14. republiško tekmovanje v znanju računalništva (1990)

NALOGE ZA PRVO SKUPINO

1990.1.1

R: 6

Ko se računalnik vključi v komunikacijsko mrežo, ne pozna drugih računalnikov, ki so vključeni vanjo. Vsak računalnik v mreži ima svojo številko in ime. Računalnikov je največ MaxRac. **Napiši podprogram**, ki bo izpisal imena vseh računalnikov v mreži, vendar vsakega samo enkrat.

Na voljo imaš naslednja dva podprograma:

KdoSem — funkcija, ki vrne številko našega računalnika;

Vprašaj(Naslov, Ime, Sosedi, SosediL) — vpraša računalnik Naslov za njegovo ime in spisek njegovih sosedov. Ime računalnika dobimo v parametru Ime (8 znakov), število sosedov dobimo v SosediL, spisek številk sosedov pa v tabeli Sosedi. Največje mogoče število sosedov je MaxSosedi.

Opomba: številke računalnikov niso nujno zaporedna naravna števila; nekatera števila ne ustrezajo nobenemu računalniku. Podprogramu Vprasaj je dovoljeno podati le veljavno številko nekega (že znanega) računalnika. Za ime in spisek sosedov lahko seveda vprašaš tudi samega sebe.

1990.1.2

R: 8

V tabeli imamo podatke o imenih in letu rojstva množice oseb. Podatki so urejeni po imenih oseb v abecednem vrstnem redu. Vsa leta rojstva so med (vključno) 1880 in 1990. Podatke želimo urediti po letu rojstva, pri čemer želimo pri istem letu rojstva ohraniti urejenost po abecedi.

Opiši postopek, ki podatke iz podane tabele prepiše v drugo tabelo tako, da bodo urejeni po želenem kriteriju. Velikost dodatnih spremenljivk naj bo neodvisna od števila podatkov. Napiši rešitev, ki bo čim manjkrat pregledala posamezno osebo.

Namig: Ker je možnih let rojstva malo, lahko prešteješ, koliko oseb je rojenih v posameznem letu.

1990.1.3

R: 9

Podjetje „Zaphodove nore naprave“ načrtuje grafično kartico, ki naj bi se ponašala z zelo hitrim risanjem. Zato ima kartica na vsako točko rastrske slike (pixel) povezan svoj procesor. Vsak procesor izvaja svojo kopijo istega programa. Povezan je s svojimi štirimi sosedi. Vezi so oštevilčene od 1 do 4 v smeri urinega kazalca, začenši z zgornjo.

Barve so predstavljene s celimi števili. Neko ploskev na zaslonu, ki je omejena s točkami podane barve MejnaBarva (konstanta), želimo pobarvati s pravokotnim ponavljajočim se vzorcem velikosti XMax × YMax. Točka je znotraj ploskve, če ni mejne barve in je znotraj ploskve vsaj ena soseda.

Definiraj vsebino sporočila, ki si ga bodo procesorji pošiljali. Nato **napiši postopek**, ki bo tekel na vseh procesorjih in bo na zaslonu pobarval ploskev, omejeno z mejno barvo. Predpostaviš lahko, da bo sporočilo pravega formata na magičen način prišlo po zvezi v nek procesor znotraj ploskve.

Na voljo imaš naslednje podprograme:

JeSporocilo vrne številko vezi, kjer čaka sporočilo, ali 0, če ni nobenega sporočila;

Sprejmi(Zveza) vrne sporočilo iz zveze Zveza; dokler sporočila ni, čaka;

Poslji(Zveza, Sporocilo) pošlje sporočilo po zvezi Zveza;

MojaBarva vrne trenutno barvo točke, ki jo upravlja procesor;

PobarvajMe(Barva) pobarva točko, ki jo upravlja procesor, z barvo Barva;

Vzorec(x, y) vrne barvo, ki je na mestu (x, y) v vzorcu; za koordinate, ki so izven vzorca (če torej ne velja $0 \le x < XMax$ in $0 \le y < YMax$), vrednost ni definirana.

R: 10

1990.1.4

Kaj izpiše naslednji program? Namesto pisanja števil lahko napišeš tudi kratek program, ki da enake rezultate.

```
program Marvin(Output);
const
  Els = 128;
var
  a: array [1..Els] of integer;
  b: array [1..Els] of integer;
  c, d, e: integer;
begin {Marvin}
  c := 1; b[Els] := 0;
  for d := 1 to Els do begin a[d] := d; if d > 1 then b[d - 1] := d end;
  d := 0;
  while c <> 0 do begin e := b[c]; b[c] := d; d := c; c := e end;
  while d <> 0 do begin WriteLn(a[d]); e := b[d]; b[d] := c; c := d; d := e end;
end. {Marvin}
```

NALOGE ZA DRUGO SKUPINO

1990.2.1

R: 11

Kljub temu, da je Marvin hiperinteligentni robot, mu tokrat ne preostane drugega, kot da godrnjaje naredi preprosto inventuro skladišča transgalaktičnega podjetja A & F. Skladišče je zgrajeno kot n-dimenzionalni kvader (n je med 1 in MaxN). Podatki, ki jih Marvin dobi pri vhodu v skladišče, so število dimenzij skladišča (n) in dolžina skladišča Dolz v vsaki od dimenzij (Dolz je tabela velikosti n). **Napiši program**, ki bo Marvinu izpisal seznam koordinat osnovnih celic, ki jih mora pregledati (vsako celico natanko enkrat).

1990.2.2

R: 13

Naj bo (a) strogo naraščajoče zaporedje realnih števil $a_1 < a_2 < \ldots < a_n$. Tedaj element $a_{(n+1)\,\mathrm{div}\,2}$ imenujemo *srednji element zaporedja* (a). Tako je na primer v zaporedju -35, 2, 7, 14, 15 srednji element 7, v zaporedju 1, 3,14, 18, 19,5 pa je srednji element 3,14.

Naj bosta (a) in (b) dolgi, strogo naraščajoči zaporedji realnih števil, vsako s po n elementi, pri čemer je vsak element zaporedja (a) različen od vseh elementov zaporedja (b). **Opiši postopek**, ki dobi zaporedji (a) in (b) v tabelah in ki brez dodatne tabele kar se da hitro določi srednji element strogo naraščajočega zaporedja, ki ga skupaj tvorijo elementi zaporedij (a) in (b).

1990.2.3

R: 15

Sto procesorjev (vsak izvaja svoj proces) je povezanih med seboj v obroč tako, da ima vsak stik le s svojim levim in desnim sosedom. Vsak procesor izvaja svojo kopijo naslednjega programa:

```
program Gorilnik;

var
  Gori: boolean;      { gorilnik gori }
  Gorivo: integer;    { količina goriva v gorilniku }
  Temper: integer;    { temperatura gorilnika }

  function Netilec: boolean; external;
  function LeviGori: boolean; external;
  function DesniGori: boolean; external;
  procedure Cakaj; external;

begin {Gorilnik}
  Gori := false; Gorivo := 100; Temper := 0;
  if Netilec then { natanko en procesor je „netilec“ }
    begin Gori := true; Temper := 100 end;
  repeat
    while (Temper < 80) or (Gorivo < 50) do begin      { ne gori }
      if LeviGori or DesniGori then                    { segrevanje od sosedov }
```

```
        begin Temper := Temper + 11; if Temper > 100 then Temper := 100 end;
      if Temper > 0 then Temper := Temper − 1;        { ohlajanje }
      if Gorivo < 100 then Gorivo := Gorivo + 1;      { dotok goriva }
      Cakaj;
    end; {while}
    Gori := true; Temper := 100;                      { vžig }
    while Gorivo >= 10 do                             { gorenje }
      begin Gorivo := Gorivo − 10; Cakaj end;
    Gori := false; Gorivo := 0;                       { ugasnitev }
  until false;
end. {Gorilnik}
```

Podprogram Cakaj zadrži izvajanje programa za 0,1 sekunde, sicer pa lahko predpostavimo, da je izvajanje preostalega programa zelo hitro. Vsi procesorji pričnejo izvajati svoj program hkrati. Funkciji LeviGori in DesniGori vrneta ob klicu stanje spremenljivke Gori v levem oziroma desnem procesu. Funkcija Netilec vrne true le enemu procesorju, vsem ostalim vrne false.

Če opazujemo stanje spremenljivke Gori v vsakem procesu v obroču, lahko opazimo določeno podmnožico procesov, v katerih velja Gori = true.

Opiši, kako se podmnožica „gorečih“ procesorjev spreminja s časom (kako se „plamen“ seli). Ali se ta podmnožica sčasoma ustali ali je spreminjanje stalno? Odgovore **utemelji**.

R: 16

1990.2.4

Hkrati poženemo dva programa, ki tečeta vsak na svojem procesorju. Oba imata dostop do iste globalne spremenljivke, ki je bila pred tem nastavljena na vrednost nič. Med izvajanjem vsak program natanko desettisočkrat prebere vrednost spremenljivke, prišteje ena in novo vrednost zapiše nazaj, ne da bi vedel, kaj počne drugi program. Upoštevaj, da je hitrost izvajanja programov lahko močno neenakomerna.

Kolikšni sta najmanjša in največja vrednost, ki ju spremenljivka lahko zavzame, ko oba programa končata z delom? Ali lahko zavzame poljubno vrednost med najmanjšo in največjo? Kaj pa, če imamo n programov, kjer je n poljubno število med vključno 1 in 42? Odgovor **utemelji**.

NALOGE ZA TRETJO SKUPINO

R: 17

1990.3.1

V vrsto želimo postaviti n domin. Vsaka domina ima dve polji. Vsako polje je označeno z 1 ali 2 ali ... m pikami. **Napiši program**, ki za poljubno dano množico domin ugotovi, ali lahko vse domine zložimo v vrsto. Pri zlaganju v vrsto morata imeti soležni polji sosednjih domin enako število pik. Podatki za program so število domin in število pik na vsakem polju vsake od njih.

1990.3.2

R: 19

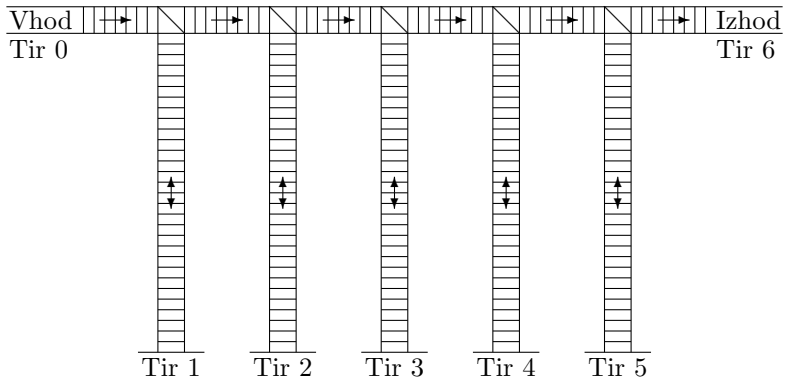
Železniška postaja je sestavljena iz 5 vzporednih slepih tirov. Na vhodu stoji 32 vagonov, oštevilčenih s številkami 1, 2, ..., 32, ki pa so med seboj premešani. Železničarji lahko premikajo po en vagon v smeri od vhoda proti izhodu. Vagon lahko torej premaknejo z vhoda postaje ali s poljubnega slepega tira na poljubni (kasnejši) slepi tir ali na izhod postaje. **Napiši program**, ki bo premikal vagone tako, da bodo na izhodni tir prihajali urejeni po naraščajočih številkah.

Na voljo imaš naslednje podprograme:

PremakniVagon(OdKod, Kam) premakne en vagon s tira OdKod na tir Kam. OdKod in Kam sta številki tirov, kjer je $0 \leq \mathsf{OdKod} < \mathsf{Kam} \leq 6$.

Prazen(Tir) vrne true, če je tir z oznako Tir prazen ($0 \leq \mathsf{Tir} \leq 6$).

Vagon(Tir) vrne številko vagona, ki je prvi na tiru Tir ($0 \leq \mathsf{Tir} \leq 6$); to je številka tistega vagona s tira Tir, ki ga je mogoče premakniti.



1990.3.3

R: 25

Med dvema računalnikoma, povezanima v komunikacijsko mrežo, potujejo datoteke. Vsaka datoteka je sestavljena iz več zapisov (vrstic), ki lahko potujejo kot paketi od pošiljatelja do prejemnika po različnih poteh, odvisno od obremenitve mreže. Paketi vsebujejo poleg zapisa z datoteke tudi identifikacijo datoteke, kateri pripadajo, zaporedno številko zapisa v datoteki in oznako, ali gre za zadnji zapis te datoteke. Istočasno lahko prihaja več datotek, katerih paketi se lahko med prenosom po različnih poteh različno zakasnijo in zato prihajajo v poljubnem vrstnem redu. Prav lahko se zgodi, da prispe zadnji zapis neke datoteke pred prvim, zato jih mora računalnik, ki jih sprejema, urediti po vrsti, preden jih dokončno shrani.

Napiši tisti del programa, ki prispele zapise izpiše v pravilnem vrstnem redu. V vsakem trenutku lahko hkrati prihaja največ 10 datotek, povprečna dolžina datoteke je 100 zapisov. Oblika prispelih paketov je takšna:

```
type PaketT = record
        VrstaPaketa: (Vmesni, Zadnji);   { vrsta prihajajočega paketa }
        Stevilka: integer;               { zaporedna številka paketa }
        Datoteka: 1..10;                 { številka datoteke, ki ji paket pripada }
        Zapis: ZapisT;                   { zapis v paketu }
      end; {PaketT}
```

Če je VrstaPaketa enaka Zadnji, potem nam njegova številka pove, koliko zapisov je v datoteki. Oblika tipa ZapisT ni pomembna. Na voljo imaš podprogram Pisi(Datoteka, Zapis), ki izpiše zapis iz enega paketa sporočila.

R: 27

1990.3.4

Marvin in Garvin sta nesrečna in malodušna robota. Raztovoriti morata vesoljsko ladjo, polno pangalaktičnega grloreza. Če se oba robota znajdeta hkrati v ladji, zavlada takó pesimistično vzdušje, da nadaljnje delo ni več mogoče, zato za medsebojno sinhronizacijo uporabljata spodaj podani algoritem. Na začetku sta oba robota izven vesoljske ladje. **Ali je možno, da se znajdeta oba robota hkrati v vesoljski ladji?** Odgovor utemelji!

Algoritem za robota Jaz (Jaz je konstanta z vrednostjo 1 ali 2):

```
const
  Jaz = ...;
var
  vLadji: 0..2;   { Ti dve spremenljivki si delita oba robota. Pomnilnik je organiziran }
  naVrsti: 1..2;  { tako, da lahko bere, piše ali testira vrednost ene spremenljivke    }
                  { le en robot naenkrat. Pred začetkom izvajanja programov je          }
                  { vrednost vLadji 0, naVrsti pa 1.                                    }

procedure Raztovarjanje;
begin
  repeat { zanka 0 }
    repeat { zanka 1 }
      repeat until (vLadji = 0) or (vLadji = Jaz); { zanka 2}
      naVrsti := Jaz;
      if vLadji = 0 then vLadji := Jaz;
    until (naVrsti = Jaz) and (vLadji = Jaz);
    { robot Jaz vstopi v ladjo, poišče zaboj grloreza, ga vzame in izstopi iz ladje }
    vLadji := 0;
  until LadjaPrazna;
end; {Raztovarjanje}
```

REŠITVE NALOG ZA PRVO SKUPINO

N: 1

R1990.1.1

Spodnji podprogram hrani v tabeli Spisek seznam vseh računalnikov, ki jih je doslej odkril. Sprehaja se po tem

spisku in vsak računalnik povpraša o sosedih; za dobljene sosede pogleda, če so že v spisku; če niso, jih doda. Tako bodo tudi na novo odkriti sosedje prej ali slej prišli na vrsto, da bomo tudi njih povprašali o njihovih sosedih in tako naprej.

```
const
  MaxRac = 1000;                    { največje število računalnikov v mreži }
  MaxSosedi = 20;                   { največje število povezav }
type ImeT = packed array [1..8] of char;
     SosediT = array [1..MaxSosedi] of integer;
var
  SpisekL: integer;                 { število računalnikov na spisku }
  Spisek: array [1..MaxRac] of record   { spisek računalnikov v mreži }
            Stevilka: integer;      { številka računalnika — naslov }
            Ime: ImeT;              { ime računalnika }
          end;

  function KdoSem: integer; external;
  procedure Vprasaj(Naslov: integer; var Ime: ImeT;
                    var Sosedi: SosediT; var SosediL: integer); external;

procedure NajdiVozlisca;
var Sosedi: SosediT;
    p, st, j, SosediL: integer;
begin
  { V spisek najprej vstavimo sebe. }
  SpisekL := 1; Spisek[SpisekL].Stevilka := KdoSem; St := 1;
  while st <= SpisekL do begin
    { Obdelamo vse sosede računalnika st. }
    Vprasaj(Spisek[st].Stevilka, Spisek[st].Ime, Sosedi, SosediL);
    for p := 1 to SosediL do begin
      { Za vsak računalnik iz Sosedi[st] pogledamo, ali je že na spisku. }
      j := 1; Spisek[SpisekL + 1].Stevilka := Sosedi[p];
      while Sosedi[p] <> Spisek[j].Stevilka do j := j + 1;
      if j > SpisekL then SpisekL := SpisekL + 1;
    end; {for}
    WriteLn(Spisek[st].Stevilka, ' ', Spisek[st].Ime);
    st := st + 1;
  end; {while}
end; {NajdiVozlisca}
```

Namesto seznama Spisek bi imeli lahko tudi razpršeno tabelo, tako da ne bi potrebovali notranje zanke **while**, pač pa bi le preverili, če je trenutni sosed že v razpršeni tabeli. To bi bilo hitreje, vendar bi pri našem problemu večino časa najbrž tako ali tako porabili za komuniciranje z drugimi računalniki v mreži.

N: 1

R1990.1.2

Ker je možnih let rojstva malo, bomo za vsako leto prešteli, koliko ljudi se je takrat rodilo. Spodnji podprogram hrani to število v so[Leto]. Zdaj vemo, da bodo v urejeni tabeli ljudje, rojeni leta 1880, pristali na indeksih od 1 do so[1880]; tisti, rojeni leta 1881, na indeksih od so[1880] + 1 do so[1880] + so[1881] in tako naprej. Tako lahko za vsako letnico izračunamo, na katerem indeksu se bodo v urejeni tabeli začeli podatki o ljudeh, rojenih tisto leto. Za te indekse lahko spet uporabimo tabelo so, ker podatkov o številu oseb, rojenih posamezno leto, kasneje ne bomo več potrebovali. Zdaj lahko opravimo drugi prehod po seznamu oseb; ko naletimo na človeka, rojenega leta Leto, ga vpišemo na indeks so[Leto] v izhodni tabeli b, vrednost so[Leto] pa povečamo za 1. Tako nam tabela so zdaj pravzaprav pove, kam vpisati naslednjo osebo, rojeno v posameznem letu. Pomembno pri tem načinu prerazporejanja oseb je, da se medsebojni vrstni red oseb, rojenih isto leto, ne bo spremenil — če je bil nekdo v tabeli a pred nekom drugim, rojenim isto leto, bo v tabeli b tudi.

Opisani postopek urejanja podatkov se imenuje „urejanje s štetjem (oz. preštevanjem)“ (*counting sort*). Uporabimo ga lahko v primerih, ko je možnih vrednosti ključa, po katerem urejamo, dovolj malo, da si lahko privoščimo za vsako možno vrednost ključa vzdrževati podatek o številu pojavitev te vrednosti v vhodnem zaporedju. (Pri naši nalogi je možnih le 111 ključev — letnice od 1880 do 1990.) Dobro je tudi, če je možnih vrednosti ključa razmeroma malo v primerjavi s številom elementov, ki bi jih radi uredili; če ni tako, bi znal biti kateri od splošnonamenskih postopkov urejanja vendarle učinkovitejši.

Lastnost, ki jo zahteva naša naloga, torej da postopek za urejanje ne spremeni medsebojnega vrstnega reda elementov z isto vrednostjo ključa (v našem primeru: ljudi, rojenih v istem letu), se imenuje *stabilnost*. Od znanih postopkov za urejanje so nekateri stabilni (npr. urejanje z mehurčki, z vstavljanjem, z izbiranjem in z zlivanjem), nekateri pa ne (npr. Shellovo urejanje, urejanje s kopico in običajna implementacija quicksorta).

```
const
  LetoMin = 1880;      { prvo možno leto rojstva }
  LetoMax = 1990;      { zadnje možno leto rojstva }
  MaxN = 13000;        { maksimalno število oseb }
type
  OsebaT = record
             Ime: packed array [1..32] of char; { podatki o osebi }
             Leto: LetoMin..LetoMax;
           end; {OsebaT}
  TabelaT = array [1..MaxN] of OsebaT;

procedure Uredi(var a, b: TabelaT; n: integer);
{ Dejansko število oseb, katerih podatke dobimo v tabeli a, je n.
  Urejene podatke o osebah vrnemo v tabeli b. }
```

```
var
  so: array [LetoMin..LetoMax] of integer; { št. oseb, rojenih v posameznem letu }
  i, j, k: integer;                        { števci }
begin
  { V tabeli so preštejemo, koliko oseb se je rodilo v posameznem letu. }
  for i := LetoMin to LetoMax do so[i] := 0; { začetne vrednosti }
  for i := 1 to n do so[a[i].Leto] := so[a[i].Leto] + 1; { štetje }
  { Podatke v tabeli preračunamo tako, da nam povedo, kje v pravilno
    urejeni tabeli b se začno osebe z danim letom rojstva. }
  j := so[LetoMin]; so[LetoMin] := 1;
  for i := LetoMin + 1 to LetoMax do
    begin k := so[i]; so[i] := so[i - 1] + j; j := k end;
  { Podatke o osebah prepišemo iz tabele a v tabelo b; upoštevamo, da nam
    so[r] pove, na katero mesto v tabeli b pride naslednja oseba z letom rojstva r. }
  for i := 1 to n do
    begin b[so[a[i].Leto]] := a[i]; so[a[i].Leto] := so[a[i].Leto] + 1 end;
end; {Uredi}
```

R1990.1.3

N: 1

Sporočila, ki si jih procesorji pošiljajo, naj bodo kar koordinate prejemnega procesorja. Ko torej nek procesor dobi sporočilo, lahko iz koordinat izračuna, kateri točki vzorca ustreza njegov položaj, tako da ve, s kakšno barvo se mora pobarvati. Nato še obvesti svoje sosede, razen seveda tistega, od katerega je sam dobil sporočilo. Poseben primer so procesorji, ki nadzirajo točke mejne barve — oni sporočil ne širijo, tako da se barvanje ustavi ob mejni barvi. Procesor lahko po tistem, ko se je že pobarval in obvestil sosede, prejme še več izvodov istega sporočila od raznih svojih sosedov, saj le-ti ne morejo vedeti, ali je bil že obveščen ali ne; taka odvečna sporočila lahko kar zavržemo.

```
procedure Pobarvaj;
var
  Zveza: integer;          { od kod smo dobili sporočilo }
  MojX, MojY: integer;     { kje v rastru smo }
  NovaBarva: integer;      { ustrezna barva iz vzorca }
begin
  if MojaBarva <> MejnaBarva then begin
      repeat Zveza := JeSporocilo until Zveza <> 0;
    repeat Zveza := JeSporocilo until Zveza <> 0; { čakamo na sporočilo }
    MojX := Sprejmi(Zveza);                       { sprejmemo svoje koordinate }
    MojY := Sprejmi(Zveza);
    NovaBarva := Vzorec(MojX mod XMax, MojY mod YMax);
    if NovaBarva <> MojaBarva then
      PobarvajMe(NovaBarva);                      { pobarvamo se }
    { Obvestimo sosede, razen tistega, ki je obvestil nas. }
    if Zveza <> 1 then begin Poslji(1, MojX); Poslji(1, MojY - 1) end;
    if Zveza <> 2 then begin Poslji(2, MojX + 1); Poslji(2, MojY) end;
```

```
    if Zveza <> 3 then begin Poslji(3, MojX); Poslji(3, MojY + 1) end;
    if Zveza <> 4 then begin Poslji(4, MojX − 1); Poslji(4, MojY) end;
  end; {if}
  repeat { počakamo, če nam bo še kdo ukazal, naj se pobarvamo }
    Zveza := JeSporocilo;
    if Zveza <> 0 then Zveza := Sprejmi(Zveza);  { zavržemo sporočilo }
  until false;
end; {Pobarvaj}
```

N: 2

R1990.1.4

Program izpiše tabelo a v obratnem vrstnem redu, torej vsa števila od 128 do 1.

Kratek program, ki da enak izpis, je tak:

```
program Marvin(Output);
const Els = 128;
var d: integer;
begin
  for d := Els downto 1 do WriteLn(d);
end. {Marvin}
```

Tabeli a in b uporablja kot seznam, v katerem je a[i] podatek in b[i] indeks naslednjega elementa seznama (kazalec na naslednji element). Del programa

```
  c := 1; b[Els] := 0;
  for d := 1 to Els do begin a[d] := d; if d > 1 then b[d − 1] := d end;
```

definira tabelo a in zgradi seznam. V danem primeru je naslednik i-tega elementa i + 1. Zadnji element ima indeks 0. Spremenljivka c vsebuje indeks prvega elementa seznama. Naslednja zanka v programu,

```
  d := 0;
  while c <> 0 do begin e := b[c]; b[c] := d; d := c; c := e end;
```

se pomika po tabeli indeksov do konca seznama. Pri tem obrne indekse v seznamu tako, da kažejo v nasprotno smer (v našem primeru i-ti element kaže na element i − 1). Ob začetku vsake ponovitve te zanke je c indeks trenutnega elementa, d pa indeks prejšnjega. (V e si začasno zapomnimo indeks naslednjega.) Po izhodu iz zanke spremenljivka d vsebuje indeks zadnjega elementa v seznamu. Zadnja zanka

```
  while d <> 0 do begin WriteLn(a[d]); e := b[d]; b[d] := c; c := d; d := e end;
```

izpisuje podatkovne elemente seznama, popravlja indekse na začetno stanje in se premika proti začetku seznama. Ob začetku vsake ponovitve te zanke kaže d na trenutni element, c na naslednjega, v e pa si začasno zapomnimo indeks prejšnjega. Po izhodu iz zanke je tabela indeksov b enaka kot po zgraditvi seznama. Spremenljivka c zopet kaže na prvi element seznama.

REŠITVE NALOG ZA DRUGO SKUPINO

R1990.2.1

N: 3

Če imamo n dimenzij in dolžine stranic od d_1 do d_n, lahko celice oštevilčimo od 0 do $m-1$ za $m = d_1 \cdot d_2 \cdot \ldots d_n$. Potem lahko preprosto naštejemo vsa ta števila in iz vsakega izračunamo koordinate celice, ki ji to število pripada. Dvorazsežni kvader bi lahko oštevilčili tako: celica s koordinatama (a_1, a_2) dobi indeks $i(a_1, a_2) = a_1 + d_1 a_2$. (Dogovorimo se, da bomo koordinate šteli od 0 do $d_i - 1$, ne od 1 do d_i.) Iz številke i lahko izračunamo koordinati po formuli $a_1 = i \bmod d_1$, $a_2 = i \operatorname{div} d_2$. V treh dimenzijah bi vzeli preslikavo $i(a_1, a_2, a_3) = a_1 + d_1 a_2 + d_1 d_2 a_3$ in obratno preslikavo $a_1 = i \bmod d_1$, $a_2 = (i \operatorname{div} d_1) \bmod d_2$, $a_3 = (i \operatorname{div} d_1) \operatorname{div} d_2$. Tako lahko nadaljujemo s poljubno mnogo dimenzijami:

$$i(a_1, \ldots, a_n) = \textstyle\sum_{j=1}^{n} a_j \prod_{k=1}^{j-1} d_j$$
$$\text{in obratno } a_j = (i \operatorname{div} \textstyle\prod_{k=1}^{j-1} d_j) \bmod d_j.$$

```
program MarvinovVodic(Input, Output);
const MaxN = 10;      { največja dimenzija kvadra }
type DimenzijeT = array [1..MaxN] of integer;
var
  n: integer;         { dimenzija kvadra }
  Dolz: DimenzijeT;   { dimenzije stranic }
  j: integer;         { števec po dimenzijah }

procedure Obhod(n: integer; var Dolz: DimenzijeT);
{ Sprehod skozi celice n-dimenzionalnega kvadra s stranicami Dolz. }
var i, j, k, m: integer;
begin
  m := 1; for j := 1 to n do m := m * Dolz[j];
  for i := 0 to m - 1 do begin
    k := i;
    for j := 1 to n do
      begin Write((k mod Dolz[j] + 1):3); k := k div Dolz[j] end;
    WriteLn;
  end; {for}
end; {Obhod}

begin {MarvinovVodic}
  Write('Dimenzija kvadra: '); ReadLn(n);
  WriteLn('Velikosti stranic');
  for j := 1 to n do begin Write(j, ': '); ReadLn(Dolz[j]) end;
  Obhod(n, Dolz);
end. {MarvinovVodic}
```

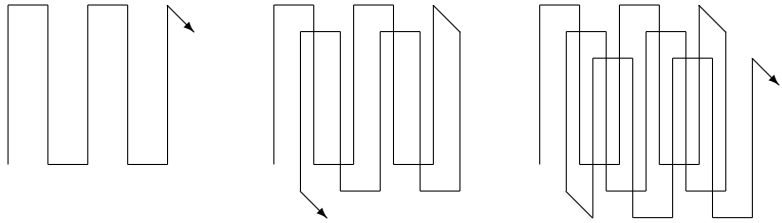
(*Opomba:* Write(j, ': ') lahko včasih pripelje do nerodnosti pri izpisu. Standard jezika pascal namreč ne predpisuje privzete širine polja pri izpisu

številskih vrednosti; torej je od prevajalnika odvisno, ali si bo Write(j) razlagal kot Write(j:1) ali pa mogoče kot Write(j:10) ali kaj podobnega. Da se izognemo morebitnim presenečenjem, bi bilo varneje uporabiti Write(j:1, ': ').)

Če bi hoteli robotu prihraniti pot (česar pa naloga ni zahtevala), bi lahko uporabili naslednji program. Z njim robot vedno stopi v *sosednjo* celico skladišča. Tu zapisani program je v tesni zvezi z nalogo o Grayevem kodiranju iz knjige „Enajsta šola računalništva“ (nalogi 1032B in 1133B), ki kaže na nekaj presenetljivih povezav med Hamiltonovo potjo skozi kvader (naš problem), Grayevim kodiranjem, hanojskimi stolpi in še čim.

```
program ZaLenobe(Input, Output);
const
  MaxN = 10;                          { največja dimenzija kvadra }
type
  DimenzijeT = array [1..MaxN] of integer;
var
  n: integer;                         { dimenzija kvadra }
  Dolz: DimenzijeT;                   { dimenzije stranic }
  Koord: DimenzijeT;                  { koordinate pregledovane kockice }
  Raste: array [1..MaxN] of boolean;  { števec v tej koordinati raste }
  Konec: boolean;                     { konec korakanja }
  Prenos: boolean;                    { stopiti moramo še po naslednji koordinati }
  i: integer;                         { števec }
begin
  Write('Dimenzija kvadra: '); ReadLn(n);
  WriteLn('Velikosti stranic');
  for i := 1 to n do begin Write(i:1, ': '); ReadLn(Dolz[i]) end;
  for i := 1 to n do begin Koord[i] := 1; Raste[i] := true end; { začetek }
  repeat
    for i := 1 to n do Write(Koord[i]:3);  WriteLn; { izpišemo, kje smo }
    i := 0; Konec := false;
    repeat
      Prenos := false; i := i + 1;      { stopimo po i-ti koordinati }
      if i > n then Konec := true       { prenos z zadnjega mesta = konec poti }
      else if Raste[i] then begin
        if Koord[i] < Dolz[i] then Koord[i] := Koord[i] + 1 { naprej }
        else begin Raste[i] := not Raste[i]; Prenos := true end;
      end
      else begin
        if Koord[i] > 1 then Koord[i] := Koord[i] - 1       { nazaj }
        else begin Raste[i] := not Raste[i]; Prenos := true end;
      end; {if}
      { če je Prenos = true, moramo stopiti še po koordinati i + 1 }
    until not Prenos;
  until Konec;
end. {ZaLenobe}
```

Pri tem načinu premikanja imamo za vsako dimenzijo predvideno neko trenutno smer gibanja (v smeri naraščajočih ali pa v smeri padajočih koordinat). Poskusimo se premakniti v smeri prve dimenzije, če pa to ne gre v želeno smer (ker bi padli ven iz kvadra), se bomo poskusili premakniti v naslednji dimenziji, smer prve dimenzije pa v mislih obrnemo. Tako bi na primer hodili najprej cik-cak v prvih dveh dimenzijah, ko pa bi te celice izčrpali, bi naredili en korak v tretji dimenziji in nato spet cik-cak v prvih dveh dimenzijah, le da v nasprotni smeri kot doslej.



Podobna tej nalogi je tudi 2002.1.1, kjer je treba preračunavati med koordinatami celice in njenim položajem na robotovi poti.

R1990.2.2

N: 3

Prva preprosta rešitev je naslednja. Če bi zlili obe zaporedji v eno samo dolgo urejeno zaporedje, bi bilo v njem $2n$ elementov in srednji element bi bil torej tisti na indeksu n. Zdaj lahko simuliramo zlivanje, dokler ne pridemo do n-tega elementa. Rešitev je še posebej preprosta, saj nam zaradi enakih dolžin zaporedij ni treba paziti, ali nam bo zmanjkalo kakega zaporedja. Spodnji program se pomika po prvem zaporedju s števcem ia, po drugem z ib, vsakič pa se premakne naprej po tistem zaporedju, ki ima na trenutnem položaju manjši element. Po $n - 1$ takšnih korakih kaže eden od teh dveh indeksov ravno na n-ti element zlitega zaporedja.

```
type TabelaT = array [1..100] of real;

function Srednji(n: integer; var a, b: TabelaT): real;
var i, ia, ib: integer;
begin
  ia := 1; ib := 1;
  for i := 1 to n − 1 do
    { Invarianta: manjši izmed elementov a[ia] in b[ib] je obenem tudi
      i-ti najmanjši v uniji vseh elementov zaporedij a in b. }
    if a[ia] < b[ib] then ia := ia + 1 else ib := ib + 1;
  if a[ia] < b[ib] then Srednji := a[ia] else Srednji := b[ib];
end; {Srednji}
```

Vendar pa je časovna zahtevnost tega postopka $O(n)$. Obstaja tudi bistveno hitrejša rešitev, ki temelji na hkratni bisekciji obeh zaporedij. Zaradi

enostavnejšega razmišljanja predpostavimo, da so v zaporedjih a in b sama različna števila (če to v praksi ne drži, se lahko pri primerjanju dveh elementov istega zaporedja, če se izkaže, da sta enaka, delamo, da je manjši tisti z manjšim indeksom; če sta iz različnih zaporedij, pa se delajmo, da je manjši tisti iz zaporedja a).

Iz definicije srednjega elementa sledi, da bo srednji element, ki ga iščemo (recimo mu x), n-ti najmanjši izmed vseh $2n$ elementov zaporedij a in b. Torej je $n-1$ elementov manjših od njega, n pa večjih od njega.

Recimo za začetek, da je n lih: $n = 2k-1$ za nek $k \geq 1$. Srednji element zaporedja a je torej a_k, srednji element zaporedja b pa je b_k. Primerjajmo a_k in b_k. Če se izkaže, da je $a_k < b_k$, to pomeni, da so od b_k manjši vsi elementi $a_1, \ldots, a_k$ in $b_1, \ldots, b_{k-1}$; to je skupaj $k+(k-1) = 2k-1 = n$ elementov. Od b_k je torej manjših vsaj n elementov (mogoče pa še kakšen več iz zaporedja a); mi pa smo v prejšnjem odstavku videli, da je od iskanega srednjega elementa x manjših natanko $n-1$ elementov. Torej je $b_k > x$, vsi nadaljnji elementi zaporedja b pa so seveda še večji in torej noben od njih ne more biti x. Zato lahko elemente $b_{k+1}, b_{k+2}, \ldots, b_n$ zavržemo.[1] Iz $a_k < b_k$ pa sledi tudi, da so od a_k večji vsi elementi $a_{k+1}, \ldots, a_n$ in $b_k, b_{k+1}, \ldots, b_n$, torej je od a_k večjih vsaj $(k-1)+k = n$ elementov. Od prej vemo, da je od x večjih natanko n elementov, torej mora biti $a_k = x$ (če je od a_k večjih natanko n elementov) ali pa $a_k < x$ (če je od a_k večjih več kot n elementov). V vsakem primeru to pomeni, da so prejšnji elementi zaporedja a, torej $a_1, \ldots, a_{k-1}$, vsi manjši od x in jih lahko tudi zavržemo.

Če bi se izkazalo, da je $a_k > b_k$, bi lahko opravili enak razmislek kot v gornjem odstavku, le namesto a bi si povsod mislili b in obratno. Učinek je v obeh primerih enak: iz enega zaporedja zavržemo $k-1$ elementov, ki so bili vsi manjši od x, iz drugega pa tudi prav toliko elementov, ki pa so bili vsi večji od x. Zato imata na ta način skrajšani zaporedji še vedno isti srednji element; obenem pa sta tudi še vedno obe enako dolgi. Torej imamo pred seboj problem enake oblike kot na začetku, le s krajšima zaporedjema, in njegova rešitev je prav isti x, ki je obenem tudi rešitev prvotnega problema.

Doslej smo govorili o možnosti, da je n lih. Recimo zdaj, da je n sod: $n = 2k$ za nek $k \geq 1$. Srednja elementa sta spet a_k in b_k. Če je $a_k < b_k$, je od b_k manjših vsaj $k+(k-1) = n-1$ elementov, torej je $b_k \geq x$ in lahko zavržemo člene $b_{k+1}, \ldots, b_n$; od a_k pa je večjih vsaj $k+(k+1) = n+1$ elementov, torej je $a_k < x$ in lahko zavržemo člene $a_1, \ldots, a_k$. — Če je $a_k > b_k$, spet razmišljamo podobno, le a in b imata zamenjani vlogi.

V vseh primerih smo dobili dve krajši zaporedji (namesto po n elementov imata po $(n+1) \operatorname{div} 2$ elementov) in zanju vemo, da je srednji element njune

[1] S to utemeljitvijo bi lahko zavrgli tudi b_k, a tega ne bomo storili. Tako bomo zagotovili, da bomo iz a zavrgli enako mnogo elementov kot iz b in bosta zaporedji tudi po tem ohranili enako dolžino. Zato bomo lahko v nadaljevanju iskali srednji element skrajšanih zaporedij z enakim postopkom kot tu na prvotnih zaporedjih.

unije isti kot srednji element x unije obeh prvotnih zaporedij. Torej lahko iskano število x poiščemo tako, da se z enakim postopkom lotimo skrajšanih zaporedij. Postopek se lahko ustavi, ko dobimo dve zaporedji dolžine 1; obe skupaj imata torej le dva elementa in takrat je „srednji element“ kar manjši izmed teh dveh elementov. V vsakem koraku imamo le konstantno mnogo dela, dolžina zaporedij pa se nam približno razpolovi, zato je časovna zahtevnost dobljenega postopka samo $O(\lg n)$.

V praksi imamo zaporedji predstavljeni z dvema tabelama in ko je treba zavreči nek del zaporedja, teh elementov ni treba zares brisati, saj vedno brišemo z začetka ali s konca zaporedij. Zato je dovolj, če si zapomnimo indeks prvega in zadnjega še nezbrisanega elementa, tadva indeksa pa potem upoštevamo tudi pri računanju srednjih elementov: če ima zaporedje namesto $a_1, \ldots, a_n$ člene $a_l, \ldots, a_r$, njegov srednji element ni $a_{(n+1)\,\mathrm{div}\,2}$, pač pa $a_{(l+r)\,\mathrm{div}\,2}$, njegova dolžina pa ni n, pač pa $r - l + 1$. To, ali je ta dolžina soda ali ne, si spodnji podprogram zapomni v spremenljivki Zamik.

```
type TabelaT = array [1..100] of real;

function Srednji(n: integer; var a, b: TabelaT): real;
var
  aLevi, bLevi, aDesni, bDesni, aSrednji, bSrednji: integer;
  Zamik: boolean;
begin
  aLevi := 1; aDesni := n; bLevi := 1; bDesni := n;
  while aLevi < aDesni do begin
    aSrednji := (aLevi + aDesni) div 2; bSrednji := (bLevi + bDesni) div 2;
    Zamik := Odd(aDesni - aLevi);
    if a[aSrednji] <= b[bSrednji] then begin
      aLevi := aSrednji; bDesni := bSrednji;
      if Zamik then aLevi := aLevi + 1;
    end else begin
      aDesni := aSrednji; bLevi := bSrednji;
      if Zamik then bLevi := bLevi + 1;
    end; {if}
  end; {while}
  if a[aLevi] < b[bLevi] then Srednji := a[aLevi] else Srednji := b[bLevi];
end; {Srednji}
```

R1990.2.3

N: 3

Hladen gorilnik se od gorečega soseda segreje do vžiga v 0,8 sekunde. Po vžigu gori največ eno sekundo, dokler ne porabi vsega goriva. Plamen se more prenesti le na sosednji, z gorivom dovolj napolnjeni gorilnik. Po ugasnitvi traja pet sekund, da se gorilnik dovolj napolni z gorivom in je ponovno pripravljen na vžig. V tem času se ohladi dovolj, da ne more priti do samovžiga. Ker gori plamen največ eno sekundo, se lahko širi le v eno smer (proti polnim gorilnikom), ne more pa se vrniti po

poti pravkar ugaslih gorilnikov. Plamen se od netilca širi na obe strani tako, da hkrati gorita eden do dva sosednja gorilnika na vsaki strani. Ker so gorilniki povezani v krog, se oba potujoča plamena srečata (v gorilniku diametralno nasproti netilca) po štiridesetih sekundah. Tam plamen ugasne zato, ker se pravkar izgoreli gorilniki v eni sekundi (dokler traja plamen) ne napolnijo z gorivom dovolj za ponoven vžig. Od tedaj se noben gorilnik ne prižge več.

N: 4

R1990.2.4

Po izteku obeh programov lahko spremenljivka zavzame poljubno vrednost med vključno 2 in 20000. Izberimo si poljuben n med 2 in 20000 in pokažimo enega od načinov, kako dobimo na koncu izvajanja programa vrednost n.

Označimo $p = n \operatorname{div} 2$ in $q = n - p$. Ker je $2 \le n \le 20000$, velja $1 \le p, q \le 10000$.

Prvi program prebere 0 in čaka. Drugi program $(10000 - p)$-krat prebere, poveča in zapiše vrednost spremenljivke. Zapisana vrednost je zdaj $10000 - p$, drugemu programu ostane še natanko p ponovitev.

Prvi program poveča svojo prebrano vrednost in zapiše 1. Drugi program prebere 1 in čaka. Prvi program $(10000 - q)$-krat prebere, poveča in zapiše vrednost spremenljivke. Zapisana vrednost je zdaj $10000 - q + 1$, prvemu programu ostane še natanko $q - 1$ ponovitev.

Drugi program poveča svojo prebrano vrednost in zapiše 2; zdaj mu ostane še natanko $p - 1$ ponovitev. Vse svoje preostale ponovitve izvede, preden prvi program nadaljuje z delom. Ob njegovem zaključku je vrednost spremenljivke enaka $p + 1$. Po zaključku dela drugega programa nadaljuje z delom prvi program: prebere vrednost $p + 1$ in jo v $q - 1$ ponovitvah poveča do $p + q = n$, ko konča s svojim delom.

Naj bo m število programov. Največja vrednost je enaka $10000 \cdot m$. Za $m = 1$ je najmanjša vrednost spremenljivke enaka največji, medtem ko je za $m \ge 2$ enaka 2. O tem se lahko prepričamo takole: če hočemo dobiti vrednost, manjšo ali enako 20000, lahko uporabimo enak postopek kot zgoraj, ostale programe pa pustimo od začetka do konca teči v času med tistim, ko prvi program prebere 0, in časom, ko ta program zapiše 1. Če pa hočemo vrednost, večjo od 20000, lahko uporabimo gornji postopek, da pridemo do 20000, na konec dodamo še toliko branj in pisanj iz ostalih programov (lepo sinhroniziranih, brez prepletanja), da bo končna vrednost enaka želeni, odvečna branja in pisanja ostalih programov pa spet lahko stlačimo v čas med tistim, ko prvi program prebere 0, in tistim, ko zapiše 1.

Vrednosti nad $10000 \cdot m$ očitno ne moremo dobiti, saj je vseh povečevanj skupaj le $10000 \cdot m$. Pri $m = 1$ tudi ne moremo dobiti drugačne vrednosti kot 10000, saj je možen potek računanja le ta, da edini program lepo po vrsti izvede vsa svoja branja in pisanja.

Prepričajmo se še, da ne moremo dobiti vrednosti, manjših od 2. Ker je spremenljivka na začetku enaka 0 in je nikoli ne zmanjšujemo, je njena

vrednost vedno nenegativna; zato pa, ko jo nek procesor prebere in poveča za 1 ter zapiše, bo gotovo zapisal pozitivno vrednost. Torej končna vrednost spremenljivke ne more biti 0 (ker bi bilo to mogoče le, če ne bi noben procesor nikoli ničesar zapisal). Končna vrednost bi bila lahko 1 le, če bi procesor, ki izvede zadnje pisanje, pri svojem zadnjem branju prebral ničlo; toda to bi se dalo le, če ne bi pred njim nihče pisal, v resnici pa je (če nihče drug) pred tistim svojim zadnjim branjem 9999-krat pisal že on sam.

REŠITVE NALOG ZA TRETJO SKUPINO

R1990.3.1

N: 4

Razpored domin bomo iskali z rekurzijo. V spodnjem programu počne to podprogram Postavi, ki poskuša na konec trenutnega razporeda dodati še eno domino. Preizkusiti mora vse možne domine, vsako v obeh možnih položajih; če se nova domina ujema s tisto, ki je bila dotlej zadnja, bomo poskusili z rekurzivnim klicem poskrbeti za razporeditev še preostalih domin. Koristno je voditi množico domin, ki smo jih že postavili v trenutni razpored (Zasedene), tako da pri razmišljanju o tem, kaj bi postavili na naslednje mesto, ne bomo izgubljali časa z njimi. Ko dodamo domino v razpored, jo dodamo tudi v to množico, ko pa se nato rekurzivni klic vrne in bomo namesto nje poskušali dodati kakšno drugo, jo moramo iz množice spet zbrisati.

```
program Domine(Output);
const
  n = 10;    { število domin, ki jih želimo postaviti v vrsto }
  m = 5;     { največje število pik na polju domine }
type
  DominaT = array [0..1] of 1..m;    { domina ima dve polji }
  PostavitevT = record               { domina v vrsti }
                  Dom: integer;      { številka domine }
                  Obrat: integer;    { liho: obrnjena; sodo: neobrnjena }
                end; {PostavitevT}
var
  Vrsta: array [0..n] of PostavitevT;    { vrsta, ki jo sestavljamo }
  Zaloga: array [1..n] of DominaT;       { domine, ki jih imamo na razpolago }
  Zasedene: set of 1..n;                 { domine, ki so že v vrsti }

procedure NapraviDomine;
{ Prebere, izračuna, napravi ali načara zalogo domin; na primer takole: }
var i: integer;

  function Random(Min, Max: integer): integer; external;

begin
  for i := 1 to n do begin
    Zaloga[i, 0] := Random(1, m); Zaloga[i, 1] := Random(1, m);
```

```
    WriteLn(i:3, ' (', Zaloga[i, 0]:1, ', ', Zaloga[i, 1]:1, ')');
  end; {for}
end; {NapraviDomine}

procedure IzpisiDomine;
var i: integer;
begin
  for i := 1 to n do with Vrsta[i] do
    WriteLn(Dom:3, ' (', Zaloga[Dom, Obrat mod 2]:1, ', ',
            Zaloga[Dom, (Obrat + 1) mod 2]:1, ')');
end; {IzpisiDomine}

function StPik(v: PostavitevT; Poz: integer): integer;
{ Vrne število pik domine na polju Poz, upoštevajoč rotacijo domine. }
begin
  StPik := Zaloga[v.Dom, (v.Obrat + Poz) mod 2];
end; {StPik}

function Primerjaj(v1, v2: PostavitevT): boolean;
{ Primerja dve domini. }
begin
  if Zasedene = [] then Primerjaj := true { prve domine ne primerjamo s prejšnjo }
  else Primerjaj := StPik(v1, 1) = StPik(v2, 0);
end; {Primerjaj}

function Postavi(Mesto: integer): boolean;
{ Postavi domine od Mesto naprej. }
var
  Nova: PostavitevT;          { domina, ki jo bomo postavili na Mesto }
  Uspeh, Neuspeh: boolean;    { uspeh/neuspeh postavitve vrste od Mesto naprej }
begin
  with Nova do begin Dom := 0; Obrat := 0 end; { začnemo s prvo domino }
  Uspeh := false; Neuspeh := false;
  repeat
    Nova.Dom := Nova.Dom + 1;
    if Nova.Dom > n then { ni šlo — poskusimo z obrnjenimi dominami }
      with Nova do begin Dom := 1; Obrat := Obrat + 1 end;
    if Nova.Obrat > 1 then
      Neuspeh := true { izčrpali smo vse možnosti — ne gre }
    else if not (Nova.Dom in Zasedene)
          and Primerjaj(Vrsta[Mesto - 1], Nova) then begin
      Vrsta[Mesto] := Nova; Zasedene := Zasedene + [Nova.Dom];
      if Mesto = n then Uspeh := true else Uspeh := Postavi(Mesto + 1);
      if not Uspeh then Zasedene := Zasedene - [Nova.Dom];
    end; {if}
  until Uspeh or Neuspeh;
  Postavi := Uspeh;
```

```
end; {Postavi}

begin {Domine}
  NapraviDomine; Zasedene := [];
  if not Postavi(1) then WriteLn('Domin se ne da postaviti v vrsto.')
  else begin WriteLn('Domine se da postaviti v vrsto:'); IzpisiDomine end;
end. {Domine}
```

R1990.3.2

N: 5

Lahko si pomagamo z zlivanjem. To je postopek, s katerim dobimo iz dveh ali več urejenih zaporedij novo, daljše urejeno zaporedje, v katerem so vsi elementi vhodnih zaporedij. Postopek je preprost: pogledamo prvi element vsakega vhodnega zaporedja in najmanjšega med njimi premaknemo iz tega zaporedja na konec izhodnega zaporedja. Ta korak zdaj ponavljamo in tako v izhodno zaporedje vsakič dodamo naslednji element po velikosti; ko se vsa vhodna zaporedja izpraznijo, je postopek končan.

Pri nas bodo zaporedja tiri, elementi zaporedij pa vagoni. Postopek se nam bo rahlo zapletel, ker vagon, ki ga kot zadnjega dodamo na nek tir, kasneje prvi pride z njega; ko torej pri zlivanju dodajamo na tir vagone po naraščajočih številkah, bo kasneje, ko bomo ta tir uporabili kot enega od vhodov pri nekem kasnejšem zlivanju, videti, kot da so vagoni na njem urejeni v nasprotnem vrstnem redu, torej po padajočih številkah, saj se bo s tega tira prvi pripeljal vagon z največjo številko in tako naprej.

Kakorkoli že, na začetku imamo le eno neurejeno zaporedje 32 vagonov (na tiru 0). Preden bomo lahko kaj zlivali, ga moramo razbiti na več krajših zaporedij. Pri tem moramo paziti na pravilo, da se vagon ne more nikoli premakniti na tir z manjšo številko od tistega, na katerem se nahaja trenutno; pri zlivanju nam torej vagoni prihajajo na tire z vse višjimi številkami in paziti moramo, da nam ne bo tirov zmanjkalo, še preden bodo opravljena vsa zlivanja.

Za začetek premaknimo en vagon s tira 0 na tir 1. Zdaj si lahko mislimo, da imata tako tir 0 kot tir 1 urejeno zaporedje dolžine 1 (na tiru 0 je sicer za tem enim vagonom še trideset drugih v nekem neznanem vrstnem redu, ampak zanje se zdajle pri zlivanju ne bomo zmenili). Ti dve zaporedji zdaj zlijmo in na tiru 2 dobimo (narobe obrnjeno) urejeno zaporedje dveh vagonov.

Premaknimo spet en vagon s tira 0 na tir 1. Lahko se delamo, da imamo na tirih 0 in 1 narobe obrnjeni urejeni zaporedji dolžine 1; na tiru 2 pa je še od prej narobe obrnjeno urejeno zaporedje dolžine 2. Vse troje zlijmo in na tiru 3 dobimo (prav obrnjeno) urejeno zaporedje dolžine 4.

Zdaj lahko s podobnimi operacijami kot prej pridelamo na tiru 2 še eno (prav obrnjeno) urejeno zaporedje dolžine 2, na tirih 0 in 1 pa po en vagon; ko vse to (vključno s štirimi vagoni na tiru 3) zlijemo, dobimo na tiru 4 narobe obrnjeno urejeno zaporedje dolžine 8.

Potem spet s podobnimi operacijami pripravimo na tirih 1, 2 in 3 narobe

obrnjena urejena zaporedja dolžine 1, 2 in 4; nato zlivamo s tirov 0–4 na tir 5 in dobimo tam prav obrnjeno urejeno zaporedje dolžine 16.

Če vse skupaj ponovimo še enkrat, da dobimo na tirih 1–4 spet urejena zaporedja dolžine 1, 2, 4 in 8, lahko zdaj zlijemo vse to skupaj s tirom 5 in še zadnjim preostalim vagonom s tira 0 ter rezultat (urejeno zaporedje dolžine 32) odpošiljamo naravnost na izhodni tir 6.

Dobro je paziti še na naslednje: pri našem postopku vedno, ko prvič pošiljamo vagone na nek tir, nastane na njem zaporedje, ki je obrnjeno ravno narobe kot ob prvem zlivanju na prejšnji tir. Ker smo začeli s tem, da smo si na tiru 1 mislili prav obrnjeno zaporedje dolžine 1, smo na tiru 2 dobili narobe obrnjeno, na tiru 3 spet prav obrnjeno in tako naprej, na koncu pa na tiru 5 tudi prav obrnjeno. Zato nam pri tistem zadnjem zlivanju vagoni odhajajo na izhodni tir po naraščajočih številkah, tako kot smo želeli. Če pa bi imeli na primer 64 vagonov in šest pomožnih tirov, bi nam pri opisanem postopku na koncu nastala narobe obrnjena zaporedja in tudi vagoni bi na izhodni tir prihajali po padajočih številkah. Očitno je, da bi morali v tem primeru že od vsega začetka ravno obrniti vrstni red na vseh tirih (torej začeti s predpostavko, da je osamljeni vagon na tiru 1 narobe obrnjeno urejeno zaporedje dolžine 1). To je odvisno od tega, ali je število tirov sodo ali liho. Spodnji program prenaša podatke o zahtevani urejenosti (naraščajoči ali padajoči) kar s parametrom ob rekurzivnih klicih. Glavno je to, da imamo pred zadnjim zlivanjem (tistim, ki bo pošiljalo vagone na izhodni tir), na vseh tirih naraščajoča zaporedja.

```
program ZelezniskaPostaja;
const
  n = 5;                { število slepih tirov }
  VhodniTir = 0;        { indeks vhodnega tira }
  IzhodniTir = n + 1; { indeks izhodnega tira }

  procedure PremakniVagon(OdKod, Kam: integer); external;
  function Prazen(Tir: integer): boolean; external;
  function Vagon(Tir: integer): integer; external;

{ Zlije vsebino tirov 1..StTirov na tir StTirov + 1. V zlivanje vključi tudi prvi vagon
  s tira 0. Če je Obrnjeno = true, predpostavi, da so vagoni na vhodnih tirih urejeni
  padajoče. Na tiru StTirov + 1 bo urejenost v vsakem primeru ravno nasprotna. }
procedure Zlivanje(StTirov: integer; Obrnjeno: boolean);
var Ze0: boolean; Tir, Min, KjeMin: integer;
begin
  Ze0 := false; { Vagona s tira 0 še nismo premaknili. }
  repeat
    { Poiščimo med prvimi vagoni s tirov 1..StTirov najmanjšega
      (ali največjega, če je Obrnjeno = true). Če je Ze0 = false,
      gledamo tudi tir 0. }
    KjeMin := -1; if Ze0 then Tir := 1 else Tir := 0;
```

```
      while Tir <= StTirov do begin
        if not Prazen(Tir) then
          if (KjeMin < 0) or ((Vagon(Tir) < Min) <> Obrnjeno) then
            begin Min := Vagon(Tir); KjeMin := Tir end;
        Tir := Tir + 1;
      end; {while}
      { Premaknimo najdeni vagon na tir StTirov + 1. }
      if KjeMin >= 0 then PremakniVagon(KjeMin, StTirov + 1);
      if KjeMin = 0 then Ze0 := true; { s tira 0 ne smemo več brati }
    until KjeMin < 0;
end; {Zlivanje}

{ Predpostavi, da so tiri 1..StTirov trenutno prazni. Poskrbi, da se za vse i od 1
  do StTirov na tiru i nahaja 2^i vagonov, urejenih naraščajoče, če je Obrnjeno = true,
  in padajoče, če je Obrnjeno = false. Vse te vagone vzame z vhodnega tira 0. }
procedure NapolniTire(StTirov: integer; Obrnjeno: boolean);
begin
  if StTirov = 1 then PremakniVagon(0, 1)
  else begin
    { Napolnimo prvih StTirov - 1 tirov v nasprotnem vrstnem redu. }
    NapolniTire(StTirov - 1, not Obrnjeno);
    { Z zlivanjem dobimo na tiru StTirov ravno prav vagonov
      v ravno pravem vrstnem redu. }
    Zlivanje(StTirov - 1, not Obrnjeno);
    { Prvih StTirov - 1 je spet praznih, napolnimo jih zdaj v
      želenem vrstnem redu. }
    NapolniTire(StTirov - 1, Obrnjeno);
  end;
end; {NapolniTire}

begin {ZelezniskaPostaja}
  { Pripravimo na i-tem tiru 2^i vagonov v pravem vrstnem redu,
    za vse i od 1 do n. En vagon ostane še na vhodnem tiru. }
  NapolniTire(n, false);
  { Zdaj jih zlijemo na izhodni tir. Kot si želimo, bodo
    prišli vagoni z nižjimi številkami prej na izhodni tir. }
  Zlivanje(n, false);
end. {ZelezniskaPostaja}
```

Malo drugačen, čeprav po svoje zelo podoben, pa je tudi naslednji razmislek. Vagone v mislih razdelimo v dve skupini: 1–16 in 17–32. Iz naloge je videti, da bi moralo biti pet pomožnih tirov dovolj za urejanje $32 = 2^5$ vagonov, torej si lahko mislimo, da bi utegnili biti za 16 vagonov dovolj že štirje tiri. Uredimo torej najprej vagone od 1 do 16 z uporabo pomožnih tirov 2–5, pomožni tir 1 pa uporabimo zato, da nanj odlagamo vagone s številkami 17–32, ko jih dobivamo z vhodnega tira. Ko je to opravljeno in smo vagone 1–16 srečno in v pravem

vrstnem redu odposlali na izhodni tir, lahko zdaj uredimo še preostalih 16 vagonov, torej tiste s številkami 17–32, ki nas čakajo na tiru 1, kamor smo jih prej odložili.

Za urejanje 16 vagonov bi seveda uporabili enak razmislek: razdelimo jih v dve skupini po osem, tiste iz druge skupine odlagamo na prvi prosti pomožni tir, ostale pa urejamo sproti. To rekurzivno razmišljanje se konča pri enem vagonu, ki ga „uredimo“ brez kakršnih koli pomožnih tirov preprosto tako, da ga pošljemo z vhodnega tira na izhodnega. Program nam malce zaplete le dejstvo, da je hkrati v teku več urejanj: tiste vagone, ki jih ne odlagamo na pomožni tir, takoj pošiljamo v obdelavo naslednjemu urejanju, ki jih bo mogoče odložilo na svoj pomožni tir, mogoče pa spet poslalo naprej in podobno.

```
procedure PrenosNaprej(STira, PomozniTir, StOd, StDo, StVagona: integer);
{ Ta podprogram opravi en korak urejanja za vagone s številkami StOd..StDo.
  Trenutno je treba nekaj narediti z vagonom StVagona, ki prihaja s tira STira. Če je
  ta vagon s prve polovice intervala StOd..StDo, ga predamo podprogramu za urejanje
  te polovice (z rekurzivnim klicem), sicer pa ga odložimo na stranski tir. }
var Meja: integer;
begin
  Meja := (StOd + StDo) div 2;
  { Vagone (Meja + 1)..StDo bomo odložili na pomožni tir,
    vagone StOd..Meja pa bomo prenesli naprej. Pri StOd = StDo
    je možen itak en sam vagon in pomožni tir je tedaj isti kot izhodni. }
  if (StVagona > Meja) or (StOd = StDo) then PremakniVagon(STira, PomozniTir)
  else PrenosNaprej(STira, PomozniTir + 1, StOd, Meja, StVagona);
end; {PrenosNaprej}

procedure SprazniTir(StTira, StOd, StDo: integer);
{ Predpostavi, da se na tiru StTira nahajajo vagoni StOd..StDo. Poskrbi,
  da se ta tir sprazni in vsi vagoni pridejo na izhodni tir v pravem vrstnem redu. }
var Meja: integer;
begin
  Meja := (StOd + StDo) div 2;
  if StTira < n then if not Prazen(StTira + 1) then
    { Naslednji tir bomo potrebovali kot pomožni tir, a so na njem
      še stari vagoni, torej najprej spraznimo tega. }
    SprazniTir(StTira + 1, StOd - (StDo - Meja), StOd - 1);
  { Spraznimo zdaj zahtevani tir. }
  while not Prazen(StTira) do begin
    PrenosNaprej(StTira, StTira + 1, StOd, StDo, Vagon(StTira));
  end; {while}
  { Z rekurzivnim klicem spraznimo še naslednje pomožne tire. }
  if StTira < n then SprazniTir(StTira + 1, Meja + 1, StDo);
end; {SprazniTir}

begin {ZelezniskaPostaja}
```

```
  SprazniTir(VhodniTir, 1, 1 shl n);
end. {ZelezniskaPostaja}
```

Isti postopek lahko zapišemo tudi bolj eksplicitno. Vagone si mislimo oštevilčene od 0 do 31 namesto od 1 do 32. Če skušamo sprazniti tir $n-b$, si mislimo, da so tiri $n-b+1, \ldots, n$ že prazni in predpostavimo, da se številke vagonov na tiru $n-b$ razlikujejo le v spodnjih b bitih, v preostalih $n-b$ bitih pa imajo vsi ti vagoni enake vrednosti. Zdaj pri vsakem vagonu poglejmo, kateri je v njegovi številki najvišji prižgani bit izmed spodnjih b bitov; če je to bit $b-1$, gre ta vagon na tir $n-b+1$, če je to bit $b-2$, gre na $n-b+2$ in tako naprej. Če ima nek vagon na spodnjih b bitih same ničle, gre naravnost na izhodni tir. Na koncu z rekurzivnimi klici obdelajmo vagone, ki so se nam nabrali na tirih $n-b+1, \ldots, n$.

```
const n = 5; { Število slepih tirov; dvojiški logaritem števila vagonov. }

  procedure PremakniVagon(OdKod, Kam: integer); external;
  function Prazen(Tir: integer): boolean; external;
  function Vagon(Tir: integer): integer; external;

{ Vrne indeks najvišjega prižganega bita v x med biti 0..(m - 1).
  Če so same ničle, vrne -1. }
function NajvisjiBit(x, m: integer): integer;
var j: integer;
begin
  j := m - 1;
  while j >= 0 do
    if (x and (1 shl j)) <> 0 then break
    else j := j - 1;
  NajvisjiBit := j;
end; {NajvisjiBit}

procedure Uredi(Tir: integer);
var i, StBitov: integer;
begin
  { Predpostavka: na tiru Tir so vagoni, katerih številke (če bi jih šteli od
    0 naprej namesto od 1 naprej) bi se razlikovale le v spodnjih StBitov bitih,
    tiri od Tir + 1 do n pa so prazni. }
  StBitov := n - Tir;
  { Razporedimo vagone med tire Tir + 1 do n + 1 glede na najvišji prižgani bit
    med spodnjimi StBitov biti. To nam zagotovi, da pridejo vsi s tira i + 1
    v vrstnem redu pred vsemi s tira i. }
  while not Prazen(Tir) do
    PremakniVagon(Tir, n - NajvisjiBit(Vagon(Tir) - 1, StBitov));
  { Počistimo tire od n do Tir + 1. }
  for i := n downto Tir + 1 do Uredi(i);
  { Rezultat: prazni so tiri od Tir do n, vagone, ki so bili prej na tiru Tir,
```

```
    pa smo zdaj v pravem vrstnem redu premaknili na izhodni tir. }
end; {Uredi}

begin {ZelezniskaPostaja}
  Uredi(0);
end. {ZelezniskaPostaja}
```

Še opomba glede funkcije NajvisjiBit. V njej smo uporabili operator **and** nad celimi števili, čeprav je v standardnem pascalu definiran le nad logičnimi vrednostmi; tudi operatorja **shl** in ukaza **break** standardni pascal nima. Če bi se torej hoteli bolj držati standarda, bi lahko naredili nekaj takega:

```
function NajvisjiBit(x, m: integer): integer;
var j: integer;
begin
  NajvisjiBit := -1;
  for j := 0 to m - 1 do begin
    if Odd(x) then NajvisjiBit := j;
    x := x div 2;
  end; {for}
end; {NajvisjiBit}
```

Majhna slabost te različice je, da gre vedno po vseh m bitih, medtem ko gre prejšnja različica od zgornjih bitov proti spodnjim in ustavi že pri prvem prižganem bitu (in od števil $0, \ldots, 2^m - 1$ je kar polovica takih, pri katerih je prižgan že kar najvišji bit).[2]

Razmislimo še o tem, kolikokrat naši algoritmi premikajo vagone. Naj bo $f(n)$ število klicev podprograma PremakniVagon, če delamo z 2^n vagoni in n pomožnimi tiri. Prvi algoritem ima $f(0) = 1$ (če imamo le en vagon, ga samo premaknemo z vhodnega tira na izhodnega) in $f(n) = 2f(n-1) + 2^{n-1}$ ($2f(n-1)$ zaradi dveh rekurzivnih klicev v podprogramu NapolniTire, 2^{n-1} pa zaradi zlivanja). Drugi in tretji algoritem imata tudi $f(0) = 1$, pri večjih n pa upoštevamo, da se pol vagonov (torej 2^{n-1} vagonov) odloži na prvi pomožni tir, z ostalimi pa ravnamo takoj tako, kot da bi imeli le $n-1$ tirov in 2^{n-1} vagonov; na koncu še tiste, ki smo jih odložili na prvi pomožni tir, uredimo po enakem postopku, torej spet kot da bi imeli le $n-1$ pomožnih tirov in 2^{n-1} vagonov. Tako smo spet dobili zvezo $f(n) = 2f(n-1) + 2^{n-1}$. Vsi trije algoritmi torej izvedejo enako število premikov. Če rekurzivno zvezo za $f(n)$ vstavljamo samo vase, lahko dobimo tudi eksplicitno obliko: $f(n) = 2f(n-1) + 2^{n-1} = 2(2f(n-2) + 2^{n-2}) + 2^{n-1} = 2^2 f(n-2) + 2 \cdot 2^{n-1} = 2^3 f(n-3) + 3 \cdot 2^{n-1} = \ldots = 2^n f(0) + n \cdot 2^{n-1} = (n+2)2^{n-1}$. Pri naši nalogi je $n = 5$ in potrebujemo 112 premikov.[3]

[2] Iskanja najvišjega prižganega bita bi se lahko lotili še na razne druge načine; glej rešitev naloge 2000.1.2.

[3] Gl. tudi *The On-Line Encyclopedia of Integer Sequences*, A001792.

Zanimivo vprašanje pri našem problemu urejanja vagonov s pomočjo pomožnih slepih tirov je tudi to, koliko pomožnih tirov potrebujemo, da lahko uredimo vsako zaporedje N vagonov. Izkaže se, da za šest ali manj vagonov zadoščata že dva pomožna tira (za sedem vagonov pa včasih potrebujemo že tri pomožne tire); potem pa, če znamo z n pomožnimi tiri urediti N vagonov, lahko z $n+1$ pomožnimi tiri uredimo $2N$ vagonov (z enakim rekurzivnim razmislekom, kakršnega smo videli že zgoraj v naši rešitvi za pet tirov in 32 vagonov). Z n pomožnimi tiri lahko torej vsekakor uredimo vsako zaporedje $3 \cdot 2^{n-1}$ ali manj vagonov; če to obrnemo, vidimo, da za urejanje poljubnega zaporedja N vagonov gotovo zadostuje $\lceil \log_2(2N/3) \rceil$ pomožnih tirov.[4]

R1990.3.3

Podprogram ShraniPaket spremlja število prispelih paketov vsake datoteke in podatke o njih hrani v tabeli Paketi. Ko prispejo vsi zapisi neke datoteke, jih z zaporednimi klici podprograma Pisi po vrsti izpiše in sprosti pomnilnik, ki so ga zasedali, da je na voljo novim datotekam. N: 5

```
const
  MaxPaketov = 1500; { več kot število datotek × povprečna dolžina }
  MaxDatotek = 10;
type
  ShranjenZapisT = record                    { oblika podatkov o prispelih zapisih }
                     Stevilka: integer;      { številka zapisa v datoteki }
                     Naslednji: integer;     { naslednji element v verigi }
                     Zapis: ZapisT;          { vsebina zapisa }
                   end; {ShranjenZapisT}
  OpisDatotekeT = record                     { oblika opisa prihajajoče datoteke }
                    ZadnjiZapis: integer;    { številka zadnjega zapisa }
                    Sprejetih: integer;      { število sprejetih zapisov }
                    PrviZapis: integer;      { prvi element v verigi zapisov }
                  end; {OpisDatotekeT}
var PrazniZapisi: integer; { prvi element v verigi prostih zapisov v tabeli Zapisi }
    Opisi: array [1..MaxDatotek] of OpisDatotekeT;      { opisi datotek }
    Zapisi: array [1..MaxPaketov] of ShranjenZapisT;    { seznam zapisov }

  procedure Pisi(Datoteka: integer; Zapis: ZapisT); external;
```

[4] To je torej zgornja meja za minimalno potrebno število pomožnih tirov, s katerimi se da urediti vsa zaporedja N vagonov; znana pa je tudi spodnja meja, namreč $\frac{1}{2}\log_2 N$ — z manj kot toliko tiri se gotovo ne bo dalo urediti vseh zaporedij N vagonov. Naš problem urejanja vagonov lahko tudi posplošimo, če skladov (= pomožnih slepih tirov) ne zvežemo v zaporedje, ampak v poljuben drug graf, in če namesto skladov v vozliščih grafa dovolimo tudi vrste. Literatura: Knuth, *The Art of Computer Programming*, 3. knjiga, nalogi 5.2.4.19–20; R. Tarjan, *Sorting using networks of queues and stacks*, Journal of the ACM, 19(2):341–346, April 1972; T. Jiang, M. Li, P. Vitányi, *Average-case analysis of algorithms using Kolmogorov complexity*, J. of Comp. Science and Technology, 15(5):402–408, September 2000.

```
procedure PripraviPakete;
{ Pripravi tabeli Opisi in Zapisi. }
var i: integer;
begin
  { Cela tabela Zapisi je ena sama dolga veriga praznih zapisov. }
  PrazniZapisi := 1;
  for i := 1 to MaxPaketov − 1 do Zapisi[i].Naslednji := i + 1;
  Zapisi[MaxPaketov].Naslednji := −1;
  { Na začetku še ne sprejemamo nobene datoteke. }
  for i := 1 to MaxDatotek do with Opisi[i] do
    begin ZadnjiZapis := 0; Sprejetih := 0; PrviZapis := −1 end;
end; {PripraviPakete}

procedure SprostiDatoteko(Dat: integer);
{ Sprosti prostor v tabeli Zapisi, ki ga je zasedala datoteka Dat.
  Njene zapise postavimo na začetek verige praznih zapisov. }
var i, j: integer;
begin
  j := PrazniZapisi;
  with Opisi[Dat] do begin
    i := PrviZapis; PrviZapis := −1;
    ZadnjiZapis := 0; Sprejetih := 0; PrazniZapisi := i;
  end; {with}
  while Zapisi[i].Naslednji <> −1 do i := Zapisi[i].Naslednji;
  Zapisi[i].Naslednji := j;
end; {SprostiDatoteko}

procedure ShraniPaket(p: PaketT);
{ Shrani prispeli paket p v tabelo Zapisi in izpiše datoteko, če so prišli že vsi zapisi. }
var i, j: integer;
begin
  if p.VrstaPaketa = Zadnji then
    Opisi[p.Datoteka].ZadnjiZapis := p.Stevilka;
  Opisi[p.Datoteka].Sprejetih := Opisi[p.Datoteka].Sprejetih + 1;
  i := PrazniZapisi; PrazniZapisi := Zapisi[i].Naslednji;
  with Zapisi[i] do begin
    Naslednji := Opisi[p.Datoteka].PrviZapis;
    Stevilka := p.Stevilka; Zapis := p.Zapis;
  end; {with}
  Opisi[p.Datoteka].PrviZapis := i;
  with Opisi[p.Datoteka] do
    if Sprejetih = ZadnjiZapis then begin
      for i := 1 to ZadnjiZapis do begin
        j := PrviZapis;
        while Zapisi[j].Stevilka <> i do j := Zapisi[j].Naslednji;
        Pisi(p.Datoteka, Zapisi[j].Zapis);
      end; {for}
```

```
        SprostiDatoteko(p.Datoteka);
    end; {if}
end; {ShraniPaket}
```

Slabost tega programa je, da bi odpovedal pri sprejemanju kakšne zelo dolge datoteke (več kot MaxPaketov paketov). Lahko bi ga izboljšali, da bi pomnilnik za shranjene pakete zasegal in sproščal dinamično. Če hranimo pakete v seznamih, povezanih s kazalci, bi jih lahko uredili s kakšno različico zlivanja (*merge sort*). Lahko bi tudi imeli za vsako datoteko kazalce na prvih nekaj sto paketov kar v tabeli (vsak element tabele ustreza eni od prvih toliko zaporednih številk paketa, tako da teh paketov na koncu sploh ne bo treba urejati), preostale pakete (če je datoteka tako dolga, da je to potrebno) pa bi hranili v seznamu: s tem bi se pri večini datotek izognili potrebi po urejanju paketov (na primer: če imamo v tabeli prostora za tristo paketov, povprečna datoteka pa ima sto paketov, je gotovo kvečjemu tretjina datotek daljših od tristo paketov).

R1990.3.4

Algoritem ne zagotavlja medsebojnega izključevanja — oba robota se lahko hkrati znajdeta v vesoljski ladji. Oglejmo si primer, kako lahko pride do tega: N: 6

vLadji ima vrednost 0
robot 1: preskoči zanko 2 in nastavi naVrsti := 1
preveri pogoj vLadji = 0 in vstopi v telo stavka **if**
robot 2: preskoči zanko 2 in nastavi naVrsti := 2
preveri pogoj vLadji = 0 in vstopi v telo stavka **if**
nastavi vLadji := 2
preveri (vLadji = 2) **and** (naVrsti = 2) in vstopi v ladjo
robot 1: nastavi vLadji := 1 in se vrne na začetek zanke 1
preskoči zanko 2 in nastavi naVrsti := 1
preveri (vLadji = 1) **and** (naVrsti = 1) in vstopi v ladjo

Oglejmo si še primer malo drugačnega sinhronizacijskega algoritma, ki (za razliko od tistega iz besedila naloge) uspešno preprečuje, da bi se lahko v ladji hkrati znašla oba robota. Podobno kot pri algoritmu iz besedila naloge bomo tudi tu predpostavili, da si globalne spremenljivke delita oba robota, dostop do njih pa je izveden tako, da robota ne moreta oba hkrati dostopati do ene in iste spremenljivke.

```
const Jaz = ...;
var Hocem: array [1..2] of boolean value [1..2: false];
    naVrsti: 1..2 value 1;

procedure Raztovarjanje;
begin
```

```
  repeat
    Hocem[Jaz] := true;
    naVrsti := 3 − Jaz;
    while Hocem[3 − Jaz] and (naVrsti = 3 − Jaz) do begin end;
    stopi v ladjo, poberi zaboj, izstopi iz ladje;
    Hocem[Jaz] := false;
  until LadjaPrazna;
end; {Raztovarjanje}
```

Robot n lahko prek vrednosti Hocem[n] pove, da bi bil rad zdaj v ladji (oz. bi rad vstopil vanjo, če je zaenkrat še zunaj). Če hočeta v ladjo oba robota, bo vanjo vstopil tisti, čigar oznako vsebuje spremenljivka naVrsti — kateri je to, je odvisno od tega, kateremu je uspelo prej izvesti vrstico naVrsti := 3 − Jaz.

Ko hoče na primer eden od robotov stopiti v ladjo, to najprej najavi v tabeli Hocem, nato pa postavi naVrsti na oznako drugega robota in dá s tem tudi njemu možnost stopiti v ladjo. (1) Če je drugi robot takrat že v ladji, bo prvi robot čakal v svoji zanki **while**, dokler ne bo drugi ob izstopu iz ladje postavil svoje Hocem na false. (2) Če drugi robot takrat čaka na vstop (v svoji zanki **while**, bo zdaj lahko vstopil, naš prvi robot pa bo začel v svoji zanki **while** čakati na njegov izstop, tako kot pri primeru (1). (3) Če je drugi robot takrat zunaj ladje in si še ne prizadeva vstopiti vanjo, je njegov Hocem že zdaj false in bo lahko naš prvi robot takoj vstopil. (4) Če pa je drugi robot sicer že postavil svojo Hocem na true, ni pa še izvedel druge vrstice, s katero bi postavil naVrsti na oznako prvega robota, bo naš prvi robot počakal, dokler drugi ne izvede še tiste druge vrstice, nato pa bo drugi začel čakati v svoji zanki **while**, naš prvi robot pa bo v *svoji* zanki **while** opazil, da pogoj ni več izpolnjen, in bo vstopil v ladjo.[5]

[5] O tem, da ta algoritem res prepreči, da bi se oba robota hkrati znašla v ladji, se lahko prepričamo tudi tako, da sistematično pregledamo vsa možna stanja celotnega sistema. Stanje v tem primeru obsega vrednosti vseh spremenljivk in za vsakega od robotov še trenutno mesto izvajanja v podprogramu Raztovarjanje. Opisani algoritem je predlagal Gary Peterson leta 1981; gl. Wikipedijo *s. v.* Peterson's algorithm in njegov članek *Myths about the mutual exclusion problem*, Information Processing Letters, 12(3):115–116, June 1981. Glej tudi Y. Bar-David, G. Taubenfeld, *Automatic discovery of mutual exclusion algorithms*, Proc. DISC 2003; v tem članku avtorja opisujeta, kako sta s sistematičnim generiranjem in preverjanjem vseh možnih algoritmov (do določene dolžine) našla še veliko podobnih sinhronizacijskih postopkov, ki delujejo prav tako dobro kot Petersenov.