17. državno tekmovanje v znanju računalništva (1993)

NALOGE ZA PRVO SKUPINO

1993.1.1

Našli smo kos papirja, na katerem je zapisan program. Ali lahko ugotoviš, **kaj program izpisuje?** Pri tem upoštevaj, da so vrednosti konstant c1, c2 in c3 dovolj majhne, da pri računanju ne pride do prekoračitve obsega celih števil. Odgovor utemelji! R: 7

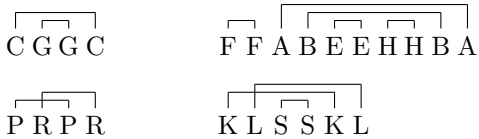
```
program Mac(Output);

const
  c1 = ...;
  c2 = ...;
  c3 = ...;
var
  a, b, c: integer;

begin
  a := c1;
  b := c1 + c2;
  c := c1 + c2 + c3;
  repeat
    WriteLn(a);
    a := b - a;
    b := c - b;
    b := a + b;
  until false;
end.
```

```
#include <stdio.h>

#define c1 ...
#define c2 ...
#define c3 ...

int a, b, c;

void main(void)
{
  a = c1;
  b = c1 + c2;
  c = c1 + c2 + c3;
  do {
    printf("%d\n", a);
    a = b - a;
    b = c - b;
    b = a + b;
  } while (1);
}
```

1993.1.2

Z vhoda preberemo niz znakov. Znaki, ki nastopajo v nizu, se pojavijo *natanko* dvakrat. Vsak par znakov določa povezavo. R: 8

Primeri povezav:



Napiši program, ki bo preveril, ali pride do križanja povezav.

R: 9

1993.1.3

Slalomska proga z nVrat vratci je podana s koordinatami vratc: vy[v], vxLevi[v], vxDesni[v] (tri tabele realnih števil). Vratca so podana po vrsti od startnih do ciljnih, pri tem vy narašča (vy[v + 1] > vy[v]), velja pa tudi: vxLevi[v] < vxDesni[v].

Smučarjeva pot je podana s tabelama koordinat sx in sy. Med zaporednima točkama smučar vozi v ravni črti. Predpostaviš lahko, da stoji ob startu smučar nad prvimi vratci (sy[1] < vy[1]) in da smuča le navzdol. Tudi vsi sx[i] in sy[i] so realna števila, pri tem pa ni nujno, da so sy podani na mestih, kjer so postavljena vratca.

Napiši del programa, ki izpiše, ali je smučar pravilno prevozil vso progo (da ni zapeljal zunaj nobenih vratic in da je pripeljal do cilja). Predpostaviš lahko, da so podatki že shranjeni v tabelah:

```
const
  nVratMax = 100;
  nPotMax = 100;
var
  vy, vxLevi, vxDesni: array [1..nVratMax] of real;      { koordinate vratic }
  sx, sy: array [1..nPotMax] of real;                  { smučarjeve koordinate }
  nVrat: integer;                                      { dejansko število vratic }
  nPot: integer;          { dejansko število podanih točk v smučarjevi poti }
```

R: 10

1993.1.4

Napiši program, ki bere angleško besedilo in ga pri tem delno uredi. To pomeni, da večkratne presledke med besedami združi v enega in uredi presledke ob ločilih „.“ in „,“ na naslednji način:

- pred ločiloma „.“ in „,“ ni presledka,
- za ločilom „.“ sta natanko dva presledka, če se naslednja beseda začne z veliko začetnico,
- za ločilom „,“ je natanko en presledek,
- med zaporednimi ločili ni presledkov.

Na voljo imaš podprograma:

```
function GetCh(var ch: char): boolean;
procedure PutCh(ch: char);
```

GetCh prebere znak z vhoda in vrne kot funkcijsko vrednost true. Če ni več podatkov na vhodu, vrne false, ch pa ni definiran. PutCh izpiše znak na izhod. Predpostaviš lahko, da GetCh zamenja konce vrstic s presledki, PutCh pa samodejno lomi besedilo na vrstice.

NALOGE ZA DRUGO SKUPINO

1993.2.1

Programer neznanih kvalitet je napisal naslednji program. **Kaj izpiše program** in zakaj izpiše ravno to? **Odgovor utemelji!** R: 11

```
program Kaj(Input, Output);
const
  n = 42;
  n1 = n − 1;
var
  t: array [0..n1] of integer;
  i: integer;

  function Mac(i: integer): integer;
  var x: integer;
  begin
    if i >= n then Mac := 0
    else begin
      x := t[i];
      t[Mac(i + 1)] := x;
      Mac := n − i;
    end;
  end; {Mac}

begin
  for i := 0 to n − 1 do
    Read(t[i]);
  i := Mac(0);
  for i := 0 to n − 1 do
    Write(t[i], ' ');
end.
```

```
#include <stdio.h>

#define n 42

int t[n];

int Mac(int i)
{
  int x;
  if (i >= n) return 0;
  else {
    x = t[i];
    t[Mac(i + 1)] = x;
    return n − i;
  }
}

void main(void)
{
  int i;
  for (i = 0; i < n; i++)
    scanf("%d", &t[i]);
  i = Mac(0);
  for (i = 0; i < n; i++)
    printf("%d ", t[i]);
}
```

1993.2.2

Koordinatni risalnik s peresi različnih barv sprejema risalne ukaze z računalnika in jih sproti izvršuje. Risalnik pozna naslednje ukaze: R: 12

PeroGor	dvigne pero (naslednji Premik ne riše črte)
PeroDol	spusti pero (naslednji Premik riše daljico)
Pero n	zamenja pero — v roko vzame pero številka n
Premik x, y	premakne pero v ravni črti iz trenutne točke v točko (x, y)
Konec	zaključuje vsako risbo.

Zamenjava peresa je zamudna operacija, zato lahko izris večbarvne risbe precej pospešimo, če zberemo skupaj čim večje število ukazov za risanje daljic v isti

barvi. Ker ne moremo spremeniti obstoječe programske opreme tako, da bi pošiljala ukaze risalniku v optimalnem zaporedju, nam preostane le možnost, da med glavni računalnik in risalnik priključimo preprost vmesni računalnik z majhno količino pomnilnika, ki poskrbi za optimizacijo risanja.

Napiši algoritem za vmesni računalnik, ki sprejema ukaze glavnega računalnika in jih pošilja risalniku tako preurejene, da doseže čim hitrejši izris slike. Na vmesnem računalniku obstajajo funkcije za branje ukaza z glavnega računalnika in za pošiljanje ukaza risalniku.

R: 16

1993.2.3

Imamo kvadratno tabelo števil $n \times n$, ki je urejena tako, da števila strogo naraščajo po vrsticah in po stolpcih.

Primer tabele 5×5:

1	3	7	9	11
2	4	8	12	14
3	5	9	13	15
4	6	12	15	17
5	7	15	16	18

Za nek podatek želimo izvedeti, kolikokrat in kje nastopa v tej tabeli. **Napiši podprogram**, ki dobi kot argument iskano število in izpiše koordinate vseh mest v tabeli, kjer se nahaja to število.

Nasvet: obstaja rešitev, pri kateri ni treba pregledati vseh elementov, kar je pomemben prihranek časa pri velikih tabelah.

R: 17

1993.2.4

Znake v Morsejevi abecedi sestavljajo različno dolga zaporedja kratkih in dolgih piskov (od enega do šest piskov).

- Trajanje dolgih piskov (črtic) je trikrat daljše od trajanja kratkih (pik).
- Pavza med dvema piskoma znotraj enega znaka (črke) traja približno toliko, kot traja kratek pisk.
- Znaki (črke) so med seboj ločeni s pavzo, ki traja približno toliko kot dolg pisk.
- Besede so ločene s pavzo, dolgo za več kot dva dolga piska.

Napiši program, ki bo sprejemal besedilo v Morsejevi telegrafiji in **sproti** izpisoval sprejete pike kot „.“ in črtice kot „_“. Znake naj loči med seboj z enim presledkom, vsaka beseda pa naj bo izpisana v novi vrsti. Primer izpisa:

```
... .__. .-. . .___ . __
__ ___ .-. ... . .___ . ..._ .
.- _... . _.-. . _.. .
```

Na voljo imaš naslednje podprograme:

Start	postavi štoparico na 0 in jo požene,
Cas: integer	vrne čas v milisekundah od starta štoparice,
Piska: boolean	vrne true, če trenutno sprejemamo pisk, sicer false.

Ker je predpisano le razmerje med dolžino kratkega in dolgega piska, ne pa tudi njuno absolutno trajanje, in ker se lahko hitrost telegrafije tudi nekoliko spreminja, naj se bo program sposoben prilagajati trenutni hitrosti telegrafiranja. Ker v začetku sprejema še ne poznamo hitrosti telegrafiranja, je dopustno, da je prvih nekaj znakov napačno prepoznanih (pike prepoznane kot črte ali obratno) ali celo izgubljenih.

Kako bi zagotovil, da bi bili tudi znaki ob začetku sprejemanja vedno pravilno prepoznani, ne glede na hitrost telegrafiranja? (**Postopek** le **opiši**, dopolnjevanje programa ni potrebno.)

NALOGE ZA TRETJO SKUPINO

1993.3.1

Našli smo kos papirja, na katerem je zapisan program. Ali lahko ugotoviš, **kaj program izpisuje?** Odgovor **utemelji**. R: 19

```
program Mac(Output);

const
  c1 = ...;
  c2 = ...;
  c3 = ...;
var
  a, b, c: integer;

begin
  a := c1;
  b := c1 xor c2;
  c := c1 xor c2 xor c3;
  repeat
    WriteLn(a);
    a := b xor a;
    b := c xor b;
    b := a xor b;
  until false;
end.
```

```
#include <stdio.h>

#define c1 ...
#define c2 ...
#define c3 ...

int a, b, c;

void main(void)
{
  a = c1;
  b = c1 ^ c2;
  c = c1 ^ c2 ^ c3;
  do {
    printf("%d\n", a);
    a = b ^ a;
    b = c ^ b;
    b = a ^ b;
  } while (1);
}
```

Opomba: operator **xor** ni del standardnega pascala, najdemo pa ga v nekaterih narečjih. Izvede operacijo „ekskluzivni ali“ nad istoležnimi biti v dvojiškem zapisu celih števil — enako kot operator ^ v C-ju.

Velja: 0 **xor** $0 = 0$; 0 **xor** $1 = 1$; 1 **xor** $0 = 1$; 1 **xor** $1 = 0$.
Primer: 9 **xor** $7 = 1001_2$ **xor** $0111_2 = 1110_2 = 14$.

R: 19

1993.3.2

Marvin je poskeniral obris Slovenije, ker bi rad s svojim avtomatskim šivalnim strojem našil obliko slovenske države. Vektorizacijski program je iz bitne slike naredil množico daljic, opisanih z začetno in končno točko, pri tem pa se je malo zapletel — ni ohranil njihovega vrstnega reda.

Zapiši algoritem, ki bo dobljene daljice uredil v pravo zaporedje. Vhod naj bo množica daljic, podanih s krajiščema. Začetno daljico lahko algoritem izbere poljubno, nato pa naj jih izpiše tako, da bosta imeli dve zaporedni daljici vedno eno skupno točko. Eno skupno točko morata imeti tudi prva in zadnja izpisana daljica. Upoštevaj, da je opis Slovenije enostaven in zaključen — v njem ni nobenih presečišč in križanj. Zato se v vsakem krajišču stikata le dve daljici, krajišča pa se popolnoma pokrivajo — k vsaki daljici lahko najdemo natanko dve njeni sosedi.

R: 22

1993.3.3

Za naš mali procesor (CPU) bi radi napisali razhroščevalnik (angl. debugger). Predvsem nas zanima funkcija, ki bo izvajanje v procesorju vrnila za enega ali več ukazov nazaj — pri tem mora v pomnilniku in procesorju vzpostaviti enako stanje, kot je veljalo prej. V razhroščevalniku potrebujemo dva podprograma. Podprogram PredUkazom se samodejno sproži pred izvajanjem vsakega ukaza — njegova naloga je shraniti ustrezne podatke o trenutnem stanju procesorja. Drugi podprogram UkazNazaj lahko uporabnik pokliče kadarkoli in tudi večkrat zaporedoma — njegova funkcija je vrniti izvajanje procesorja po en ukaz nazaj ob vsakem klicu.

Definicija procesorja je taka:

```
type
  BesedaT = 0..65535;
  UkazT = (LDA, LDAI, STA, STAI, MSUB, MSUBI, JPOS, JPOSI);
var
  M: array [BesedaT] of BesedaT;      { pomnilnik }
  P: BesedaT;                         { programski števec }
  A: BesedaT;                         { akumulator }
```

Procesorjevi ukazi so naslednji (n je tipa BesedaT):

```
LDA n     A := n; P := P + 2                          { nalaganje v akumulator }
LDAI n    A := M[n]; P := P + 2
STA n     M[n] := A; P := P + 2                       { shranjevanje akumulatorja }
STAI n    M[M[n]] := A; P := P + 2
MSUB n    A := n − A; P := P + 2                      { računanje z akumulatorjem }
MSUBI n   A := M[n] − A; P := P + 2
JPOS n    if A ≥ 0 then P := n else P := P + 2        { pogojni skok }
JPOSI n   if A ≥ 0 then P := M[n] else P := P + 2
```

Posamezen ukaz je shranjen v dveh pomnilniških celicah:

M[n] := Ord(ukaz); M[n + 1] := operand ukaza;

Napiši podprograma PredUkazom in UkazNazaj.

1993.3.4

Imamo pravokotno mrežo med seboj povezanih procesorjev, ki delujejo vzporedno. Vsak procesor upravlja z eno točko na zaslonu in komunicira s svojimi štirimi sosedi. Na vseh procesorjih tečejo kopije istega programa. R: 24

Neka višja sila enemu izmed procesorjev pove, da je postal središče kroga s polmerom r (točk), ki se mora pobarvati.

Napiši program, ki bo tekel v vseh procesorjih in bo poskrbel za to, da bodo po nekem končnem času procesorji narisali krog.

V programu lahko uporabiš tri podprograme:

Poslji(Sporocilo: SporociloT; vSmer: integer);
: Pošlje sporočilo v poljubni smeri (1: sever; 2: vzhod; 3: jug; 4: zahod).

Sprejmi(**var** Sporocilo: SporociloT; **var** izSmeri: integer);
: Prebere sporočilo in ga vrne. V spremenljivki izSmeri dobimo podatek o tem, iz katere smeri je prišlo sporočilo. Poleg vrednosti 1..4 je možna tudi vrednost 0, ki pomeni, da je sporočilo prišlo „od višje sile“. Zagotovljeno je, da se sporočila ne izgubljajo. Poslji vedno počaka, da sosedov Sprejmi prevzame sporočilo; tudi Sprejmi vedno počaka na sporočilo, če še ni prisotno.

Pobarvaj;
: Pobarva točko, s katero je povezan procesor.

Po želji (in potrebi) lahko dopolniš zapis SporociloT, ki je definiran takole:

```
type
  SporociloT = record
    r: real;
    { prostor za tvoje dodatke }
  end;
```

REŠITVE NALOG ZA PRVO SKUPINO

R1993.1.1

Oglejmo si, kaj se dogaja s spremenljivkami a, b in c med izvajanjem programa. N: 1

	a	b	c	izpis
po inicializaciji	c1	c1 + c2	c1 + c2 + c3	
				c1
a := b − a	c2			
b := c − b		c3		
b := a + b		c2 + c3		
				c2
a := b − a	c3			
b := c − b		c1		
b := a + b		c1 + c3		
				c3
a := b − a	c1			
b := c − b		c2		
b := a + b		c1 + c2		
				c1

Vidimo, da po treh izvajanjih zanke zopet dobimo začetno stanje. Program zato neprestano izpisuje ista tri števila — c1, c2 in c3.

N: 1

R1993.1.2

Recimo, da se pri nekem konkretnem nizu s povezave ne sekajo. Prepričajmo se, da v njem nekje nastopata dva enaka znaka eden tik ob drugem. Naj bo a prvi znak niza; če ne pride takoj za njim še drugi a, pač pa neka druga črka b, se mora druga pojavitev b-ja pojaviti še pred drugo pojavitvijo a-ja, drugače bi se povezavi a-a in b-b križali. Če ti dve pojavitvi b-ja ne nastopata skupaj, mora takoj za prvim b-jem priti neka druga črka, recimo c, in druga pojavitev c-ja mora priti še pred drugo pojavitvijo b-ja. Če tako nadaljujemo, vidimo, da sta si pojavitvi vsake naslednje črke bližje skupaj kot pojavitvi prejšnje, to pa se seveda ne more nadaljevati v nedogled, ampak prej ali slej pridemo do para enakih črk, med katerima ni nobene druge črke. No, če bi zdaj ti dve sosednji enaki črki zbrisali iz niza, je v njem zdaj ena povezava manj kot prej in se torej povezave še vedno ne križajo. Zato bi lahko za novi niz opravili enak razmislek in torej spet našli dve sosednji enaki črki ter ju pobrisali; tako bi lahko nadaljevali, dokler ne bi ostal niz čisto prazen.

Po drugi strani pa, če se v opazovanem nizu s kakšni povezavi križata, je s oblike $s_1as_2bs_3as_4bs_5$. Preden bi lahko postopek iz prejšnjega odstavka pobrisal črki a, bi moral pobrisati vse črke iz podniza s_2bs_3, med drugim torej tudi tisti b; toda preden bi lahko pobrisal b-ja, bi moral pobrisati vse iz podniza s_3as_4, torej tudi tisti a. Tako vidimo, da v resnici ne bo mogel pobrisati niti a-jev niti b-jev, saj bi moral pobrisati a-ja pred b-jema in obratno.

Torej lahko za preverjanje, če se kakšni povezavi križata, uporabimo kar gornji postopek; če nam ta uspe niz v celoti pobrisati, križanja povezav v prvotnem ni bilo, sicer pa je bilo. Za učinkovito izvedbo tega postopka si je

koristno pomagati s skladom. Ko bi preiskovali niz s od leve proti desni, bi v nekem trenutku recimo prvič naleteli na dve sosednji enaki črki; s je torej oblike s_1aas_2 in v s_1 ni nikjer dveh sosednjih enakih črk. Če se zdaj odločimo par aa pobrisati, nam ostane niz s_1s_2 in v njem bi načeloma morali pognati enak postopek. Toda odveč bi bilo, če bi začeli s_1s_2 preiskovati spet na začetku, saj že od prej vemo, da v s_1 ni dveh sosednjih enakih črk; prvi možni položaj, kjer bi se lahko pojavili dve sosednji enaki črki, je torej kot zadnja črka s_1 in prva črka s_2. Zato bo naš program na skladu hranil niz s_1; ko prebere naslednjo črko niza s, jo primerja z zadnjo črko s_1 in če sta enaki, oboje pobriše, sicer pa naslednjo črko odloži na vrh sklada in s tem na konec niza s_1. Da nam ne bi bilo treba vnaprej omejevati dolžine sklada s_1 in s tem dolžine vhodnega niza s, bomo sklad izvedli kar kot seznam elementov, povezanih s kazalci.

```
program Povezave(Input, Output);
type CrkaP = ↑CrkaT;
     CrkaT = record C: char; Nasl: CrkaP end;
var Sklad, P: CrkaP; C: char;
begin
  Sklad := nil;
  while not Eoln do begin
    Read(C);
    if Sklad <> nil then if Sklad↑.C = C then { Zbrišimo C s sklada. }
      begin P := Sklad↑.Nasl; Dispose(Sklad); Sklad := P; continue end;
    { Dodajmo C na sklad. }
    New(P); P↑.C := C; P↑.Nasl := Sklad; Sklad := P;
  end; {while}
  if Sklad = nil then WriteLn('Povezave se ne križajo.')
  else WriteLn('Povezave se križajo.');
end. {Povezave}
```

R1993.1.3

N: 2

V zanki se bomo z indeksom `v` sprehajali po zaporedju vratic, obenem pa z indeksom `s` po zaporedju smučarjevih položajev. Ko se postavimo v naslednja vratca, indeks `s` po potrebi povečujemo tako dolgo, dokler ne kaže na zadnji položaj smučarja pred trenutnimi vratci. Potem preverimo, če gre daljica od položaja `s` do položaja `s + 1` res skozi vratca; v ta namen moramo izračunati x-koordinato, ki jo ima smučar, ko doseže y-koordinato vratic, in preveriti, če je ta x-koordinata res med levo in desno koordinato teh vratic. Če gre daljica od (x_1, y_1) do (x_2, y_2), pomeni, da se x-koordinata spremeni za $x_2 - x_1$, ko se y-koordinata spremeni za $y_2 - y_1$. Če nas torej zanima, za koliko se spremeni x-koordinata, če se y-koordinata spremeni za $y_3 - y_1$, nam sklepni račun pokaže, da za $(x_2 - x_1) \cdot (y_3 - y_1)/(y_2 - y_1)$. Pri y-koordinati y_3 dosežemo torej x-koordinato

$$x_3 := x_1 + (x_2 - x_1) \cdot (y_3 - y_1)/(y_2 - y_1).$$

```
program Slalom(Input, Output);

const
  nVratMax = 100;
  nPotMax = 100;

var
  v: integer;          { zaporedna številka vratic }
  s: integer;          { številka točke v smučarjevi poti }
  vy, vxLevi, vxDesni: array [1..nVratMax] of real;   { koordinate vratic }
  sx, sy: array [1..nPotMax] of real;                 { smučarjeve koordinate }
  nVrat: integer;      { število vratic }
  nPot: integer;       { število podanih točk v smučarjevi poti }
  x: real;             { x smučarja med vratci }
  Odstop: boolean;

begin
  ReadLn(nVrat);
  for v := 1 to nVrat do ReadLn(vy[v], vxLevi[v], vxDesni[v]);
  ReadLn(nPot); for s := 1 to nPot do ReadLn(sy[s], sx[s]);
  Odstop := false; v := 1; s := 1;
  if nPot <= 1 then WriteLn('Odstopil na startu.')
  else begin
    while not Odstop and (v <= nVrat) do begin
      while not Odstop and (sy[s + 1] < vy[v]) do begin
        { do naslednjih smučarjevih koordinat ni vratic, premaknemo smučarja }
        s := s + 1;
        if s >= nPot then begin WriteLn('Odstopil.'); Odstop := true end;
      end; {while}
      { prevozi vratca v }
      while not Odstop and (v <= nVrat) and (sy[s + 1] >= vy[v]) do begin
        { preverimo, če res pelje skozi vratca }
        x := sx[s] + (sx[s + 1] - sx[s]) / (sy[s + 1] - sy[s]) * (vy[v] - sy[s]);
        if (x >= vxLevi[v]) and (x <= vxDesni[v]) then v := v + 1
        else begin WriteLn('Izpustil vratca ', v, '.'); Odstop := true end;
      end; {while}
    end; {while}
    if not Odstop then WriteLn('Smučar je uspešno prevozil progo.');
  end; {if}
end. {Slalom}
```

N: 2

R1993.1.4

Znake, ki jih beremo, lahko sproti izpisujemo, razen če niso presledki. Pri teh si le zapomnimo, da so se pojavili (spremenljivka Presledek v spodnjem programu). V spremenljivki Locilo si zapomnimo, če je bil zadnji doslej prebrani ne-presledek slučajno pika ali vejica. Če res naletimo na piko ali vejico, presledkov pred njo tako ali tako ne smemo izpisati, ločilo pa si zapomnimo v spremenljivki Locilo. Če naletimo na

kak drug znak, ki ni presledek, pa se moramo le še odločiti, koliko presledkov izpisati pred njim: med piko in veliko začetnico sta vedno dva, za vejico je točno eden, v ostalih primerih pa izpišemo enega ali pa nobenega, odvisno od vrednosti spremenljivke Presledek.

```
program Besedilo(Input, Output);

var Znak, Locilo: char;
    Presledek: boolean;

  function GetCh(var ch: char): boolean; external;
  procedure PutCh(ch: char); external;

begin
  Locilo := ' ';
  Presledek := false;
  while GetCh(Znak) do begin
    if Znak = ' ' then Presledek := true
    else if (Znak = '.') or (Znak = ',') then begin
      Locilo := Znak;
      Presledek := false;
      PutCh(Znak);
    end else begin
      if (Locilo = '.') and (Znak in ['A'..'Z']) then
        begin PutCh(' '); PutCh(' ') end
      else if Locilo = ',' then PutCh(' ')
      else if Presledek then PutCh(' ');
      Locilo := ' ';
      Presledek := false;
      PutCh(Znak);
    end; {if}
  end; {while}
end. {Besedilo}
```

REŠITVE NALOG ZA DRUGO SKUPINO

R1993.2.1

N: 3

Program najprej prebere n števil v tabelo t, nato v funkciji Mac obrne vrstni red števil v tabeli in jih nazadnje v obrnjenem vrstnem redu izpiše.

V nalogi je zanimiva predvsem funkcija Mac, ki obrne vrstni red števil. Osnovno idejo smo prenesli iz prologovega programa naive-reverse:[1]

[1] Gl. npr. Ivan Bratko, *Prolog Programming for Artificial Intelligence*, 3. izd. (2001), razd. 8.5.4, str. 188. Opisani prologov program je „naiven“ zato, ker moramo iti pri običajnih prologovih seznamih prek celega seznama, preden lahko dodamo na koncu nek nov element. Naš postopek za obračanje seznama dodaja elemente na konec izhodnega seznama enega po enega, zato ima z obračanjem seznama n elementov kar $O(n^2)$ dela; se pa zato včasih

```
reverse([], []).              % prazen seznam obrnemo v prazen seznam
reverse([X | L1], L3) :-      % vhodni seznam razdelimo na
                              %    glavo X in preostanek seznama L1
  reverse(L1, L2),            % seznam L1 obrnemo v seznam L2
  append(L2, [X], L3).        % obrnjenemu seznamu L2 dodamo na konec element X
```

Na podoben način deluje funkcija Mac. Tabelo najprej razdeli na trenutni element (shranimo ga v x) in preostanek, ki ga obrnemo s klicem Mac(i + 1). Na koncu shranjeni element postavimo na ustrezno mesto (tega spotoma izračuna rekurzivni klic Mac(i + 1)).

N: 3

R1993.2.2

Naloga bi lahko kaj bolj natančno povedala, kaj naredi risalnik, ko prejme ukaz o zamenjavi peresa: Ali pred premikom do vrtiljaka s peresi (*carousel*) dvigne pero, če je bilo prej spuščeno? Ali se, ko vzame novo pero, premakne na položaj, kjer je bil, ko je prejel ukaz za zamenjavo peresa? Ali pa le obstane na nekem vnaprej definiranem položaju? Ali novo pero spusti, če je bilo spuščeno tudi staro pero pred zamenjavo? Te reči moramo poznati, če hočemo pošteno simulirati vsa možna zaporedja ukazov. Lahko bi na primer predpostavili, da ukazi, ki jih dobivamo z računalnika, pred zamenjavo peresa vsebujejo tudi ukaz za dvig peresa, po zamenjavi pa se eksplicitno premaknejo na nek nov položaj in šele tam spustijo pero. Če pa kaj od tega manjka, bi morali načeloma za te reči poskrbeti mi. Recimo za primer, da ima program nekaj ukazov za eno pero, nato preklopi na drugo, spet nekaj riše, preklopi nazaj na prvo in nariše še nekaj črt. Če bi zdaj mi prvo in tretjo skupino ukazov hoteli združiti (ker se obe nanašata na isto pero), je npr. mogoče, da se prva ne konča z ukazom za dvig peresa, ker je uporabnik vedel, da bo naslednji ukaz (za preklop na drugo pero) itak implicitno tudi dvignil pero; tretja skupina ukazov pa se mogoče začne s premikom, ki bi se opravil z dvignjenim peresom, če velja, da je pero po zamenjavi vedno dvignjeno; v tem primeru bi morali mi, ko združujemo prvo in tretjo skupino ukazov, vmes vriniti ukaz za dvig peresa.

Predpostavili bomo, da risalnik, ko dobi ukaz za zamenjavo peresa, najprej dvigne pero (če je bilo prej spuščeno), se odpelje do vrtiljaka s peresi, odloži dosedanje pero, vzame novo in nato obstane z dvignjenim novim peresom na nekem fiksnem in vnaprej znanem položaju. To je smiselno, saj prav gotovo nihče, ko pošlje risalniku ukaz za zamenjavo peresa, od njega ne pričakuje, naj nariše spotoma še črto v stari barvi od trenutnega položaja do vrtiljaka in nato v novi barvi od vrtiljaka nazaj do istega položaja ali kaj podobno neumnega. Predpostavili bomo tudi, da v zaporedju ukazov, ki prihajajo z računalnika, takoj za ukazom za zamenjavo peresa vedno pride ukaz za premik. Tudi to je precej smiselno, saj je po zamenjavi peresa trenutni položaj nekje pri vrtiljaku in verjetnost, da hoče uporabnik risati črto ravno od tam, je neznatno majhna.

uporablja kot benchmark pri merjenju hitrosti interpreterjev prologa.

Vprašanje je še, kaj risalnik naredi, če dobi ukaz, naj zamenja pero s tistim, ki ga že zdaj drži. Glede tega bomo predpostavili, da bi ravnal enako, kot če bi moral res vzeti neko drugo pero; skratka, da bo dvignil pero, se odpeljal do vrtiljaka, ga odložil in takoj spet pobral.

Naš program bo ukaze, ki prihajajo z računalnika, odlagal v nek vmesni pomnilnik (v vrsto), razen če se ne nanašajo na pero, ki ga risalnik trenutno drži v roki — take pa lahko pošljemo naravnost risalniku. Zato moramo ločeno voditi podatek o peresu, ki je res v risalniku (PeroRisalnika), in o tistem, za katerega računalnik misli, da je v risalniku (PeroRacunalnika — to je pero, ki bi tudi res bilo v risalniku, če se ne bi vmešal naš program). Da poraba pomnilnika ne bo naraščala v nedogled, se dogovorimo za neko največjo dovoljeno velikost vrste (MaxVrsta ukazov); ko se vrsta napolni, bomo zamenjali pero in takoj izvedli tiste ukaze iz vrste, ki se nanašajo na novo pero. Da bi bilo treba peresa menjati čim redkeje, si vedno izberimo tako pero, na katerega se nanaša največ ukazov iz vrste, tako da se bo vrsta čim bolj spraznila. Zato bomo v tabeli Kolikokrat za vsako pero hranili podatek o tem, kolikokrat se pojavlja v vrsti.

Podprogram ObdelajUkaz izvede dani ukaz, če se nanaša na trenutno pero, sicer pa ga doda v vrsto. Podprogram PocistiPero pa zamenja pero v risalniku in še enkrat prebere vse ukaze iz vrste ter jih pošilja skozi ObdelajUkaz. Ob tem se bodo torej vsi ukazi, ki se nanašajo na novo pero risalnika, tudi res izvedli, ostali pa bodo prišli nazaj v vrsto. Posebej opozorimo na dejstvo, da je pri spodnjem programu ves čas, razen v okviru izvajanja podprograma PocistiPero, na začetku vrste nek ukaz za menjavo peresa.[2] Zato ni pomembno, kakšna je vrednost spremenljivke PeroRacunalnika tik pred klicem podprograma PocistiPero; ta bo v vsakem primeru pravilno izvedel natanko tiste ukaze, ki se nanašajo na novo barvo peresa, na koncu pa bo imela tudi PeroRacunalnika spet tako vrednost kot na začetku (torej vrednost, ki je ostala po zadnjem ukazu

[2]Res: (1) Na začetku, do prvega ukaza za menjavo peresa, predpostavimo, da hoče uporabnik risati s peresom, ki je že od prej v risalniku, pa kakršne koli barve že pač je; zato postavimo na začetku tako PeroRacunalnika kot PeroRisalnika na 0 in do prvega ukaza za menjavo peresa sproti izvajamo vse ukaze, tako da v vrsto ne pride nič. Ko torej ukaze začnemo dodajati v vrsto, se prav gotovo v njej prvi znajde nek ukaz za menjavo peresa. (2) Kasneje pa, ko ukaze jemljemo iz vrste, se to vedno počne le znotraj podprograma PocistiPero, ta pa iz vrste pobere vse ukaze in jih posreduje podprogramu ObdelajUkaz. Tu lahko ločimo dve možnosti: ali se ukaz za menjavo peresa z začetka vrste ujema s peresom NovoPero ali pa ne. (2a) Če se ujema, bi ObdelajUkaz takoj izvedel tega in vse nadaljnje ukaze do prve naslednje menjave peresa, tako da bi se vsebina vrste nato spet začela z ukazom za menjavo peresa. (2b) Če pa se ne ujema, bi ta ukaz takoj spet dodali v vrsto in ko bi zanka v PocistiPero prišla do konca, bi bil ta ukaz, torej nek ukaz za menjavo peresa, spet na začetku vrste. (3) Do težave bi lahko prišlo le še pri dodajanju v prazno vrsto, toda če je vrsta prazna, pomeni, da smo ali šele na začetku ali pa se je ravnokar izvedla PocistiPero in se je pred tem cela vrsta nanašala na pero v eni sami barvi. V vsakem primeru je bilo torej po tistem pero računalnika enako peresu tiskalnika, tako da, če se je zdaj pojavila potreba po tem, da bi v vrsto nekaj dodali, pomeni, da se je moralo pero računalnika spremeniti, torej smo dobili tudi ukaz za menjavo peresa in ravno ta bo šel prvi v vrsto.

za menjavo peresa), kot je tudi prav.

Če hoče ObdelajUkaz dodati ukaz v vrsto, pa opazi, da je ta že polna, pokliče PocistiPero, kar bo iz vrste zanesljivo odstranilo vsaj en ukaz. Ko PocistiPero opravi svoje delo, lahko ObdelajUkaz obdela trenutni ukaz na enak način, kot da bi ga ravnokar šele prejel: zaradi klica PocistiPero je zdaj mogoče pero računalnika isto kot pero risalnika in lahko pošljemo novi ukaz naravnost risalniku, če pa ga bo treba dodati v vrsto, je v njej po vrnitvi iz PocistiPero zagotovo dovolj prostora še vsaj za en ukaz. Ko dodamo ukaz v vrsto, seveda ne smemo pozabiti povečati števca, koliko ukazov v vrsti se nanaša na to pero (Kolikokrat[PeroRacunalnika]).

Če dobi ObdelajUkaz ukaz za menjavo peresa z istim, ki je že zdaj v risalniku, namesto tega pošlje risalniku le ukaz za dvig peresa. V skladu s prej navedenimi predpostavkami si namreč mislimo, da bi risalnik ukaz za zamenjavo res izvedel in s tem po nepotrebnem tratil čas, zato mu tega ukaza raje ne pošiljamo; po drugi strani pa bi po zamenjavi veljalo, da je pero dvignjeno, zato ga moramo tudi mi zdaj dvigniti: pero je bilo namreč doslej mogoče spuščeno in ker ukazu za zamenjavo verjetno sledi ukaz za premik, ne bi bilo dobro, če bi risalnik vlekel črto do novega položaja (saj se uporabnik zanaša na to, da se je pred tem izvedel ukaz za menjavo peresa, ki je pustil pero dvignjeno). Zanašamo se na to, da je ukaz za dvig peresa hiter, zato se ne bomo obremenjevali s tem, če ga kdaj pa kdaj slučajno izvedemo brez potrebe (ker je bilo pero že itak dvignjeno) — drugače bi morali poleg podatka o trenutnem peresu risalnika vzdrževati pač tudi podatek o trenutnem položaju peresa (dvignjeno ali spuščeno).

Ko dobimo od računalnika ukaz za konec risbe, moramo, preden ga pošljemo risalniku, še izprazniti vrsto. V ta namen si tudi lahko pomagamo s podprogramom PocistiPero, tako da tudi zdaj ne bomo menjali peres večkrat, kot je nujno potrebno (po enkrat za vsako pero, ki ga sploh potrebujemo).

Program bi lahko še malo izboljšali, da bi pospeševal izvajanje kakšnih patoloških zaporedij ukazov (npr. takih, ki pridno menjajo peresa, vmes pa le mahajo z njimi po zraku, ne da bi jih res spustili na papir in z njimi kaj narisali — očitno je, da bi lahko tako menjavo peresa in vse pripadajoče gibe čisto opustili).

Še ena izboljšava bi bila, da bi imeli za vsako pero ločeno vrsto oz. poseben seznam ukazov za to pero. Pri tem bi lahko še vedno hranili podatke o dolžini teh seznamov in pazili, da skupna dolžina ne preseže neke dogovorjene meje. Lepo pri tej rešitvi bi bilo, da nam ob klicu PocistiPero ne bi bilo treba pregledati vseh ukazov v vrsti, pač pa le tiste, ki se nanašajo na izbrano pero.

```
program KrmiljenjeRisalnika;
const StPeres = 10;
type OperacijaT = (opPeroGor, opPeroDol, opPero, opPremik, opKonec);
     UkazT = record
```

```
        case Kaj: OperacijaT of
          opPero: (n: integer);
          opPremik: (x, y: integer);
      end;

  procedure BeriUkaz(var Ukaz: UkazT); external;
  procedure PosljiUkaz(Ukaz: UkazT); external;

const MaxVrsta = 100;
var Vrsta: array [0..MaxVrsta - 1] of UkazT;
    Glava, DolVrste: integer;

  procedure VrstaVzemi(var Ukaz: UkazT);
  begin
    Ukaz := Vrsta[Glava]; Glava := Glava + 1; DolVrste := DolVrste - 1;
    if Glava = MaxVrsta then Glava := 0;
  end; {VrstaVzemi}

  procedure VrstaDodaj(Ukaz: UkazT);
  begin
    Vrsta[(Glava + DolVrste) mod MaxVrsta] := Ukaz;
    DolVrste := DolVrste + 1;
  end; {VrstaDodaj}

var Kolikokrat: array [1..StPeres] of integer;
    PeroRisalnika, PeroRacunalnika: integer;

  procedure PocistiPero(NovoPero: integer); forward;

  procedure ObdelajUkaz(Ukaz: UkazT);
  var Pero, NajPero: integer;
  begin
    if Ukaz.Kaj = opPero then { Zapomnimo si novo pero. }
      PeroRacunalnika := Ukaz.n;
    if PeroRacunalnika = PeroRisalnika then begin
      if Ukaz.Kaj = opPero then Ukaz.Kaj := opPeroGor;
      PosljiUkaz(Ukaz);
    end else begin
      if (DolVrste = MaxVrsta) then begin
        { Za katero pero imamo največ ukazov? }
        NajPero := 1;
        for Pero := 2 to StPeres do
          if Kolikokrat[Pero] > Kolikokrat[NajPero] then NajPero := Pero;
        { Preklopimo na to pero in takoj izvedimo ukaze zanj. }
        PocistiPero(NajPero);
        ObdelajUkaz(Ukaz);
      end else begin
        VrstaDodaj(Ukaz);
```

```
          Kolikokrat[PeroRacunalnika] := Kolikokrat[PeroRacunalnika] + 1;
        end; {if}
      end; {if}
    end; {ObdelajUkaz}

    procedure PocistiPero(NovoPero: integer);
    var Pero, StUkazov: integer; Ukaz: UkazT;
    begin
      { Ukaze bomo na novo prebirali in posiljali podprogramu ObdelajUkaz.
        Zato postavimo števce na 0, saj jih bo ObdelajUkaz ponovno povečeval. }
      for Pero := 1 to StPeres do Kolikokrat[Pero] := 0;
      { Preklopimo na to pero. }
      Ukaz.Kaj := opPero; Ukaz.n := NovoPero; PosljiUkaz(Ukaz);
      PeroRisalnika := NovoPero;
      { Pošljimo vse ukaze iz vrste še enkrat skozi podprogram
        ObdelajUkaz. Tisti, ki se nanašajo na novo pero, se bodo
        pri tem zares izvedli, ostali pa bodo prišli nazaj v vrsto. }
      StUkazov := DolVrste; Pero := 0;
      while StUkazov > 0 do begin
        VrstaVzemi(Ukaz);
        ObdelajUkaz(Ukaz);
        StUkazov := StUkazov - 1;
      end; {while}
    end; {PocistiPero}

  var Ukaz: UkazT; Pero: integer;
  begin {KrmiljenjeRisalnika}
    for Pero := 1 to StPeres do Kolikokrat[Pero] := 0;
    PeroRacunalnika := 0; PeroRisalnika := 0;
    while true do begin { Neskončna zanka. }
      BeriUkaz(Ukaz);
      while Ukaz.Kaj <> opKonec do begin
        ObdelajUkaz(Ukaz);
        BeriUkaz(Ukaz);
      end; {while}
      { Risbe je konec; pošljimo risalniku še ukaze, ki čakajo v vrsti. }
      for Pero := 1 to StPeres do
        if Kolikokrat[Pero] > 0 then PocistiPero(Pero);
      PosljiUkaz(Ukaz); { Pošljimo mu še ukaz za konec risbe. }
    end; {while}
  end. {KrmiljenjeRisalnika}
```

N: 4

R1993.2.3

Najhitrejša rešitev naloge je naslednja: začnemo v levem spodnjem oglišču. Če je trenutni element manjši od iskanega, se premaknemo v desno. Če je večji, se premaknemo navzgor. Če je element enak iskanemu, je vseeno, ali se premaknemo navzgor ali v desno. Postopek skoraj brez sprememb deluje tudi na pravokotni tabeli.

```
var t: array [1..n, 1..n] of integer;

procedure Poisci(Element: integer);
var x, y, Stevec: integer;
begin
  x := 1;
  y := n;
  Stevec := 0;
  while (x <= n) and (y >= 1) do begin
    if Element > t[x, y] then x := x + 1
    else if Element < t[x, y] then y := y - 1
    else begin
      WriteLn('Našel sem iskano vrednost ', Element,
              ' v celici t[', x, ',', y, '].');
      Stevec := Stevec + 1;
      x := x + 1;
    end; {if}
  end; {while}
  WriteLn('Število najdenih primerkov vrednosti ', Element,
          ': ', Stevec, '.');
end; {Poisci}
```

O pravilnosti tega postopka se lahko prepričamo z naslednjo invarianto: na začetku vsakega izvajanja glavne zanke **while** v gornjem programu vsebuje spremenljivka Stevec število takih pojavitev vrednosti Element, ki imajo ali $X < x$ ali $Y > y$. Na začetku to očitno velja, saj pri $x = 1$ in $y = n$ ustreznih položajev (X, Y) sploh ni, Stevec pa je enak 0. Če potem opazimo, da je element (x, y) manjši od iskane vrednosti, pomeni, da so tudi tisti nad njim v istem stolpcu manjši od te vrednosti (saj vemo, da elementi dol po stolpcu naraščajo); torej lahko povečamo x za 1, pa bo invarianta še vedno veljala. Podobno pa, če opazimo, da je element (x, y) večji od iskane vrednosti, so tisti desno od njega v isti vrstici tudi večji od nje, tako da lahko odpišemo celo vrstico: ko zmanjšamo y za 1, invarianta še vedno velja. Ob koncu izvajanja podprograma je $x \geq n$ ali pa $y \leq 1$ in pogoju „$X < x$ ali $Y > y$“ ustrezajo vsi položaji (X, Y) v tabeli, tako da je v Stevec očitno res pravo število pojavitev vrednosti Element v celi tabeli.

R1993.2.4

N: 4

Spodnji program hrani v spremenljivki dt1 dolžino kratkega piska, kot jo je ocenil iz dosedanjih meritev. Naslednji pisk naj bi bil torej dolg ali dt1 ali pa trikrat toliko. Ker naloga pravi, da se lahko dolžina piska počasi tudi spreminja, bomo dolžino naslednjega piska primerjali z dvakratnikom enote dt1 — če je pisk krajši, si ga bomo razlagali kot kratek pisk, drugače pa kot dolg pisk. V vsakem primeru lahko zdaj iz tega piska ocenimo novo enotsko dolžino kratkega piska (ndt1) in jo potem skombiniramo z dosedanjo (dosedanja ima težo Utez, nova pa težo 1); to nam zagotovi,

da referenčne dolžine dt1 ne bomo prehitro spremenili le zaradi enega ali dveh neobičajno dolgih ali kratkih piskov, če pa so takšne spremembe trajnejše, bo dt1 sčasoma lahko prišla poljubno blizu nove dolžine piska. Ko smo uspešno prepoznali pisk, moramo izmeriti še trajanje pavze za njim, da ugotovimo, če gre morebiti za konec črke ali celo konec besede.

```
program Morse(Input, Output);

const Pika = '.'; Crta = '_';

type CasT = integer;

var
  dt: CasT;      { izmerjeni časovni interval }
  dt1: CasT;     { referenčna osnovna enota — trajanje pike }
  ndt1: CasT;    { osnovna enota, kot smo jo izmerili pri zadnjem znaku }
  Utez: integer; { stopnja nespremenljivosti referenčne enote }
  p: boolean; Znak: char;

  procedure Start; external;
  function Cas: CasT; external;
  function Piska: boolean; external;

begin
  repeat until not Piska;
  dt1 := 0; Utez := 0;
  while true do begin
    repeat until Piska; { počakamo začetek piska }
    Start;
    repeat until not Piska; { počakamo konec piska }
    dt := Cas;
    Start;
    if dt < 2 * dt1 { primerjamo trajanje piska z referenco }
      then begin Znak := Pika; ndt1 := dt end{ pika traja eno enoto }
      else begin Znak := Crta; ndt1 := dt div 3 end; { črtica traja tri enote }
    Write(Znak);
    dt1 := (Utez * dt1 + ndt1) div (Utez + 1); { prilagodimo časovno enoto }
    if Utez < 5 then Utez := Utez + 1; { referenčna enota postaja zanesljivejša }
    { počakamo začetek piska ali dovolj dolgo, da velja konec črke }
    repeat p := Piska until p or (Cas > 2 * dt1);
    if not p then Write(' ');
    { počakamo začetek piska ali dovolj dolgo, da velja konec besede }
    repeat p := Piska until p or (Cas > 5 * dt1);
    if not p then WriteLn;
  end; {while}
end. {Morse}
```

Da bi pravilno prepoznali tudi znake na začetku telegrama (ko je vrednost referenčne časovne enote še neznana), moramo trajanja piskov shranjevati in

odlašati z njihovo kategorizacijo v pike ali črtice ter z izpisom tako dolgo, da se med shranjenimi časi pojavita dve kategoriji trajanj — kratko in dolgo. Na podlagi teh kategorij lahko določimo začetno vrednost časovne enote (reference) in glede na njo kategoriziramo in izpišemo shranjene znake. Nadaljnji postopek je enak tukaj sprogramiranemu, shranjevanje ni več potrebno.

Mimogrede, pike in črtice v primeru iz besedila naloge pomenijo: „SPREJEM MORSEJEVE ABECEDE“.

REŠITVE NALOG ZA TRETJO SKUPINO

R1993.3.1

Oglejmo si, kaj se dogaja s spremenljivkami a, b in c med izvajanjem programa: N: 5

	a	b	c	izpis
po inicializaciji	c1	c1 **xor** c2	c1 **xor** c2 **xor** c3	
				c1
a := b **xor** a	c2			
b := c **xor** b		c3		
b := a **xor** b		c2 **xor** c3		
				c2
a := b **xor** a	c3			
b := c **xor** b		c1		
b := a **xor** b		c1 **xor** c3		
				c3
a := b **xor** a	c1			
b := c **xor** b		c2		
b := a **xor** b		c1 **xor** c2		
				c1

Pri tem smo upoštevali eno najlepših lastnosti operacije **xor**: a **xor** a = 0.

Vidimo, da po treh izvajanjih zanke zopet dobimo začetno stanje. Program zato neprestano izpisuje ista tri števila — c1, c2 in c3.

R1993.3.2

Različnih pristopov k reševanju te naloge je zelo veliko. Razložili bomo enega najhitrejših, ki je žal rahlo razsipen s prostorom. Osnovnih principov, ki smo jih uporabili tu, ni težko prenesti v manj potraten, a počasnejši algoritem. N: 6

Resnično pomembna v spodnjem programu sta le podprograma PreberiUredi in Izpisi. Prvi prebere podatke o daljicah in vsako posebej vstavi v seznam. Drugi se sprehodi po tem seznamu in izpiše daljice v pravem vrstnem redu.

Namesto da bi v seznam vstavljali daljice, bomo raje vstavljali njihova krajišča. Ko preberemo novo daljico, poiščemo obe krajišči v seznamu. Če

krajišča ne najdemo, ga v seznam vstavimo. Nato v seznam zapišemo še povezavo med njima.

Iskanje in vstavljanje v spodnjem programu zelo učinkovito počne prodprogram Vstavi. Za razumevanje je malce težji, ker uporablja razpršene tabele, prav tako dober (le počasnejši) pa bi bil kar navaden linearni seznam.

Na koncu se le zapeljemo čez povezave, ki smo jih naredili med gradnjo seznama, in izpišemo vse točke. Pazimo le, da med dvema možnima nadaljevanjema (vsako krajišče ima dve povezavi) izberemo pravo (napačna povezava je seveda tista, ki smo jo ravnokar uporabili).

```
program Daljice(Input, Output);

const Najvec = 1020;
type
  KoordT = integer;
  EnaT = record
    x, y: KoordT;
    Link1, Link2: integer;
  end;
  TockeT = array [0..Najvec] of EnaT;
var
  Tocke: TockeT;
  Zadnji: integer;

  procedure Init;
  var i: integer;
  begin
    for i := 0 to Najvec do with Tocke[i] do Link1 := -2;
  end; {Init}

  function Preberi(var x1, y1, x2, y2: KoordT): boolean;
  begin
    if Eof then Preberi := false
    else begin ReadLn(x1, y1, x2, y2); Preberi := true; end;
  end; {Preberi}

  function Vstavi(x1, y1: KoordT): integer;
  var Indeks, Prvi, Dodatek: integer;
      Nasel: boolean;
  begin
    Indeks := (x1 + y1) mod (Najvec + 1);
    if Tocke[Indeks].Link1 <> -2 then begin
      Nasel := (Tocke[Indeks].x = x1) and (Tocke[Indeks].y = y1);
      if not Nasel then begin
        Prvi := Indeks;
        Dodatek := 1 + ((x1 + y1) mod (Najvec - 1));
        repeat
```

```
        Indeks := (Indeks + Dodatek) mod (Najvec + 1);
        Nasel := (Tocke[Indeks].x = x1) and (Tocke[Indeks].y = y1);
      until (Tocke[Indeks].Link1 = −2) or (Indeks = Prvi) or Nasel;
      if (Indeks = Prvi) and (not Nasel) then
        begin WriteLn('Tabela je polna!'); Halt end;
    end; {if}
  end
  else Nasel := false;
  if not Nasel then with Tocke[Indeks] do begin
    Link1 := −1; Link2 := −1;
    x := x1; y := y1;
  end; {if}
  Vstavi := Indeks;
end; {Vstavi}

procedure PreberiUredi;
var xz, yz, xk, yk: KoordT;
    tz, tk: integer;
begin
  Zadnji := −1;
  while Preberi(xz, yz, xk, yk) do begin   { preberemo podatke o daljici }
    tz := Vstavi(xz, yz);                  { poiščemo/vstavimo podatke o prvem }
    tk := Vstavi(xk, yk);                  { in drugem krajišču }
    with Tocke[tz] do                      { povežemo prvo krajišče z drugim }
      if Link1 = −1 then Link1 := tk else Link2 := tk;
    with Tocke[tk] do                      { in drugo s prvim }
      if Link1 = −1 then Link1 := tz else Link2 := tz;
    Zadnji := tk;                          { zapomnimo si zadnje vstavljeno krajišče }
  end; {while}
end; {PreberiUredi}

procedure Izpisi;
var Tren, Nas: integer;
begin
  if Zadnji <> −1 then begin
    Tren := Zadnji;                        { izpisovati začnemo pri zadnjem krajišču }
    Nas := Tocke[Tren].Link1;
    if Nas <> −1 then repeat
      WriteLn('(', Tocke[Tren].x, ',', Tocke[Tren].y,
              ') - (', Tocke[Nas].x, ',', Tocke[Nas].y, ')');
      { premik v naslednje krajišče; pazimo, da nadaljujemo v pravi smeri }
      if Tren = Tocke[Nas].Link1
        then begin Tren := Nas; Nas := Tocke[Nas].Link2 end
        else begin Tren := Nas; Nas := Tocke[Nas].Link1 end;
    until (Nas = −1) or (Tren = Zadnji);
    if Nas = −1 then WriteLn('Link ni zaprt!');
  end; {if}
```

```
  end; {Izpisi}

begin {Daljice}
  Init;
  PreberiUredi;
  Izpisi;
end. {Daljice}
```

Podprogram Vstavi uporablja tabelo Tocke kot „zaprto“ razpršeno tabelo. Ko naletimo na neko točko (x, y) (kot krajišče neke daljice), si želimo čim hitreje ugotoviti, če smo na to točko že naleteli pri kakšni drugi daljici (kajti če je tako, moramo ti dve daljici povezati). Da nam ne bi bilo treba preiskovati cele tabele in primerjati vseh točk v njej z našo novo točko (x, y), se dogovorimo, da bomo točko (x, y) v tabeli vedno (če se bo le dalo) hranili na indeksu $(x+y) \bmod (n+1)$, pri čemer je $n+1$ dolžina naše tabele (indekse pa štejemo od 0 do n). Težava nastopi zaradi možnosti, da bi morali več točk hraniti na istem indeksu (na primer: (x, y) in (y, x), pa tudi $(x+1, y-1)$ in tako naprej). V takem primeru se premikajmo naprej po tabeli, dokler ne pridemo do prazne celice; gornji program dela skoke po $1 + ((x + y) \bmod (n - 1))$. Lepo je, če pri tem skakanju vemo, da bomo preizkusili vse možne indekse, preden bomo prišli nazaj na začetnega in nad vsem skupaj obupali; gornji program ima tabelo z 1021 elementi, kar je dobro, ker je 1021 praštevilo, tako da je dolžina skoka gotovo tuja številu 1021 in bomo res pristali na vseh možnih indeksih, preden bomo prišli nazaj na prvega. Kakorkoli že, med tem skakanjem po tabeli bomo prej ali slej prišli do prazne celice (razen če ni tabela že čisto polna), kamor lahko zdaj vpišemo svojo novo točko; lahko pa med tem skakanjem to točko tudi najdemo (in se s tem izkaže, da je točka že bila v tabeli, le da je takrat, ko smo jo vanjo dodajali, že nismo več mogli vpisati na prvotni indeks, pač pa smo takrat s skakanjem po tabeli prišli do neke prazne celice in jo vpisali tja).

Vsaka celica tabele ima prostor za indeksa Link1 in Link2, ki bosta na koncu pri vsaki točki kazala na indeksa drugih dveh krajišč tistih dveh daljic, ki se stikata v tej točki. To, da je celica še prazna, prepoznamo po tem, da imata Link1 in Link2 vrednost -2, če pa smo vanjo že vpisali neko točko, ki pa je še nismo povezali z drugima dvema krajiščema, sta Link2 in mogoče tudi Link1 enaka -1.

N: 6

R1993.3.3

Osnovna ideja za nalogo izhaja iz sistema Dynascope, katerega avtor je Rok Sosič. Sistem je namenjen predvsem opazovanju poljubnih programov med njihovim izvajanjem. Ena od funkcij je tudi vračanje izvajanja opazovanega programa na neko predhodno točko. Dejanska izvedba te funkcije je podobna kot v tem programu.

Pred izvajanjem vsakega ukaza moramo shraniti vrednosti tistih elementov računalnika, ki se pri tem ukazu spreminjajo (zapis PopravekT). Sem vsekakor sodi vrednost programskega števca (P), mogoče pa tudi vrednost akumulatorja

(A) ali pa neke pomnilniške celice (M in njen Naslov). Kaj od tega je res treba shraniti, je odvisno od ukaza. Pri podprogramu UkazNazaj je treba le vpisati stare vrednosti (polje TipPopravka pove, katere vrednosti v zapisu so veljavne). Zapise hranimo v tabeli Sled, ki jo pravzaprav uporabljamo kot sklad — PredUkazom dodaja zapise na konec, UkazNazaj pa jih od tam briše.

```
program Dynascope(Output);

const MaxDolzinaSledi = 100; { največja dolžina shranjene sledi izvajanja }
type
  BesedaT = 0..65535;
  UkazT = (LDA, LDAI, STA, STAI, MSUB, MSUBI, JPOS, JPOSI);
  tpT = (tpAP, tpMNP, tpP);
  PopravekT = record                    { podatki, ki jih spreminjajo ukazi }
    P: BesedaT;                         { vedno shranimo programski števec }
    case TipPopravka: tpT of
      tpAP: (A: BesedaT);               { shranimo tudi akumulator }
      tpMNP: (M, Naslov: BesedaT);      { shranimo naslov in vsebino pomn. celice }
      tpP: ( );                         { shranimo le programski števec }
  end;
var
  M: array [BesedaT] of BesedaT;        { pomnilnik }
  P: BesedaT;                           { programski števec }
  A: BesedaT;                           { akumulator }
  Sled: array [1..MaxDolzinaSledi] of PopravekT; { sled izvajanja }
  DolzinaSledi: integer;                { število shranjenih ukazov v tabeli Sled }

  procedure Napaka(Sporocilo: string); external;

procedure PredUkazom;
var Korak: PopravekT;
    Ukaz, Operand: BesedaT;
begin
  Ukaz := M[P]; Operand := M[P + 1];
  Korak.P := P; { vsak ukaz spremeni vsaj P }
  case Ukaz of
    Ord(LDA), Ord(LDAI), Ord(MSUB), Ord(MSUBI): { ukazi spremenijo A in P }
      begin Korak.TipPopravka := tpAP; Korak.A := A end;
    Ord(STA), Ord(STAI): begin { ukaza spremenita pomnilniško celico in P }
      Korak.TipPopravka := tpMNP;
      if Ukaz = Ord(STA) then Korak.Naslov := Operand
      else Korak.Naslov := M[Operand];
      Korak.M := M[Korak.Naslov];
    end;
    Ord(JPOS), Ord(JPOSI): Korak.TipPopravka := tpP; { ukaza spremenita le P }
    else Napaka('neznan ukaz');
  end; {case}
```

```
  if DolzinaSledi >= MaxDolzinaSledi then Napaka('sled predolga');
  DolzinaSledi := DolzinaSledi + 1;          { podaljšamo sled izvajanja }
  Sled[DolzinaSledi] := Korak;               { spremembo stanja dodamo v sled }
end; {PredUkazom}

procedure UkazNazaj;
var Korak: PopravekT;
begin
  if DolzinaSledi < 1 then Napaka('sledi ni');
  Korak := Sled[DolzinaSledi];               { vzamemo zadnjo spremembo stanja }
  DolzinaSledi := DolzinaSledi - 1;          { skrajšamo sled izvajanja }
  P := Korak.P;                              { vedno popravimo P }
  case Korak.TipPopravka of
    tpAP: A := Korak.A;                      { popravimo A }
    tpMNP: M[Korak.Naslov] := Korak.M;       { popravimo pomnilniško celico }
    tpP: ;                                   { P smo že popravili }
  end; {case}
end; {UkazNazaj}
```

Mogoče bi bilo koristno program spremeniti tako, da v primeru, ko je tabela Sled že polna, ne bi javil napake, pač pa bi uporabljal Sled kot krožno tabelo in s podatki o novem ukazu preprosto povozil najstarejši zapis sledi. Tako bi sled vedno omogočala sledenje nazaj za zadnjih DolzinaSledi ukazov.

N: 7

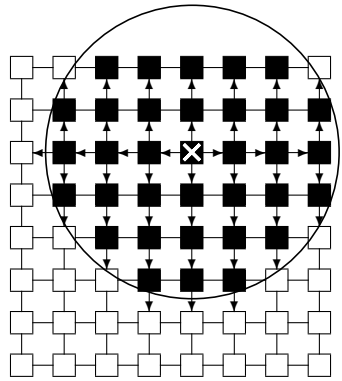
R1993.3.4

Uporabili bomo Pitagorov izrek. Točka pripada krogu, če velja $x^2 + y^2 \leq r^2$. Potrebujemo torej koordinati x in y vsake točke. Pare koordinat si procesorji sporočajo med seboj, zato smo v zapis SporociloT dodali polji x in y.

Procesorju v središču kroga dodelimo koordinati $(0, 0)$. Vsak procesor pošlje svoje koordinate sosedom, ki s pomočjo sprejetih koordinat in smeri, iz katere je sporočilo prispelo, izračunajo svoj položaj. r^2 izračuna samo procesor v središču kroga.

Sporočila pošiljamo pametno, tako da noben procesor ne sprejme več kot enega sporočila. Središčni procesor razširi sporočilo v vse štiri smeri, njegovi zahodni in vzhodni sosedje ga pošljejo naprej v vse smeri, razen v tisto, iz katere je sporočilo prišlo. Vsi ostali procesorji sporočilo le posredujejo severano (ali južno). Procesorji, ki ne ležijo v območju kroga, sporočila ne posredujejo dalje.

Širjenje sporočila si lahko ogledamo na naslednji sliki (s križcem je označen središčni procesor):



```
program Krog;

type SporociloT = record
                    r: real;
                    x, y: integer;
                  end;

var Sprejeto: SporociloT;
    Smer, Stevec: integer;

  procedure Poslji(Sporocilo: SporociloT; vSmer: integer); external;
  procedure Sprejmi(var Sporocilo: SporociloT; var izSmeri: integer); external;
  procedure Prizgi; external;

  { Ta podprogram bo tudi prižgal našo točko, če res leži v krogu. }
  function VKrogu: boolean;
  begin
    if Sprejeto.x * Sprejeto.x + Sprejeto.y * Sprejeto.y > Sprejeto.r
      then VKrogu := false
      else begin Prizgi; VKrogu := true end;
  end; {VKrogu}

begin
  repeat
    Sprejmi(Sprejeto, Smer);
    case Smer of
      0: begin { središče kroga }
        Sprejeto.x := 0; Sprejeto.y := 0;
        Sprejeto.r := Sprejeto.r * Sprejeto.r;
        Prizgi;
        for Stevec := 1 to 4 do Poslji(Sprejeto, Stevec);
      end;
      2, 4: begin { vzhodno ali zahodno od središča }
        Sprejeto.x := Sprejeto.x + Smer - 3;
```

```
          if VKrogu then { obvestimo tri sosede }
            for Stevec := 1 to 4 do if Stevec <> Smer then
              Poslji(Sprejeto, Stevec);
        end;
        1, 3: begin { severno ali južno }
          Sprejeto.y := Sprejeto.y + Smer - 2;
          if VKrogu then Poslji(Sprejeto, 4 - Smer);
        end;
      end; {case}
    until false;
end. {Krog}
```