvi par bi bil torej namenjen podpisovanju (ugotavljanju istovetnosti pošiljatelja), drugi pa šifriranju (da sporočila ne more prebrati nihče razen prejemnika).