13. republiško tekmovanje v znanju računalništva (1989)

NALOGE ZA PRVO SKUPINO

1989.1.1 V tabeli velikosti 10 × 10 znakov nastavi vsem elementom začetne vrednosti `'.'` (pika). Nato v tabelo na naključna mesta shrani števila od 1 do 9 tako, da številke niso v sosednjih elementih. Uporabi deklaracije:

Rešitev: str. 4

```
type Tabela = array [0..9, 0..9] of char;
var t: Tabela;
function Random(Obmocje: integer): integer; external;
```

Funkcija Random vrne naključne vrednosti v mejah $0 \le x <$ Obmocje.

1989.1.2 **Program** naj prebere sto celih in sto realnih števil. Za tem naj prebere število stolpcev, v katerih naj nato ta števila izpiše urejena po velikosti od najmanjšega do največjega. (Izpisuje naj jih po vrsticah in v vsaki vrstici od leve proti desni, ne najprej po stolpcih in pri vsakem stolpcu od zgoraj navzdol.) Števila, ki jih je prebral kot realna, naj izpiše na dve decimalki natančno. Če se neko število pojavi večkrat, naj ga izpiše le enkrat; če se pojavi tako med celimi kot med realnimi, naj ga izpiše kot celo število (torej brez decimalne vejice in ničel za njo).

Rešitev: str. 5

Primer (s tremi celimi in štirimi realnimi števili):

```
  10   5   10   -15.5   -10.4   5.0   20.6
```

izpiše v treh stolpcih

```
-15.50  -10.40       5
    10   20.60
```

1989.1.3 Na voljo imaš niz N^2 znakov ter matriko znakov $N \times N$. **Napiši program**, ki znake iz niza prepiše v matriko tako, da bodo tvorili spiralo z začetkom v gornjem levem kotu. Spirala mora teči v smeri urinega kazalca.

Rešitev: str. 7

Primer: `qwertyuiopasdfgh` prepiši v

```
q w e r
s d f t
a h g y
p o i u
```

1989.1.4 **Napiši proceduro**, ki bo z besedami izpisala števila od vključno 0 do 999.

Rešitev: str. 8

NALOGE ZA DRUGO SKUPINO

Rešitev: str. 9

1989.2.1

Napiši proceduro, ki iz datoteke prebere največ deset trojic ⟨*ime, rezultat, nivo*⟩. Trojice so urejene po padajočem vrstnem redu, ključ urejanja je *rezultat.* V zaporedje na pravo mesto dodaj novo trojico, podano kot argument procedure. Novo zaporedje zapiši nazaj v isto datoteko. Če vsebuje novo zaporedje več kot deset trojic, zapiši le prvih deset. Uporabi naslednje deklaracije:

```
type Top = record
             Ime: array [1..20] of char;
             Rezultat, Nivo: integer;
           end;
var f: file of Top;
```

Rešitev: str. 10

1989.2.2

Podana je dovolj velika količina šifriranega besedila. Za šifriranje je bila uporabljena metoda, pri kateri vsako črko zamenjamo z istoležno črko iz ključa. Ključ vsebuje vse črke abecede, vsako natanko enkrat. Primer šifriranja:

Besedilo:	`to besedilo bo sifrirano s spodnjim kljucem`
Ključ:	`abcdefghijklmnopqrstuvwxyz` `qazwsxedcrfvtgbyhnujmikolp`
Šifrirano besedilo:	`jb asuswcvb ab ucxncnqgb u uybwgrct fvrmzst`

Napiši proceduro Desifriraj, ki bo brez poznavanja šifrirnega ključa po najboljših močeh dešifrirala to besedilo. Šifrirano besedilo vsebuje samo male črke brez ostalih znakov! Besedilo je napisano v angleščini, katere abeceda vsebuje 26 črk!

```
type Ver = array [1..26] of record
                 c: char;      { črka }
                 v: real;      { pogostost črke v angleškem besedilu, }
               end;                          { podana v odstotkih }
procedure Desifriraj(var f: text;   { datoteka s šifriranim besedilom }
                         v: Ver);     { tabela pogostosti črk }
```

Tabela Ver vsebuje podatke o pogostosti črk v tipičnem angleškem besedilu. Črke so urejene po padajoči pogostosti, pogostosti pa so izražene v odstotkih (od 0 do 100). Dešifrirano besedilo izpisuj na zaslon.

Rešitev: str. 13

1989.2.3

Napiši podprogram, ki dano vrednost, zapisano v danem številskem sistemu, pretvori v drugi številski sistem. Najvišja številska osnova je 36; za števke uporabljamo poleg običajnih števk `0`, …, `9` še velike črke angleške abecede, torej `A`, …, `Z`. Tvoj podprogram naj ustreza naslednjim deklaracijam:

```
const MaxDolzina = ...;
type Niz = packed array [1..MaxDolzina] of char;
procedure Pretvori(var Stevilo: Niz; IzOsnove, VOsnovo: integer; var Dolzina: integer);
```

Ob klicu bo tvoj podprogram dobil število v tabeli Stevilo; dolgo je Dolzina števk in zapisano v številskem sistemu z osnovo IzOsnove. Ob vrnitvi iz podprograma naj bo v spremenljivki Stevilo isto število, zapisano v številskem sistemu z osnovo VOsnovo, v spremenljivki Dolzina pa število njegovih števk v tem novem zapisu.

1989.2.4 V tabeli dimenzij $N \times M$ naredi labirint brez vhoda in izhoda. Številka 0 v tabeli predstavlja zid, enica pa pot. Poti so lahko široke en element, labirint pa mora biti enostaven, poln[1] in acikličen (brez krožnih poti). Zato sta dimenziji N in M lihi števili.

Rešitev: str. 14

NALOGE ZA TRETJO SKUPINO

1989.3.1 **Napiši proceduro** Isci, ki bo po datoteki iskala zaporedja zlogov (bytov). Iskana zaporedja (rečemo jim tudi „vzorci“) so lahko dolga največ 100 zlogov in jih je lahko največ 20. Procedura mora najti vsa zaporedja, vsebovana v datoteki, in izpisati, katero zaporedje se začne na najdenem mestu. Primer izpisa:

Rešitev: str. 19

```
zaporedje 3 odmik 45312
zaporedje 5 odmik 51200
. . . . . . . . . . . . .
```

```
type Vzorci = array [1..20] of record
                 Zlog: array [1..100] of byte;      { iskani vzorec }
                 Dolzina: integer;                  { dolžina vzorca }
               end;

{ Podprogram naj v datoteki f poišče pojavitve vzorcev v[1], ..., v[n]. }
procedure Isci(var f: file of byte; v: Vzorci; n: integer);
```

1989.3.2 V tabeli $N \times M$ so zapisane nepravilne površine. **Napiši proceduro**, ki bo označila notranji rob ene površine. Iskana površina je določena s parametroma x in y, ki označujeta koordinati začetka površine (najbolj leva gornja točka). Površine so označene s pozitivnimi vrednostmi, praznine pa z nič; robove površin naj tvoj podprogram označi z -1.

Rešitev: str. 20

```
var Tabela: array [1..N, 1..M] of integer;
procedure Obrobi(x, y: integer);
```

1989.3.3 **Napiši program**, ki prebere število točk v ravnini. Za tem prebere koordinate vseh točk in izpiše najmanjše število premic, ki jih potrebujemo, da vsako izmed danih točk pokrijemo z vsaj eno premico. Pri izračunu zaradi realnih števil upoštevamo točko kot krog s polmerom 10^{-4}.

Rešitev: str. 21

[1] Takšno besedilo naloge se nam je ohranilo v biltenu s tekmovanja leta 1989. Ni čisto očitno, kaj je mišljeno z „enostaven“ in „poln“. Verjetno „enostaven“ pomeni, da se mora dati priti (po samih enicah) od vsake enice do vsake druge enice — z drugimi besedami, labirint ni sestavljen iz več nepovezanih delov. „Poln“ pa verjetno pomeni, da nočemo imeti kakšnih debelih zidov, npr. kvadrat 2×2 samih ničel, ampak mora biti labirint maksimalno prepreden s potmi. — Nerodno pri tej nalogi pa je tudi to, da je težko natančno definirati, kaj točno je labirint. Konec koncev bi lahko naredili eno samo dolgo kačasto pot, ki bi se vila cikcak gor in dol po celi tabeli, kar bi brez dvoma ustrezalo vsem zahtevam te naloge, obenem pa bi se vsakdo strinjal, da je to presneto klavrn labirint.

Rešitev: str. 26

1989.3.4 **Napiši funkcijo** Eval, ki izračuna vrednost podanega aritmetičnega izraza. Izraz je zapisan v datoteki f kot zaporedje znakov, končano z znakom za konec vrstice (konstanta EOL). Izraz lahko vsebuje realne konstante, operatorje `*`, `/`, `+`, `-`, unarni minus (negativni predznak) ter oklepaje. Funkcija kot svojo vrednost vrne vrednost izraza. Realna števila so lahko zapisana v decimalni obliki (`13.4`, `.6`, `12`, `12.`).

Primer:

```
14*(0.5+.01)      --5
```

vrne vrednost

```
12.14
```

```
function Eval(var f: text): real;
```

REŠITVE NALOG ZA PRVO SKUPINO

Naloga: str. 1

R1989.1.1 Števke lahko vpisujemo v tabelo eno za drugo; pri vsaki si naključno izberemo koordinati (x, y) celice, kamor jo bomo vpisali. Nato moramo še preveriti, če v tej ali sosednjih celicah (x1, y1) res ni še nobenega števila. Pri tem moramo paziti na možnost, da smo si izbrali celico na robu tabele in zato sosed v kakšnih smereh sploh nima. Bilo bi prikladno, če bi lahko tip Tabela deklarirali kot **array** [−1..10, −1..10] **of** char in v prvo ter zadnjo vrstico ter stolpec postavili pike, tako da bi tiste celice delovale kot neke vrste stražarji. Ker pa naloga že določa, da naj bo Tabela le tabela 10×10, bomo morali pač posebej preverjati koordinate sosednjih celic, preden jih bomo poskusili pogledati v tabeli.

Po vsakem vpisu neke števke v tabelo postane največ devet celic neprimernih za vpis nadaljnjih števk. Ker imamo sto celic in devet števk, nam torej ni treba skrbeti, da bi v tabeli zmanjkalo primernih položajev, še preden bi nam uspelo vpisati vse števke.

```
program StevilkeVTabeli(Input, Output);
var Tabela: array [0..9, 0..9] of char;
    Stevka: char;
    x, y, x1, y1: integer;
    OK: boolean;
begin
  for x := 0 to 9 do
    for y := 0 to 9 do
      Tabela[x, y] := '.';
  for Stevka := '1' to '9' do
    repeat
      { Naključno izberimo nek položaj v tabeli. }
      x := Random(10); y := Random(10);
      { Preverimo, če so ta celica in njene sosede proste. }
      OK := true;
      for x1 := x - 1 to x + 1 do
        for y1 := y - 1 to y + 1 do
          if (x1 >= 0) and (x1 <= 9) and
             (y1 >= 0) and (y1 <= 9) then
```

```
        if Tabela[x1, y1] <> '.' then OK := false;
      { Če je vse v redu, vpišimo trenutno števko v to celico. }
      if OK then Tabela[x, y] := Stevka;
    until OK;
  for x := 0 to 9 do begin
    for y := 0 to 9 do Write(Tabela[x, y]:2);
    WriteLn;
  end; {for}
end. {StevilkeVTabeli}
```

R1989.1.2

Naloga: str. 1

Naloga zahteva za števila, ki smo jih prebrali kot realna, drugačen izpis kot za tista, ki smo jih prebrali kot cela. Lahko bi prebrali vse v tabelo spremenljivk tipa real, jih uredili, odstranili duplikate in nato pred izpisom vsakega števila pogledali, če je celo ali ne; vendar bi se lahko zgodilo, da bi kakšno število, ki je po vrednosti sicer celo, prebrali le v okviru branja realnih (kjer je bilo npr. podano kot `12.0`) in bi ga morali zdaj (če se hočemo res natančno držati navodil naloge) izpisati na dve decimalki, saj je bilo pač prebrano kot realno. Torej bi si morali pravzaprav že ob branju pri vsakem številu zapomniti, ali smo ga prebrali kot realno ali kot celo, in potem te podatke prenašati naokoli tudi pri urejanju; pri odstranjevanju duplikatov pa paziti na primere, ko se neko število pojavlja tako med celimi kot med realnimi; takšno število moramo obdržati kot celo, ker naloga v takem primeru zahteva, naj ga izpišemo kot celo. Da ne bomo zapletali na ta način, bomo števila raje hranili v dveh tabelah, eni za cela in eni za realna; vsako posebej bomo uredili in odstranili duplikate, nato pa ju pri izpisu „zlivali“ — v vsakem koraku pogledamo, katera od njiju bi nam dala na naslednjem mestu manjše število; tako lahko zagotovimo, da bo izpis kot celota pravilno urejen, pa še v primeru, da se neko število pojavi v obeh, lahko poskrbimo, da ga bomo izpisali kot celo število.

Za urejanje tabele števil obstaja veliko postopkov, ker pa bosta naši dve tabeli majhni, bomo uporabili kar enega od najpreprostejših — urejanje z izbiranjem (*selection sort*). Najprej poiščemo najmanjše število v tabeli in ga postavimo na prvo mesto; nato poiščemo najmanjše med preostalimi števili in ga postavimo na drugo mesto; tako nadaljujemo, dokler ne poiščemo na koncu manjšega od največjih dveh števil in ga postavimo na predzadnje mesto; ostane le še eno število, ki je največje in je že na pravem mestu. Za prestavljanje elementov po tabeli uporabljamo „zamenjave“, torej operacije tipa t := x[i]; x[i] := x[j]; x[j] := t, kjer je t pomožna spremenljivka. Po teh treh prirejanjih je v celici x[i] tista vrednost, ki je bila prej v x[j], in obratno. Med urejanjem lahko tudi izločamo duplikate: če opazimo, da sta dve števili enaki, lahko eno od njiju zavržemo iz tabele (čezenj na primer napišemo tisto, ki je bilo prej v zadnji celici tabele, nato pa zmanjšamo števec elementov za 1). Zaradi izločanja duplikatov se lahko zaporedji števil skrajšata; namesto StCelih in StRealnih imamo le še nc celih in nr realnih števil.

Pri izpisu se pomikamo po tabeli celih števil x z indeksom i, po tabeli realnih števil y pa z indeksom j. Če ni še nobeden od teh dveh indeksov prišel do konca tabele, primerjamo naslednje celo število, x[i], in naslednje realno, y[j]; če je celo manjše ali enako, bomo izpisali tega; če sta enaki, moramo y[j] kar preskočiti (tako j povečamo j). Če pa smo pri eni od tabel že prišli do konca (i > nc ali pa j > nr), moramo pač izpisati naslednje število iz druge tabele. Ko pridemo do konca obeh tabel, končamo. Z zastavico IzpC si zapomnimo, ali bo treba izpisati celo število ali ne. Števec StIzp pa pove, v katerem stolpcu smo; z njegovo pomočjo vemo, kdaj bo treba iti v novo vrstico.

```
program Urejanje(Input, Output);
const StCelih = 100;
      StRealnih = 100;
var x: array [1..StCelih] of integer;
    y: array [1..StRealnih] of real;
    i, j, nc, nr, StStolpcev, StIzp, x1: integer; y1: real;
    IzpC: boolean;
begin

  { Preberimo podatke. }
  for i := 1 to StCelih do
    begin Write(i, '-to celo število je: '); ReadLn(x[i]) end;
  for i := 1 to StRealnih do
    begin Write(i, '-to realno število je: '); ReadLn(y[i]) end;
  ReadLn(StStolpcev);

  { Uredimo podatke — cele posebej in realne posebej. }
  nc := StCelih; i := 1;
  while i < nc do begin
    { V x[i] hočemo dobiti najmanjšega od elementov x[i..nc]. }
    j := i + 1;
    while j <= nc do
      { Spotoma še brišimo duplikate: če sta dve števili enaki,
        eno pobrišimo in skrajšajmo zaporedje. }
      if x[j] = x[i] then begin x[j] := x[nc]; nc := nc − 1 end
      else begin
        if x[j] < x[i] then begin x1 := x[j]; x[j] := x[i]; x[i] := x1 end;
        j := j + 1;
      end; {if}
    i := i + 1;
  end; {while}
  nr := StRealnih; i := 1;
  while i < nr do begin
    { V y[i] hočemo dobiti najmanjšega od elementov y[i..nc]. }
    j := i + 1;
    while j <= nr do
      if y[j] = y[i] then begin y[j] := y[nr]; nr := nr − 1 end
      else begin
        if y[j] < y[i] then begin y1 := y[j]; y[j] := y[i]; y[i] := y1 end;
        j := j + 1;
      end; {if}
    i := i + 1;
  end; {while}

  { Lotimo se izpisovanja. }
  i := 1; j := 1; StIzp := 0;
  while (i <= nc) or (j <= nr) do begin
    { Naj izpišemo naslednje celo število ali naslednje realno? }
    IzpC := false;
    if j > nr then IzpC := true { Če so ostala le še cela, bo treba izpisati celo. }
    else if i <= nc then
```

```
      { Imamo tako cela kot realna; poglejmo, katero je manjše. }
      if x[i] <= y[j] then begin
        IzpC := true;
        if x[i] = y[j] then j := j + 1;
      end; {if}
    if StIzp = StStolpcev { če je to potrebno, začnimo novo vrstico }
      then begin WriteLn; StIzp := 1 end
      else StIzp := StIzp + 1;
    if IzpC then { izpis celega števila }
      begin Write(x[i]:8); i := i + 1 end
    else { izpis realnega števila }
      begin Write(y[j]:8:2); j := j + 1 end;
  end; {while}
  WriteLn;
end. {Urejanje}
```

R1989.1.3

Naloga: str. 1

Spiralo, v katero moramo postaviti črke, si lahko predstavljamo kot sestavljeno iz več kvadratnih okvirjev (z robom, širokim eno celico), ki so vloženi eden v drugega. Trenutni kvadrat pokriva vrstice in stolpce First..Last. Na začetku je First enak 1, Last pa n, nato pa First povečujemo in Last zmanjšujemo. V spremenljivki Smer si zapomnimo, katero stranico trenutnega kvadrata bomo risali v nadaljevanju (0 = zgornjo, 1 = desno, 2 = spodnjo, 3 = levo), v spremenljivki p pa indeks znaka, ki ga bomo naslednjega vpisali v matriko. Za oglišča kvadrata je pravzaprav vseeno, h kateri stranici jih štejemo, važno je le, da vsako oglišče le k eni stranici. Spodnji program šteje zgornji dve oglišči k zgornji, spodnji dve pa k spodnji stranici.

```
program Spirala(Output);
const n = 5;
var Niz: packed array [1..n * n] of char;
  Matrika: array [1..n, 1..n] of char;
  x, y, First, Last, p, Smer: integer;
begin
  Niz := 'ABCDEFGHIJKLMNOPQRSTUVWXY';
  x := 1; y := 1; p := 1;
  Smer := 0; First := 1; Last := n;
  repeat
    case Smer of
      0: for x := First to Last do
        begin Matrika[x, First] := Niz[p]; p := p + 1 end;
      1: for y := First + 1 to Last − 1 do
        begin Matrika[Last, y] := Niz[p]; p := p + 1 end;
      2: for x := Last downto First do
        begin Matrika[x, Last] := Niz[p]; p := p + 1 end;
      3: for y := Last − 1 downto First + 1 do
        begin Matrika[First, y] := Niz[p]; p := p + 1 end;
    end; {case}
    Smer := (Smer + 1) mod 4;
    if Smer = 0 then begin Last := Last − 1; First := First + 1 end;
  until p > n * n;
  { Izpišimo matriko na zaslon, čeprav naloga tega pravzaprav ne zahteva. }
  for y := 1 to n do begin
    for x := 1 to n do Write(Matrika[x, y]);
```

```
    WriteLn;
  end; {for}
end. {Spirala}
```

Naloga: str. 1

R1989.1.4

Vsako število najprej razbijmo na posamezne števke (S za stotice, D za desetice in E za enice). Enice so ostanek po deljenju z 10; če pa nato vzamemo količnik po deljenju z 10 in še njega delimo z 10, so desetice ostanek po tem drugem deljenju, stotice pa količnik.

Koristno je imeti pri roki imena posameznih števk. To bo opravil program IzpisiStevko, ki mu lahko tudi povemo, ali naj 2 piše kot *dve* (npr. v 2, 102 ali 200) ali kot *dva* (npr. v 12, 20, 32), poleg tega pa zna za števko izpisati še dani niz ZaStevko.

Zapis števila je iz dveh delov: najprej stotice, nato desetice in enice (npr. *sto triindvajset*). Če sta prisotna oba dela, je vmes presledek. Pri izpisu stotic je *sto* poseben primer, ostala imena stotic pa lahko sestavimo iz imena števke D1 in niza `sto`. Pri izpisu desetic in enic je treba posebej obdelati primer, ko desetic sploh ni (izpišemo le enice), in primer, ko je D2 = 1 (takrat izpišemo enice in *najst*; izjemi sta *deset* in *enajst*); drugače pa izpišemo najprej enice, nato *in*, desetice in še *jset* (za 20..29) ali *deset* (za 30..99).

```
procedure IzpisiStevko(Stevka: integer; Dva: boolean; ZaStevko: string);
begin
  case Stevka of
  1: Write('ena');
  2: if Dva then Write('dva') else Write('dve');
  3: Write('tri');
  4: Write('štiri');
  5: Write('pet');
  6: Write('šest');
  7: Write('sedem');
  8: Write('osem');
  9: Write('devet');
  end; {case}
  Write(ZaStevko);
end; {IzpisiStevko}

procedure IzpisiStevilo(Stevilo: integer);
var S, D, E: integer;
begin
  if Stevilo = 0 then
    Write('nič')
  else begin
    E := Stevilo mod 10;
    Stevilo := Stevilo div 10;
    D := Stevilo mod 10;
    S := Stevilo div 10;
    case S of
      1: Write('sto');
      2..9: IzpisiStevko(S, false, 'sto');
    end; {case}
    if (S > 0) and (D + E > 0) then Write(' ');
    case D of
      0: if E > 0 then IzpisiStevko(E, false, '');
```

```
      1: case E of
            0: Write('deset');
            1: Write('enajst');
            2..9: IzpisiStevko(E, true, 'najst');
         end;
      2..9:
         begin
            if E > 0 then IzpisiStevko(E, true, 'in');
            IzpisiStevko(D, true, '');
            if D = 2 then Write('jset') else Write('deset');
         end;
    end; {case}
  end; {if}
  WriteLn;
end; {IzpisiStevilo}

procedure Izpisi1000Stevil;
var Stevilo: integer;
begin
  for Stevilo := 0 to 999 do IzpisiStevilo(Stevilo);
end; {Izpisi1000Stevil}
```

REŠITVE NALOG ZA DRUGO SKUPINO

R1989.2.1 Sproti, ko beremo zapise iz datoteke, lahko tudi primerjamo njihove rezultate s tistim pri novem zapisu; slednjega vrinemo v zaporedje pred prvi tak zapis, ki ima manjši rezultat. Zastavica Vrinjen nam pove, če smo ga že vrinili v zaporedje — da ga ne bi slučajno še enkrat. Če ga ne uspemo vriniti v zaporedje, ga dodamo na konec — očitno ima slabši rezultat od vseh iz datoteke. Na koncu shranimo zapise v datoteko. Na koncu shranimo zapise v datoteko — vse, če jih je deset ali manj, sicer pa le prvih deset.

Naloga: str. 2

```
type Top = record
              Ime: array [1..20] of char;
              Rezultat, Nivo: integer;
            end;
     TopFile = file of Top;

procedure InTop10(var f: TopFile; Novi: Top);
var Zadnji, Tekoci: integer;
    Vrinjen: boolean;
    Top10: array [1..11] of Top;
begin
  Vrinjen := false;
  Reset(f);
  Zadnji := 0;
  while not Eof(f) and (Zadnji < 10) do begin
    Zadnji := Zadnji + 1;
    Read(f, Top10[Zadnji]);
    if (Top10[Zadnji].Rezultat < Novi.Rezultat) and not Vrinjen then begin
      { Novi zapis bo treba vriniti pred ravnokar prebranega. }
      Top10[Zadnji + 1] := Top10[Zadnji];
```

```
        Top10[Zadnji] := Novi;
        Zadnji := Zadnji + 1;
        Vrinjen := true;
      end; {if}
    end; {while}
    { Če novega zapisa še nismo vrinili v zaporedje, ga dajmo na konec.
      Če smo prebrali deset zapisov, se bo tam na koncu sicer izgubil,
      če pa smo jih prebrali manj, bo prišel še prav. }
    if not Vrinjen then begin
      Zadnji := Zadnji + 1;
      Top10[Zadnji] := Novi;
    end; {if}
    { Največ deset zapisov shranimo nazaj v datoteko. }
    if Zadnji > 10 then Zadnji := 10;
    Rewrite(f);
    for Tekoci := 1 to Zadnji do Write(f, Top10[Tekoci]);
  end; {InTop10}
```

Naloga: str. 2

R1989.2.2

Ker ključa nismo dobili, si bomo morali pomagati s pogostostmi črk. Pri šifriranju se vsaka črka c spremeni v neko drugo črko $f(c)$; različne črke se preslikajo v različne črke. To pomeni, da, če je bila neka c_1 najpogostejša črka v prvotnem besedilu, bo $f(c_1)$ najpogostejša črka v kodiranem besedilu. Podobno velja za drugo najpogostejšo in tako naprej. Če torej poiščemo najpogostejšo črko v kodiranem besedilu, bi morala biti to koda najpogostejše črke prvotnega besedila (torej tiste, ki jo dobimo v v[1].c — v praksi bi bila to črka e). Druga najpogostejša črka v kodiranem besedilu mora biti koda črke v[2].c in tako naprej. Recimo temu postopku postopek 1.

```
procedure Desifriraj1(var f: text; v: Ver);
var
  Pog: array ['a'..'z'] of integer;
  VrstniRed: array [1..26] of char;
  Kod: array ['a'..'z'] of char;
  c: char; i, j: integer;
begin
  { Preštejmo pogostosti črk v datoteki f. }
  for c := 'a' to 'z' do Pog[c] := 0;
  while not Eof(f) do begin
    Read(f, c);
    if (c >= 'a') and (c <= 'z') then
      Pog[c] := Pog[c] + 1;
  end; {while}
  { Uredimo črke po padajoči pogostosti. }
  for i := 1 to 26 do begin
    j := i − 1; c := Chr(Ord('a') + j);
    while j >= 1 do
      if Pog[VrstniRed[j]] >= Pog[c] then break
      else begin VrstniRed[j + 1] := VrstniRed[j]; j := j − 1 end;
    VrstniRed[j + 1] := c;
  end; {for}
  { i-ta najpogostejša črka iz f naj se dekodira v
    i-to najpogostejšo črko iz v. }
  for i := 1 to 26 do
    Kod[VrstniRed[i]] := v[i].c;
```

```
  { Dekodirajmo zdaj vsebino datoteke. }
  Reset(f);
  while not Eof(f) do begin
    Read(f, c);
    if (c >= 'a') and (c <= 'z') then
       c := Kod[c];
    Write(c);
  end; {while}
end; {Desifriraj1}
```

Spodnji program pa uporablja še malo drugačen postopek (recimo mu postopek 2). Za vsako črko kodiranega besedila izračuna njeno pogostost (v odstotkih glede na celotno dolžino kodiranega besedila) in nato poišče najbližjo pogostost med črkami nekodiranih besedil, torej najbližjo med vrednostmi v[i].v. V ta namen okoli vsake v[i].v ustanovi interval, ki sega pol poti do pogostosti prejšnje in naslednje črke, torej od (v[i − 1].v + v[i].v) / 2 do (v[i].v + v[i + 1].v) / 2. Izjema sta najpogostejša in najredkejša črka (pri prvi je zgornja meja kar 100 %, pri zadnji pa je spodnja meja 0 %). Za pogostost vsake črke kodiranega besedila torej pogledamo, v kateri interval pade, in to črko dekodiramo v črko, ki ji pripada najdeni interval. Nerodno pri tem postopku je, da se nam lahko več črk dekodira v isto, če se pogostosti v obdelovanem besedilu dovolj razlikujejo od tistih v prvotnem; mogoče pa to niti ni tako slabo, če bo zato vsaj ena od teh več črk, ki se dekodirajo v isto, dekodirana pravilno.

```
procedure Desifriraj2(var f: text; v: Ver);
type Interval = record c: char; zg, sp: real end;
var Int: array [1..26] of Interval;
    Pog: array ['a'..'z'] of real;

  procedure IzracunajIntervale;
  var i: integer;
  begin
    for i := 2 to 25 do begin
      Int[i].c := v[i].c;
      Int[i].zg := v[i].v + (v[i −1].v − v[i].v) / 2;
      Int[i].sp := v[i].v − (v[i].v − v[i + 1].v) / 2;
    end; {for}
    Int[1].c := v[1].c; Int[1].sp := Int[2].zg; Int[1].zg := 100;
    Int[26].c := v[26].c; Int[26].zg := Int[25].sp; Int[26].sp := 0;
  end; {IzracunajIntervale}

  procedure PrestejCrke;
  var c: char; Sum: real;
  begin
    for c := 'a' to 'z' do Pog[c] := 0;
    Sum := 0;
    while not Eof(f) do begin
      Read(f, c);
      if (c >= 'a') and (c <= 'z') then
        begin Pog[c] := Pog[c] + 1; Sum := Sum + 1 end;
    end; {while}
    if Sum = 0 then Sum := 1;    { Datoteka f je prazna. }
    for c := 'a' to 'z' do
      Pog[c] := 100 * Pog[c] / Sum;
```

```
  end; {PrestejCrke}

  procedure Zamenjaj;
  var c: char;

    function VrniCrko(c: char): char;
    var i: integer; Nasel: boolean;
    begin
      Nasel := false;
      VrniCrko := '?';
      i := 1;
      while not Nasel and (i <= 26) do
        if (Pog[c] >= Int[i].sp) and (Pog[c] <= Int[i].zg) then
          begin Nasel := true; VrniCrko := Int[i].c end
        else i := i + 1;
    end; {VrniCrko}

  begin
    while not Eof(f) do begin
      Read(f, c);
      if (c <= 'z') and (c >= 'a') then
        Write(VrniCrko(c));
    end; {while}
  end; {Zamenjaj}

begin {Desifriraj}
  IzracunajIntervale;
  PrestejCrke;
  Reset(f);
  Zamenjaj;
end; {Desifriraj2};
```

Še ena preprosta, a koristna izboljšava te rešitve bi bila, da ne bi klicali VrniCrko za vsako črko posebej, ampak bi jo poklicali po enkrat za vsako črko od 'a' do 'z' in rezultate shranili v neko tabelo; odtlej pa bi za dekodiranje le uporabili ustrezni element te tabele in ne bi bilo treba več vsakič prečesavati celega seznama intervalov.

Naredili smo preprost poskus. Vzeli smo nekaj besedil, spremenili velike črke v male, odstranili vse nečrkovne znake, uporabili eno besedilo za merjenje pogostosti črk v tabeli v in nato pogledali, kako bi se odrezala gornja dva postopka, če bi bilo treba dekodirati neko drugo besedilo.

Besedilo za tabelo v	Besedilo za dekodiranje	% napačno dekodiranih znakov	
		Postopek 1	Postopek 2
Gibbon 1 (1,27 M)	Gibbon 2 (1,41 M)	49,258	39,600*
Gibbon 1–3 (3,89 M)	Gibbon 4–6 (3,60 M)	24,220*	31,781
David Copperfield (1,49 M)	Nicholas Nickleby (1,43 M)	27,707*	39,816
Dickens 1 (7,63 M)	Dickens 2 (7,56 M)	17,425*	29,140
Reuters 1 (114,0 M)	Reuters 2 (107,5 M)	0,000*	10,142
Reuters 1–2 (221,5 M)	Reuters 3–4 (223,0 M)	17,198*	20,347
Reuters 1–4 (444,5 M)	Reuters 5–8 (446,1 M)	3,201	1,862*
Gibbon 1–6 (7,49 M)	Dickens 1–2 (15,20 M)	53,536*	54,772
Gibbon 1–6 (7,49 M)	Reuters 1–8 (890,6 M)	63,381	61,505*
Dickens 1–2 (15,20 M)	Reuters 1–8 (890,6 M)	70,384	66,052*

* = boljši

V oklepajih so dolžine besedil (v milijonih črk). Odstotki napačno dekodiranih znakov upoštevajo tudi pogostost posamezne črke: na primer, če napačno dekodiramo tisto, v kar se je zakodiral a, nam da to precej več napak kot pa napačno dekodiranje tistega, v kar se je zakodiral z. Besedila so bila: Gibbonova *Zgodovina zatona in propada rimskega cesarstva* (v šestih zvezkih, skupaj 7 492 576 črk); nekaj Dickensovih del (skupaj 15 205 102 črk, ki smo jih razdelili v dve približno enako veliki skupini); in znana zbirka 806 791 Reutersovih člankov (ki smo jih razdelili na 8 delov s po 100 848 ali 100 849 zaporednimi članki). Po starosti si sledijo ta besedila v razmiku približno sto let: Gibbonova *Zgodovina* je bila objavljena v letih 1776–88, Dickensova dela sredi 19. stoletja, Reutersovi članki pa so iz obdobja od 20. avgusta 1996 do 19. avgusta 1997.

Vidimo, da je postopek 1 ponavadi malo boljši, vendar so v praksi napake še vedno kar precejšnje, tudi če imamo veliko besedil in bi torej človek pričakoval, da bi morale biti frekvence precej stabilne. Čeprav so vsa besedila v istem jeziku, se frekvence črk dovolj spreminjajo, da nam to resno otežkoča dekodiranje. Pogosto se na primer zgodi, da imata dve črki zelo podobni frekvenci, tako da je to, katera je malenkost pogostejša od druge, pravzaprav stvar naključja, torej se metoda 1 pri teh dveh črkah čisto lahko zmoti; podobno pa se lahko frekvence hitro spremenijo vsaj za toliko, da se zmoti pri kakšni od teh črk tudi metoda 2. Levji delež napake pri dekodiranju Reuters 3–4 s frekvencami iz Reuters 1–2 izvira ravno od tega, ker je na Reuters 1–2 druga najpogostejša črka t (8,62 %), tretja pa a (8,59 %), na Reuters 3–4 pa je ravno obratno (a 8,64 %; t 8,55 %), tako da pri dekodiranju oba postopka pobrkljata a in t.

Deset najpogostejših črk (in njihove pogostosti v promilih):

Gibbon	e 130	t 94	a 79	o 76	i 75	s 67	n 67	r 63	h 57	d 41
Dickens	e 122	t 89	a 80	o 77	n 71	i 70	s 62	h 62	r 59	d 45
Reuters	e 119	a 86	t 86	n 76	i 75	o 73	s 70	r 68	l 42	d 41

Najbrž bi si bilo pri dekodiranju pametno pomagati še s slovarjem angleških besed — ko razmišljamo o tem, v kaj bi dekodirali neko črko, bi bilo dobro preveriti, da nam ne bo nastalo v dekodiranem besedilu ogromno besed, ki jih ni v slovarju. Zmedi med črkama a in t bi se lahko izognili s pomočjo dejstva, da je *the* v angleščini običajno daleč najpogostejša tročrkovna beseda (pravzaprav najpogostejša beseda sploh), tako da ni težko ugotoviti, v kaj se je zakodiral t.

R1989.2.3

Naloga: str. 2

V številskem sestavu z osnovo b imajo števke vrednosti $0, 1, \ldots, b-1$. Če številu z vrednostjo n na desni pripišemo števko c, se mu vrednost spremeni v $n \cdot b + c$. Zato ni težko izračunati, kakšno vrednost predstavlja neko število, zapisano v b-iškem sestavu: začnemo z 0, nato pa beremo števke od leve proti desni, število vsakič pomnožimo z b in mu prištejemo pravkar prebrano števko.

Za pretvorbo v obratno smer, torej vrednosti v zaporedje števk, moramo izvajati obratne operacije. Zadnja (najbolj desna) števka števila n je (če je $n \geq 0$) kar ostanek po deljenju n-ja z osnovo b, torej $n \bmod b$. Celi del količnika n/b pa je ravno število n brez te skrajno desne števke (saj za $n \operatorname{div} b$ res velja, da če mu pripišemo števko $n \bmod b$, dobimo n: $n = (n \operatorname{div} b) \cdot b + (n \bmod b)$). Tako lahko pulimo iz n-ja števke eno za drugo, od desne proti levi. Pred izpisom bomo morali njihov vrstni red še obrniti, saj moramo najprej izpisati bolj leve števke (tiste z največjo težo).

```
const MaxDolzina = 15;
type Niz = packed array [1..MaxDolzina] of char;

procedure Pretvori(var Stevilo: Niz; IzOsnove, VOsnovo: integer; var Dolzina: integer);
const Stevke = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
var i, T, Vrednost: integer;
begin
  { Izračunajmo vrednost, ki jo predstavlja niz Stevilo. }
  Vrednost := 0;
  for i := 1 to Dolzina do begin
    T := 0;   { Katero števko predstavlja znak Stevilo[i]? }
    while (T < IzOsnove) and (Stevke[T + 1] <> Stevilo[i]) do
      T := T + 1;
    if T >= IzOsnove then
      begin WriteLn('Neveljavna števka: ', Stevilo[i]); T := 0 end;
    Vrednost := Vrednost * IzOsnove + T;
  end; {for}

  { Koliko števk bo imelo to število v zapisu z osnovo VOsnovo? }
  Dolzina := 0;
  T := Vrednost;
  repeat
    T := T div VOsnovo;
    Dolzina := Dolzina + 1;
  until T = 0;

  { V niz Stevilo vpišimo predstavitev števila Vrednost z osnovo VOsnovo. }
  for i := Dolzina downto 1 do begin
    Stevilo[i] := Stevke[(Vrednost mod VOsnovo) + 1];
    Vrednost := Vrednost div VOsnovo;
  end; {for}
end; {Pretvori}
```

Naloga: str. 3

R1989.2.4

Za začetek bomo ves labirint zazidali — v vse celice postavimo zidove. Potem se bomo postavili v nek začetni položaj, ga „odzidali“ in se začeli premikati iz njega v sosednje celice ter pri tem spreminjati zidove v poti. To bo zagotovilo, da bo labirint povezan. Na vsakem položaju naredimo naslednje: če sta v kakšni od štirih smeri (gor, dol, levo, desno) ob trenutni celici dve zaporedni zazidani celici, ju obe spremenimo v prosti celici in se premaknemo v drugo od njiju (na primer, če smo na (x, y) in sta tako $(x, y-1)$ kot $(x, y-2)$ zazidani, ju spremenimo v prosti in se premaknimo v $(x, y-2)$); če je mogoč tak premik v več smeri, si smer izberemo naključno; če pa ni nobene take smeri, se umaknemo nazaj (po poti, po kateri smo prišli) na položaj, iz katerega smo prvikrat prišli v trenutni položaj. Da se ne bo dalo priti ven iz labirinta, morajo ostati prvi in zadnji stolpec in vrstica popolnoma zazidani; to zagotovimo, če se delamo, da so zunaj labirinta same poti.

Označimo x-koordinate celic z $0, \ldots, m-1$ in y-koordinate z $0, \ldots, n-1$; naj bo (x_0, y_0) naš začetni položaj (ki si ga izberemo naključno; omejitev je le ta, da morata biti, kot bomo videli, obe tidve koordinati lihi). Ker se vedno premaknemo za dve celici naenkrat v isto smer, bodo vsi naši kasnejši položaji oblike (x_0+2a, y_0+2b) za neki celi števili a in b. Navpični hodniki (torej skupine dveh ali več nezazidanih celic, ki se dotikajo z vodoravnimi stranicami) imajo

	15	14	13		113		103	102	85	86	87	88	89	90	91	92	93		101	
	16		12		112		104		84								94		100	
	17		11		111	110	105		83	82	81	80	79		97	96	95	98	99	
	18		10				106						78							
	19		9	8	7		107	108	109		75	76	77		67	66	65	64	63	
	20				6						74				68				62	
	21	22	23		5	4	3	2	1		73	72	71	70	69	114	115		61	
			24																60	
	119		25		31	32	33	34	35		43	44	45		51	52	53	54	55	
	118		26		30				36		42		46		50				56	
	117	116	27	28	29		39	38	37	40	41		47	48	49		59	58	57	

Primer labirinta. Osenčene celice so zazidane, številke v prostih celicah pa povedo, v kakšnem vrstnem redu jih je obiskoval naš program.

torej vedno x-koordinato oblike $x_0 + 2a$ in pokrivajo y-koordinate od $y_0 + 2b$ do $y_0 + 2b'$ za neke a, b, b'. Podobno vodoravni hodniki pokrivajo x-koordinate od $x_0 + 2a$ do $x_0 + 2a'$ pri y-koordinati $y_0 + 2b$, za neke a, a', b.

No, rekli smo, da bo moral ostati zunanji rob (stolpca $x = 0$ in $x = m - 1$ ter vrstici $y = 0$ in $y = n - 1$) labirinta zazidan. Ker so $(0, y)$ sami zidovi, ne smeta biti nobeni dve celici $(1, y)$ in $(1, y + 1)$ hkrati zazidani, ker bi imeli v tem primeru debel zid (kvadrat 2×2 samih zidov), tega pa nočemo. Torej bo vsaj vsaka druga celica tega stolpca prosta (nezazidana); vsaka prosta celica pa je del nekega vodoravnega in/ali nekega navpičnega hodnika. Če je del vodoravnega hodnika, se ta tu konča (saj na $x = 0$ ne more iti), torej je $x = 1$ oblike $x_0 + 2a$; če je del navpičnega hodnika, pa mora spet veljati (spomnimo se lastnosti hodnikov, ki smo jih omenili v prejšnjem odstavku), da je $x = 1$ oblike $x_0 + 2a$. V vsakem primeru sledi $x_0 = 1 - 2a$, torej je x_0 lih. Podoben razmislek bi lahko zdaj začeli tudi iz dejstva, da so v vrhnji vrstici sami zidovi, torej v vrstici $y = 1$ ne moreta biti po dve sosednji celici zazidani, itd., in prišli bi do ugotovitve, da mora biti tudi y_0 lih. (Če bi tako razmišljali še za skrajno desni stolpec, $x = m - 1$, in najnižjo vrstico, $y = n - 1$, bi videli tudi, da mora biti x_0 po parnosti enak vrednosti $m - 2$, y_0 pa vrednosti $n - 2$. Torej morata biti $m - 2$ in $n - 2$ tudi liha, zato pa tudi m in n; no, besedilo naloge na srečo obljublja, da to zagotovo velja.) Kakorkoli že, zdaj smo ugotovili, da morata biti koordinati našega začetnega položaja obe lihi.

Ali se lahko našemu algoritmu zgodi, da v labirintu dobi cikle? Da bi nastal cikel, bi morali v nekem trenutku podreti zid v celici, ki ima že dve nezazidani sosedi. Recimo, da je A naš trenutni položaj in podremo celici B in C. C je očitno pred tem še zazidana, sicer tega sploh ne bi počeli; torej je podiranje zidu B lahko problem le, če je od prej že prosta D ali pa E. Toda da bi bila katera od njiju podrta, bi morala biti del vodoravnega ali pa navpičnega hodnika, toda za vodoravni hodnik ima napačno parnost y-koordinate, za navpičnega pa napačno parnost x-koordinate (ker ima v obeh primerih nasprotno parnost kot ustrezna A-jeva koordinata, ta pa ima pravo, saj je A trenutni položaj). Torej podiranje zidu B ni problem. Kaj pa podiranje zidu C? To bi bil problem, če bi bila od prej prosta katera od F, G in H. F ne more biti del navpičnega hodnika (zaradi parnosti x-koordinate), če pa bi bila del vodoravnega, ne more biti njegov začetek ali konec; torej bi bila v tem primeru C vsekakor že tudi podrta, kar je protislovje. Enak razmislek velja za

```
**H**
*FCG*
*DBE*
**A**
```

G, podoben pa tudi za H, ki zaradi parnosti svoje y-koordinate ne more biti del vodoravnega hodnika niti konec navpičnega, torej bi bila tudi C zagotovo že prosta, če bi bila prosta H. Torej tudi podiranje zidu C ni problem.

Ta razmislek nam zagotavlja tudi, da v labirintu ne bomo dobili velikih soban, npr. kvadrata 2×2 samih nezazidanih celic, saj je to tudi že cikel.

Ali se lahko našemu algoritmu zgodi, da v labirintu pusti debele zidove, torej nekje recimo kvadrat 2×2 zazidanih celic? Recimo, da obstaja tak kvadrat; gotovo si lahko izberemo takega, ki ima vsaj eno prosto sosedo, recimo u. Ker je u prosta, je del vodoravnega in/ali navpičnega hodnika. Če vodoravnega, se ta pri u očitno konča, torej je bil v u nekoč trenutni položaj; toda takrat bi opazili, da sta W in V zazidani, in bi ju prekopali; torej je to nemogoče. Torej je u del navpičnega hodnika; če se ta hodnik konča pri u, je bil nekoč tu trenutni položaj in imamo enak problem kot prej; če pa se ne konča pri u, se mora nadaljevati tudi pri u-jevi spodnji sosedi in je vsaj ena od njiju bila nekoč trenutni položaj in bi morali takrat opaziti in izkopati ali V in W (če je bil trenutni položaj u) ali pa X in Y (če je bil trenutni položaj v u-jevi spodnji sosedi). — Analogno lahko razmišljamo v primeru, ko je u ob zgornji, levi ali desni stranici našega kvadrata.

```
****
*VWu
*XY*
****
```

Tako smo se prepričali, da naš algoritem poišče labirinte, ki ustrezajo vsem zahtevam naloge: povezani, aciklični, brez velikih sob in debelih zidov.

```
const XS = 77; YS = 21;
type Celica = (Zid, Pot);
var Tabela: array [0..XS − 1, 0..YS − 1] of Celica;

function JeZid(x, y: integer): boolean;
begin
  if (x < 0) or (x >= XS) or (y < 0) or (y >= YS) then JeZid := false
  else JeZid := Tabela[x, y] = Zid;
end; {JeZid}

procedure Labirint(x, y: integer);
var nx, ny: integer;
begin
  Tabela[x, y] := Pot;
  while JeZid(x − 2, y) or JeZid(x + 2, y) or
       JeZid(x, y − 2) or JeZid(x, y + 2) do begin
    nx := x; ny := y;
    case Random(4) of
      0: nx := x + 2;
      1: nx := x − 2;
      2: ny := y + 2;
      3: ny := y − 2;
    end; {case}
    if JeZid(nx, ny) then begin
      Tabela[(x + nx) div 2, (y + ny) div 2] := Pot;
      Labirint(nx, ny);
    end; {if}
  end; {while}
end; {Labirint}

procedure ZazidajTabelo;
var x, y: integer;
begin
```

```
  for x := 0 to XS − 1 do
    for y := 0 to YS − 1 do
      Tabela[x,y] := Zid;
end; {ZazidajTabelo}

procedure PrikaziTabelo;
var x, y: integer;
begin
  for y := 0 to YS − 1 do begin
    for x := 0 to XS − 1 do
      if Tabela[x, y] = Zid then Write('#') else Write('.');
    WriteLn;
  end; {for}
end; {PrikaziTabelo}

begin
  ZazidajTabelo;
  Labirint(2 * Random(XS div 2) + 1, 2 * Random(YS div 2) + 1);
  PrikaziTabelo;
end.
```

Zgoraj opisani postopek za speljevanje poti po labirintu se je pravzaprav zgledoval po preiskovanju grafov v globino, le da si naključno izbira, v kakšnem vrstnem redu bo preiskoval sosede trenutnega položaja. Lahko pa bi se namesto po tem zgledovali tudi po Primovem ali pa po Kruskalovem algoritmu za iskanje minimalnih vpetih dreves v grafu. Primov algoritem bi gradil labirint tako, da bi v vsakem koraku naključno izbral eno od prostih polj (x, y), eno od štirih možnih smeri in nato podrl zidova v dveh sosednjih poljih v tej smeri (če je to dopustno, torej če sta na obeh teh dveh poljih res zidova). Kruskalov algoritem pa bi za začetek podrl zidove na vseh poljih, ki imajo obe koordinati lihi, nato pa bi si naključno izbiral po neko polje in neko smer ter podrl zid v sosednjem polju v tisti smeri, če seveda ni mogoče do polja, ki je od trenutnega oddaljeno za dva koraka v tisti smeri, priti že zdaj po kakšni bolj oddaljeni poti). Pri tem algoritmu bi prišla prav znana podatkovna struktura za disjunktne množice,[2] s katero bi lahko učinkoviteje preverjali, ali sta neki dve polji že povezani s potjo ali ne; vendar pa, ker so naši labirinti bolj majhni, uporablja spodnji podprogram kar tabelo, v kateri za vsako točko piše, kateri povezani komponenti labirinta pripada.

```
procedure LabirintPrim(x0, y0: integer);
type TockaT = record x, y: integer end;
var ToDo: array [1..XS * YS] of TockaT;
   nToDo, x, y, nx, ny, i: integer;
begin
  nToDo := 1; with ToDo[1] do begin x := x0; y := y0 end;
  Tabela[x0, y0] := Pot;
  while nToDo > 0 do begin
    { Naključno izberimo kakšno izmed tistih celic, ki imajo še kakšno primerno
      zazidano sosedo. Seznam takih celic je v tabeli ToDo. }
    i := Random(nToDo) + 1; x := ToDo[i].x; y := ToDo[i].y;
    if not (JeZid(x − 2, y) or JeZid(x + 2, y) or
```

[2] Glej npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 22.3 v prvi izdaji, 21.3 v drugi.

```
        JeZid(x, y - 2) or JeZid(x, y + 2)) then begin
      { Vse njene sosede smo že pregledali — pobrišimo jo iz množice ToDo. }
      ToDo[i] := ToDo[nToDo]; nToDo := nToDo - 1;
    end
    { Izberimo eno od zazidanih sosed trenutne celice in tisti zid porušimo.
      Sosedo dodajmo na seznam ToDo, da bomo o priliki pregledali še njene sosede. }
    else while true do begin
      nx := x; ny := y;
      case Random(4) of
        0: nx := x + 2;   1: nx := x - 2;   2: ny := y + 2;   3: ny := y - 2;
      end; {case}
      if JeZid(nx, ny) then begin
        Tabela[(x + nx) div 2, (y + ny) div 2] := Pot;
        Tabela[nx, ny] := Pot;
        nToDo := nToDo + 1; ToDo[nToDo].x := nx; ToDo[nToDo].y := ny;
        break;
      end; {if}
    end; {while}
  end; {while}
end; {LabirintPrim}

procedure LabirintKruskal;
type TockaT = record x, y: integer end;
var ToDo: array [1..XS * YS] of TockaT;
    nToDo, x, y, xt, yt, k, nk, nx, ny, i: integer;
    Komp: array [1..XS, 1..YS] of integer;

  function IstaKomp(x, y, KotKomp: integer): boolean;
  begin
    if (x < 0) or (x >= XS) or (y < 0) or (y >= YS) then IstaKomp := true
    else IstaKomp := Komp[x, y] = KotKomp;
  end; {IstaKomp}

begin
  { Labirint bo med gradnjo razdeljen na „komponente“ — dele labirinta,
    ki so z zidovi povsem ločeni drug od drugega. Na začetku je vsaka celica,
  nToDo := 0;
  for x := 0 to (XS div 2) - 1 do for y := 0 to (YS div 2) - 1 do begin
    nToDo := nToDo + 1; Komp[2 * x + 1, 2 * y + 1] := nToDo;
    ToDo[nToDo].x := 2 * x + 1; ToDo[nToDo].y := 2 * y + 1;
    Tabela[2 * x + 1, 2 * y + 1] := Pot;
  end; {for}

  { Dokler obstaja več kot ena komponenta: izberimo naključno neko celico
    in neko njeno sosedo; če nista v isti komponenti, zid med njima porušimo
    (obe komponenti se s tem zlijeta v eno samo). }
  while nToDo > 0 do begin
    i := Random(nToDo) + 1; x := ToDo[i].x; y := ToDo[i].y; k := Komp[x, y];
    if IstaKomp(x - 2, y, k) and IstaKomp(x + 2, y, k) and
       IstaKomp(x, y - 2, k) and IstaKomp(x, y + 2, k) then begin
      ToDo[i] := ToDo[nToDo]; nToDo := nToDo - 1;
    end
    else while true do begin
      nx := x; ny := y;
      case Random(4) of
```

```
      0: nx := x + 2;   1: nx := x − 2;   2: ny := y + 2;   3: ny := y − 2;
    end; {case}
    { Porušimo zid med celicama in zlijmo njuni komponenti. }
    if not IstaKomp(nx, ny, k) then begin
      Tabela[(x + nx) div 2, (y + ny) div 2] := Pot;
      nk := Komp[nx, ny];
      for xt := 0 to (XS div 2) − 1 do for yt := 0 to (YS div 2) − 1 do
        if Komp[2 * xt + 1, 2 * yt + 1] = nk then
          Komp[2 * xt + 1, 2 * yt + 1] := k;
      break;
    end; {if}
  end; {while}
 end; {while}
end; {LabirintKruskal}
```

Po naših opažanjih imajo labirinti, ki jih pripravlja Primov algoritem, več daljših hodnikov kot tisti pri iskanju v globino; labirinti, dobljeni s Kruskalovim algoritmom, pa so še bolj zaviti kot tisti, dobljeni z iskanjem v globino. Poskusili smo tudi oceniti, kako težko je priti z enega konca labirinta do drugega, torej od polja $(1,1)$ do $(m-2, n-2)$. Pri tem si mislimo, da se v vsakem koraku, če imamo več možnih smeri za nadaljevanje poti, naključno odločimo za eno od tistih, v katere še nismo šli; če pa pridemo v slepo ulico, gremo nazaj v smeri, od koder smo prišli, dokler ne pridemo do prvega takega križišča, iz katerega vodi kakšna še neprehojena smer. Spodnja tabela kaže povprečno število polj, ki smo jih morali pri takem tavanju prehoditi, da smo prišli od zgornjega levega do spodnjega desnega kota; kaže pa tudi število polj pri najkrajši poti. Vse številke so povprečja prek 10000 naključnih labirintov.

Algoritem	Povprečno število korakov pri tavanju	Povprečna dolžina najkrajše poti
Iskanje v globino	$504{,}2 \pm 126{,}1$	$284{,}5 \pm 64{,}4$
Primov algoritem	$434{,}0 \pm 135{,}0$	$106{,}8 \pm 8{,}2$
Kruskalov algoritem	$453{,}2 \pm 95{,}1$	$149{,}0 \pm 19{,}0$

REŠITVE NALOG ZA TRETJO SKUPINO

R1989.3.1

Naloga: str. 3

Ker vemo, da ni nobeden od vzorcev daljši od sto znakov, je dovolj, če si med branjem datoteke zapomnimo le zadnjih sto prebranih znakov. Po branju vsakega znaka pa pojdimo po vseh vzorcih in preglejmo, če se kakšen od njih mogoče pojavi v datoteki tako, da se pojavitev konča prav pri ravnokar prebranem znaku. Zadnjih sto znakov bomo hranili v tabeli Okno, ki jo bomo uporabljali kot krožni pomnilnik: stoprvi znak bomo zapisali v Okno[1] (in s tem povozili prvega, ki pa ga tako ali tako ne bomo več potrebovali), stodrugi znak v Okno[2], dvestodrugega spet v Okno[2] in tako naprej. Seveda moramo biti zato pri delu z indeksi previdni — delati se moramo, kot da pred znakom Okno[1] pride znak Okno[100].

```
const MaxDolz = 100;
type Vzorci = array [1..20] of record
              Zlog: array [1..MaxDolz] of byte;
              Dolzina: integer;
```

```
        end;
    Datoteka = file of byte;

procedure Isci(var F: Datoteka; V: Vzorci; N: integer);
var Okno: array [1..MaxDolz] of byte;
    Polozaj, Konec, i, j, Prebrano: integer;
begin
    Prebrano := 0; Konec := 0;
    while not Eof(F) do begin
     Konec := Konec + 1; if Konec > MaxDolz then Konec := 1;
     Read(F, Okno[Konec]);
     if Prebrano < MaxDolz then Prebrano := Prebrano + 1;
     for i := 1 to N do with V[i] do if Prebrano >= Dolzina then begin
       { S števcem j se bomo sprehajali po trenutnem vzorcu, V[i].Zlog[. . . ],
         s števcem Polozaj pa po vsebini okna (zadnjih V[i].Dolzina prebranih zlogov). }
       Polozaj := Konec − Dolzina + 1; j := 1;
       if Polozaj <= 0 then Polozaj := Polozaj + MaxDolz;
       while j <= Dolzina do begin
         if Okno[Polozaj] <> Zlog[j] then break;
         j := j + 1; Polozaj := Polozaj + 1; if Polozaj > MaxDolz then Polozaj := 1;
       end; {while}
       if j > Dolzina then { Vse se ujema! }
         WriteLn('zaporedje ', i, ' odmik ', FilePos(F) − Dolzina);
     end; {if, with, for}
  end; {while}
end; {Isci}
```

Ta postopek bi lahko še izboljšali, če bi se zgledovali po kakšnem od znanih postopkov za iskanje podnizov v nizih, npr. Knuth-Morris-Prattovem ali Boyer-Moorovem.[3]

Naloga: str. 3

R1989.3.2

Naloga pravi, da že poznamo najbolj levo zgornjo točko površine, ki nas zanima. Ta torej že pripada notranjemu robu površine, od tu naprej pa bo najbolj učinkovito, če se bomo premikali ves čas po tem robu, recimo v smeri urinega kazalca. Vsako polje, ki ga obiščemo, označimo kot rob (vanj vpišemo -1), nato pa se moramo odločiti, v kateri smeri nadaljevati. Možnih smeri je le osem, saj ima vsako polje v karirasti mreži le osem sosedov. Smer naslednjega premika je lahko ista kot smer prejšnjega premika, lahko pa se glede na tisto smer tudi odkloni za 45, 90 ali 135 stopinj v levo ali desno. (Odklon za 180° pa nas ne zanima, saj bi ta že pomenil vrnitev nazaj na prejšnji položaj.) Ker se gibljemo v smeri urinega kazalca in hočemo ostati na robu površine (ne pa iti v notranjost), moramo ves čas siliti karseda v levo. Torej poskusimo smer premika najprej spremeniti za 135° v levo; če opazimo, da bi nas to vrglo iz površine, poskusimo le 90° v levo in tako naprej. Pred prvim premikom, torej ko smo še na začetnem položaju, se lahko delamo, kot da smo sem prišli s premikom v desno.

Spodnji program ima smerne vektorje za vseh osem smeri navedene v smeri urinega kazalca (če si mislimo, da os y kaže navzdol) v tabeli Okolica. Zasuku za 135° v levo tako ustreza premik za tri mesta nazaj po tej tabeli; ali pa, kar je isto, za pet mest naprej (saj je 135° v levo isto kot 225° v desno).

---

[3] Gl. npr. Cormen *et al.*, *Introduction to Algorithms*, 34.4–5 v prvi izdaji, 32.4 v drugi; Boyer-Moore v slednji žal ni opisan.

```
const N = ...; M = ...; var Polje: array [1..N, 1..M] of integer;

{ Pove, če je dana celica prosta. Zunaj polja si mislimo same proste celice. }
function Prosta(X, Y: integer): boolean;
begin
  if (X < 1) or (Y < 1) or (X > N) or (Y > M)
  then Prosta := false else Prosta := Polje[X, Y] = 0;
end; {Prosta}

procedure Meja(ZacX, ZacY: integer);
const Okolica: array [0..7] of record X, Y: integer end =
  ( (X: 0; Y: 1), (X: −1; Y: 1), (X: −1; Y: 0), (X: −1; Y: −1),
    (X: 0; Y: −1), (X: 1; Y: −1), (X: 1; Y: 0), (X: 1; Y: 1) );
var i, LD, X, Y, Xn, Yn, X1, Y1: integer; Prvi: boolean;
begin
  X := ZacX; Y := ZacY; Prvi := true;
  LD := 3; { Delajmo se, da smo na trenutni položaj prišli s premikom v desno. }
  repeat
    Polje[X, Y] := −1;
    { Poglejmo, v kateri smeri moramo nadaljevati. }
    i := 0; while (i < 8) and Prosta(X + Okolica[LD].X, Y + Okolica[LD].Y) do
      begin LD := (LD + 1) mod 8; i := i + 1 end;
    if i >= 8 then break; { Imamo „lik“ velikosti 1 × 1. }
    Xn := X + Okolica[LD].X; Yn := Y + Okolica[LD].Y;
    { Če smo korak (X, Y) → (Xn, Yn) nekoč že naredili, smo
      zdaj prišli že okoli in okoli lika in se moramo ustaviti. }
    if Prvi then begin X1 := Xn; Y1 := Yn; Prvi := false end
    else if (X = X0) and (Y = Y0) and (Xn = X1) and (Yn = Y1) then break;
    X := Xn; Y := Yn;
    { Na naslednjem položaju moramo začeti preizkušati smeri pri LD − 3, }
    LD := (LD + 5) mod 8;                 { torej z zasukom 135° v levo. }
  until false;
end; {Meja}
```

Podprogram Meja si zapomni prvi opravljeni premik (od (X0, Y0) do (X1, Y1)) in se ustavi, ko se ta premik prvič ponovi. Če bi samo preverjali, kdaj se prvič vrnemo na začetni položaj (X0, Y0), bi imeli težave v primerih, ko je treba začetni položaj obiskati večkrat, ker leži istočasno na spodnji in na zgornji meji površine (glej sliko na desni).

```
.....
..##.
.#...
.....
```

Naša rešitev tudi predpostavlja, da površina, ki nas zanima, v notranjosti nima kakšnih lukenj, okoli katerih bi morali tudi označevati rob. Take luknje bi lahko odkrili s pregledovanjem notranjosti površine, nato pa bi za vsako odkrito luknjo po postopku, podobnem kot zgoraj, označili njen zunanji rob, ki je z vidika površine pač notranji rob.

R1989.3.3

Naloga: str. 3

Preprost postopek bi bil naslednji: izberimo si dve točki in potegnimo skoznju premico. Z malo sreče pokriva ta premica še kakšno drugo točko, ne le tistih dveh, iz katerih smo jo dobili. Kakorkoli že, pokrite točke zdaj zbrišimo iz naše množice in se v nadaljevanju postopka na enak način lotimo preostalih točk. Tako pokrivamo točke, dokler ne pokrijemo vseh.

Pri tem nimamo nobenega zagotovila, da bo dobljena rešitev kaj prida. Če imamo n točk, bi se lahko (z nekaj smole) zgodilo, da bi pokrila vsaka naša premica le dve točki (torej jih potrebujemo $\lceil n/2 \rceil$), obenem pa bi obstajala neka rešitev s samo dvema premicama; torej bi bila naša rešitev vsaj $n/4$-krat slabša ob najboljše možne.

Ta postopek lahko izboljšamo, če vedno izberemo tako premico, ki pokrije največ doslej še nepokritih točk. Pokazati je mogoče, da na ta način porabimo v najslabšem primeru $(1 + \ln n)$-krat toliko premic kot pri najboljši možni rešitvi (taki z najmanj premicami). Problem pokrivanja točk s premicami je NP-težak in zaenkrat menda ni znan noben učinkovit postopek s kakšnim boljšim zagotovilom o kakovosti svojih rešitev.[4] Lahko pa bi približne rešitve iskali tudi z raznimi naključnimi postopki, na primer s simuliranim ohlajanjem; to bi verjetno dajalo še boljše rešitve, le da zdaj ne bi imeli teoretičnega zagotovila o kakovosti dobljenih rešitev.

Požrešni postopek bi lahko izvedli takole: če se postavimo v neko točko T in uredimo ostale po kotu, iz katerega se jih vidi iz T, lahko hitro ugotovimo, katere ležijo na istih premicah.[5] To naredimo pri vsaki T in tako dobimo vse premice, za vsako pa tudi vidimo, katere točke ležijo na njej. Obenem si še za vsako točko zapomnimo, na katerih premicah leži. Ko nekaj točk pobrišemo, ker smo jih pokrili z eno od premic, moramo za vse preostale premice zmanjšati število točk, ki ležijo na njih. Da nam bo vedno pri roki podatek o tem, katera premica pokriva največ točk, imamo lahko premice v kopici ali pa za vsako možno število točk nek seznam premic, ki pokrivajo točno toliko točk, in potem ob brisanju točke preselimo ustrezne premice v nižji seznam. Iskanje vseh premic nam bo vzelo $O(n^2 \lg n)$ časa (zaradi urejanja pri vsaki točki), nato pa bomo morali ob brisanju vsake točke iti po vseh premicah, ki so jo pokrivale, teh pa je največ $O(n)$, tako da bo ta del postopka porabil $O(n^2)$ časa. No, če se hočemo dosledno držati navodila iz naloge, češ da naj vsako točko obravnavamo kot krog s polmerom 10^{-4}, si je pri ugotavljanju, kaj vse leži na isti premici, težko pomagati s koti, saj se kota dveh točk glede na T lahko razlikujeta tudi za $90°$, če je ena od njiju zelo blizu T-ja (bližje kot za 10^{-4} na primer). V tem primeru bi še vedno lahko lepo naivno za vsak par točk določili premico skoznju in nato eksplicitno pregledali vse ostale točke, da bi videli, katere še ležijo na premici; to bi nam vzelo $O(n^3)$ časa.

Kako bi preverili, če leži neka točka $\vec{r} = (x, y)$ na isti premici kot točki $\vec{r}_1 = (x_1, y_1)$ in $\vec{r}_2 = (x_2, y_2)$? Lahko se delamo, da imamo 3-d vektorje (z z-koordinato 0) in si pomagamo z vektorskim produktom: če leži $\vec{r}$ na isti

[4] Dokaz NP-težkosti: N. Megiddo, A. Tamir: *On the complexity of locating linear facilities in the plane*, Operations Research Letters, 1(5):194–197 (1982). Ta problem lahko prevedemo na pokrivanje množic (*set covering*) — ustanovimo po eno množico za vsako premico, v njej pa so tiste točke, ki ležijo na tej premici. Za pokrivanje množic pa je znano, da vrača požrešni postopek največ $(1 + \ln n)$-krat slabše rešitve od optimalnih (D. S. Johnson: *Approximation algorithms for combinatorial problems*, Journal of Computer and System Sciences, 9:256–287, 1974; in zgodnejša različica v Proc. STOC 1973, 38–49); pravzaprav smo lahko še natančnejši: če nobena premica ne pokrije več kot m točk, vrne požrešni algoritem rešitev, ki je največ $(\sum_{k=1}^{m} 1/k)$-krat slabša od optimalne (vsota v oklepajih se imenuje „m-to harmonično število“ in znaša približno $\ln m + 0{,}577$).

Če pa uvedemo pri pokrivanju točk s premicami še dodatno zahtevo, da morajo biti premice vzporedne s koordinatnima osema, postane problem lažji in lahko v polinomskem času dobimo optimalne rešitve (R. Hassin, N. Megiddo: *Approximation algorithms for hitting objects with straight lines*, Discrete Applied Mathematics, 30(1):29–42, 1989).

[5] Če se nam ne bi bilo treba ubadati z numeričnimi nenatančnostmi, bi lahko kote metali kar v razpršeno tabelo in tako še lažje ugotovili, ali vidimo več točk pod istim kotom. Če bi bile koordinate naših točk recimo celoštevilske (ali pa vsaj racionalne), bi lahko namesto kotov uporabljali kar smerne koeficiente premic.

premici kot $\vec{r}_1$ in $\vec{r}_2$, sta $\vec{r}_2 - \vec{r}_1$ in $\vec{r} - \vec{r}_1$ vzporedna, tedaj pa je njun vektorski produkt enak 0. Označimo $\vec{r}_2 - \vec{r}_1$ z $\vec{d} = (d_x, d_y)$, vektor $\vec{r} - \vec{r}_1$ pa z $\vec{u} = (u_x, u_y)$. Vektorski produkt $(d_x, d_y, 0) \times (u_x, u_y, 0)$ je $(0, 0, d_x u_y - d_y u_x)$; preveriti moramo torej le, če je $d_x u_y - d_y u_x$ približno enako 0. Naša naloga pravi, naj gledamo točke kot kroge s polmerom $\varepsilon = 10^{-4}$; če bi premik $\vec{u}$ izrazili kot vsoto premika v smeri $\vec{d}$ in premika v smeri pravokotno na $\vec{d}$, bi morali torej zdaj v resnici preverjati, če je ta premik v smeri pravokotno na $\vec{d}$ krajši od ε. Ta premik je v resnici dolg $|\vec{u}| \cdot \sin\alpha$, če je α kót med vektorjema $\vec{u}$ in $\vec{d}$. Vektorski produkt $\vec{d} \times \vec{u}$ pa je po definiciji dolg $|\vec{d}| \cdot |\vec{u}| \cdot \sin\alpha$; torej bo dovolj, če preverimo, ali je $|d_x u_y - d_y u_x| / |\vec{d}| \le \varepsilon$ oz. $(d_x u_y - d_y u_x)^2 \le \varepsilon^2 (d_x^2 + d_y^2)$.

```
program Pokrivanje;

const MaxTock = 10;

type
  PPremica = ↑TPremica;
  PTockaNaPremici = ↑TTockaNaPremici;

  TTocka = record
    x, y: real;                  { koordinati }
    pp: PTockaNaPremici;         { prva premica, ki vsebuje to točko }
  end;

  TTockaNaPremici = record
    it: integer;                 { indeks točke }
    p: PPremica;                 { kazalec na premico }
    pt, nt: PTockaNaPremici;     { prejšnja in naslednja točka na tej premici }
    np: PTockaNaPremici;         { naslednja premica, ki vsebuje to točko }
  end;

  TPremica = record
    pt: PTockaNaPremici;         { prva točka na tej premici }
    n: integer;                  { število točk na tej premici }
    pp, np: PPremica;            { prejšnja in naslednja premica z n točkami }
  end;

var
  { Premice[k] je seznam premic, ki pokrivajo k točk. }
  Premice: array [1..MaxTock] of PPremica;
  t: array [1..MaxTock] of TTocka;    { točke }
  n: integer;                         { število točk }

{ Ali leži k na premici, ki jo določata i in j? }
function NaPremici(i, j, k: integer): boolean;
const eps = 1e−4;
var dx, dy, ux, uy, v, d2: real;
begin
  dx := t[j].x − t[i].x; dy := t[j].y − t[i].y;
  ux := t[k].x − t[i].x; uy := t[k].y − t[i].y;
  v := dx * uy − dy * ux; d2 := dx * dx + dy * dy;
  NaPremici := (d2 > 0) and (v * v <= eps * eps * d2);
end; {NaPremici}

{ Doda v sezname podatek, da premica p pokriva točko it. }
procedure Dodaj(it: integer; p: PPremica);
```

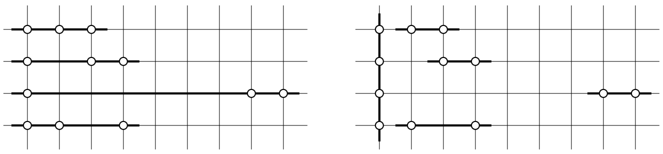
```
var tp: PTockaNaPremici;
begin
  New(tp); tp↑.it := it; tp↑.p := p; p↑.n := p↑.n + 1;
  { Dodajmo jo v seznam točk, ki jih pokriva premica p. }
  tp↑.pt := nil; tp↑.nt := p↑.pt; p↑.pt := tp;
  if tp↑.nt <> nil then tp↑.nt↑.pt := tp;
  { Dodajmo jo v seznam premic, ki pokrivajo točko it. }
  tp↑.np := t[it].pp; t[it].pp := tp;
end; {Dodaj}

{ Zbriše točko it iz vseh premic, ki jo pokrivajo. }
procedure Brisi(it: integer);
var tp: PTockaNaPremici; p: PPremica; n: integer;
begin
  while t[it].pp <> nil do begin
    tp := t[it].pp; p := tp↑.p; t[it].pp := tp↑.np;
    { Zbrišimo točko it iz seznama točk, ki jih pokriva premica p. }
    if tp↑.pt = nil then p↑.pt := tp↑.nt else tp↑.pt↑.nt := tp↑.nt;
    if tp↑.nt <> nil then tp↑.nt↑.pt := tp↑.pt;
    Dispose(tp); n := p↑.n; p↑.n := p↑.n − 1;
    { Zbrišimo p iz seznama premic, ki pokrivajo n točk. }
    if p↑.pp = nil then Premice[n] := p↑.np else p↑.pp↑.np := p↑.np;
    if p↑.np <> nil then p↑.np↑.pp := p↑.pp;
    if n > 1 then begin { Dodajmo p v seznam premic, ki pokrivajo n − 1 točk. }
      p↑.pp := nil; p↑.np := Premice[n − 1];
      if p↑.np <> nil then p↑.np↑.pp := p;
      Premice[n − 1] := p;
    end; {if}
  end; {while}
end; {Brisi}

var i, j, k, pn, StPremic: integer; p: PPremica;
begin
  { Preberimo število točk in njihove koordinate. }
  Write('Število točk: '); ReadLn(n);
  for i := 1 to n do begin
    Write('X[', i, ']: '); ReadLn(t[i].x);
    Write('Y[', i, ']: '); ReadLn(t[i].y);
    t[i].pp := nil;
  end; {for}

  { Poiščimo vse premice. }
  for i := 1 to n do Premice[i] := nil;
  for i := 1 to n − 1 do for j := i + 1 to n do begin
    New(p); p↑.n := 0; p↑.pt := 0; p↑.np := nil; p↑.pp := nil;
    { Poglejmo, katere točke pokriva premica skozi t[i] in t[j]. }
    for k := 1 to n do if NaPremici(i, j, k) then Dodaj(k, p);
    { Dodajmo premico v seznam premic, ki pokrivajo p↑.n točk. }
    if Premice[p↑.n] <> nil then Premice[p↑.n]↑.pp := p;
    p↑.np := Premice[p↑.n]; Premice[p↑.n] := p;
  end; {for}

  if n = 1 then begin { Imamo eno samo točko; pokrili jo bomo z eno premico. }
    New(p); p↑.n := 0; p↑.pt := nil; p↑.np := nil; p↑.pp := nil;
    Dodaj(1, p); Premice[p↑.n] := p;
```

```
  end; {if}

  { Požrešno izbirajmo premice. }
  pn := n; StPremic := 0;
  while pn > 0 do begin
    while Premice[pn] <> nil do begin
      { p je ena od premic, ki pokrivajo največ točk. }
      p := Premice[pn]; Write('Premica skozi');
      while p↑.pt <> nil do begin { Pobrišimo točke, ki jih p pokriva. }
        Write(' ', p↑.pt↑.it);
        Brisi(p↑.pt↑.it);
      end; {while}
      Dispose(p); WriteLn; StPremic := StPremic + 1;
    end; {while}
    { Premic, ki bi pokrivale pn točk, ni več; posvetimo se tistim, }
    pn := pn - 1;            { ki pokrivajo pn - 1 ali manj točk. }
  end; {while}

  WriteLn('Število uporabljenih premic: ', StPremic);
end. {Pokrivanje}
```

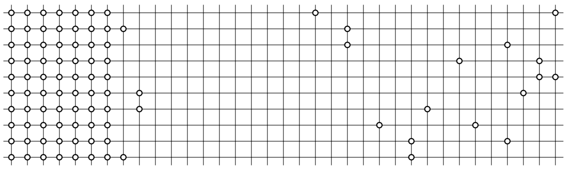
Nerodno pri tem postopku je, da hrani za vsako premico seznam vseh točk, ki jih ta premica pokriva. Ti seznami bi utegnili vsi skupaj požreti $O(n^3)$ prostora.[6] Pomnilniške zahteve bi lahko zmanjšali, če bi za vsako premico hranili samo podatek o tem, iz katerih dveh točk je bila pridobljena in koliko točk trenutno pokriva. Ko si požrešni algoritem izbere neko premico in hoče pobrisati vse točke, ki jih ta pokriva, zdaj ne bi imel pri roki seznama teh točk, pač pa bi moral iti po vseh točkah in za vsako posebej preveriti, če jo izbrana premica pokriva ali ne. Podprogram Brisi pa tudi za dano točko ne bi imel na voljo seznama premic, ki pokrivajo to točko, in bi moral iti po vseh premicah ter pri vsaki pogledati, če to točko pokriva ali ne. Za brisanje točke lahko torej zdaj porabimo $O(n^2)$ časa, tako da je časovna zahtevnost celega algoritma $O(n^3)$, torej nič slabša kot pri gornjem programu. Prostorska zahtevnost pa je le $O(n^2)$, saj hranimo za vsako premico le konstantno mnogo podatkov.

Za konec si oglejmo še nekaj primerov, pri katerih požrešni algoritem ne vrne najboljše možne rešitve. Zaradi preprostosti se bomo omejili na probleme, v katerih imajo vse točke celoštevilske koordinate. Pri vsaki sliki je opisano, kako lahko take primere sestavimo in koliko je pri njih požrešna rešitev slabša od najboljše možne (kolikokrat več premic porabi, da pokrije vse točke). Videli bomo primere, pri katerih lahko razmerje med številom premic pri požrešni in pri optimalni rešitvi doseže $\frac{1}{2}\ln n$, če je n število vseh točk. Izkaže se tudi, da to razmerje nikoli ne more preseči vrednosti $1 + \ln n$.

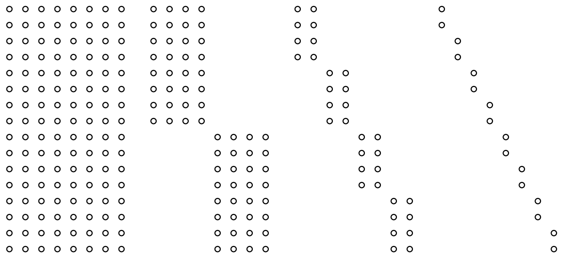
[6] To se pravzaprav lahko zgodi le zaradi numeričnih nenatančnosti. Če so recimo točke A, B in C približno na isti premici, bi se lahko zgodilo, da pri opazovanju premice skozi A in C vidimo, da na njej leži tudi B, ko pa bi gledali premico skozi A in B, bi dobili občutek, da C ne leži na njej. Zato bi lahko eno in isto premico šteli po večkrat in pri tem opazili različne podmnožice točk, ki jih ta premica zares pokriva. Če takih numeričnih nenatančnosti ne bi bilo, pa je seveda jasno, da bi lahko vsaka točka pripadala največ $n - 1$ premicam, le pri naštevanju teh premic bi morali paziti, da ne bi iste premice šteli po večkrat. Lahko bi na primer naredili takole: če bi pri opazovanju točke T_i videli, da leži na premici skozi T_i in T_j tudi neka T_k, za katero je $k < i$, bi to premico ignorirali, saj bi vedeli, da smo jo morali že opaziti, ko smo gledali premice skozi T_k. Kakorkoli že, čim vemo, da pripada vsaka točka največ $n - 1$ premicam, vemo tudi, da bomo imeli največ $n(n - 1)$ zapisov TTockaNaPremici, tako da bo prostorska zahtevnost našega postopka le $O(n^2)$.



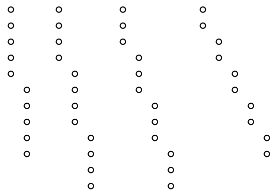
Imejmo m vrstic ($m > 3$) s po tremi točkami; ena naj bo vedno pri $x = 1$, drugi dve pa tako, da nobena poševna premica ne more pokriti več kot dveh točk naenkrat. Požrešni algoritem bi uporabil $m + 1$ premic namesto m premic.



Imejmo pravokotnik točk (m vrstic, $m - 3$ stolpcev). Dodajmo v vsako vrstico še dve točki in to tako, da nobena poševna premica ne bo mogla pokriti več kot dveh takih točk. Vse skupaj lahko pokrijemo že z m vodoravnimi premicami, požrešni algoritem pa uporabi najprej $m - 3$ navpičnih premic za pravokotnik točk na levi in nato še m premic za ostale točke; skupaj torej $2m - 3$ namesto le m premic, kar je skoraj dvakrat preveč.



Skupino $n = 2^m \cdot (2^m - 1)$ točk je mogoče razporediti v 2^m vrstic s po $2^m - 1$ točkami tako, da bi požrešni algoritem našel rešitev z $m \cdot 2^{m-1}$ premicami, čeprav se da vse točke pokriti že s samo 2^m premicami. Na tej sliki je primer za $m = 4$, le stolpce bi bilo treba še tako razmakniti, da se ne bi dalo s poševnimi premicami pokriti po več kot dveh točk naenkrat. Razmerje med požrešno in optimalno rešitvijo je približno $\frac{1}{4} \lg n \approx 0{,}36 \ln n$.



Izberimo si števili h in m (na sliki je primer za $h = 10$ in $m = 5$). Točke postavljajmo najprej v stolpce po m, dokler ne dosežemo (ali presežemo) višine h. Potem jih podobno postavljajmo v stolpce po $m - 1$, $m - 2$, …, 2. (Stolpce je treba še tako razmakniti, da poševne premice ne morejo pokriti več kot dveh točk na enkrat.) Požrešni algoritem uporablja navpične premice namesto vodoravnih in pri primernih h in m lahko porabi skoraj $(\frac{1}{2} \ln n)$-krat toliko premic kot optimalna rešitev (pri čemer je n število vseh točk); vendar pa se temu razmerju res približamo šele pri precej velikih n.

Naloga: str. 4

R1989.3.4

Preden se lotimo dela z izrazom, ga je koristno iz niza predelati v neko bolj standardno obliko, v kateri se nam ne bo treba več ukvarjati s presledki med operatorji in s tem, ali neka številka vsebuje decimalno piko ali ne. Predelali ga bomo v zaporedje „osnovnih simbolov“ ali „žetonov“ (*tokens*). To lahko storimo tako, da beremo vhodni niz znak za znakom; operatorji `+`, `-`, `*` in `/`, pa tudi oklepaji in zaklepaji, naj bodo že sami zase osnovni simboli, poleg tega pa imejmo še eno vrsto osnovnega simbola, ki bo predstavljal številske konstante (`Num` v spodnjem programu) in enega za konec izraza (`EolT` v spodnjem programu). Osnovni simbol bo predstavljen z zapisom `TokenT`, ki vsebuje tip simbola (`t` — zanj bi lahko uvedli poseben naštevni tip, še lažje pa bo, če uporabimo kar tip char), pri številskih konstantah pa tudi vrednost (polje x).

Pri branju izraza bo koristno, če bomo lahko naslednji znak le pogledali, ne pa ga tudi res prebrali (oz. če bomo lahko nek znak prebrali dvakrat). Lahko bi prebrali cel izraz v nek niz in se potem sprehajali po njem naprej in nazaj, čisto dobro pa je tudi, če v neki spremenljivki hranimo že prebrani znak, ki je bil vrnjen v zaporedje, da ga bomo prebrali še enkrat. Spodnji program to doseže s spremenljivko PutBack (ki ima vrednost Chr(0), če je „prazna") in podprogramom GetCh, ki uporabi vrednost iz PutBack, če pa je ta prazna, prebere naslednji znak iz datoteke.

Za branje naslednjega osnovnega simbola imamo podprogram NextToken. Ta najprej preskoči presledke, če jih je kaj; nato, če naleti na enoznakovne simbole (operatorje, oklepaje) ali konec niza, vrne to kot en simbol; številko pa pretvori v tip real in jo vrne kot osnovni simbol tipa Num.

Če bi takole predelali izraz v zaporedje osnovnih simbolov, bi se lahko lotili računanja čisto po zdravi pameti. Pri vrstnem redu računanja moramo upoštevati oklepaje in prioriteto operatorjev. Lahko bi se torej za začetek sprehodili po izrazu in šteli, koliko oklepajev je trenutno odprtih ter kje se začnejo. Ko bi naleteli na zaklepaj, bi del izraza od zadnjega oklepaja do tega zaklepaja izračunali z rekurzivnim klicem istega podprograma in nato v zaporedju osnovnih simbolov ves ta del izraza nadomestili z enim samim simbolom tipa Num, ki bi imel za vrednost kar vrednost tistega dela izraza. Ko bi se na ta način počasi znebili vseh oklepajev, bi lahko v mislih „razrezali" izraz pri vsakem operatorju + ali - (razen pri - na začetku zaporedja ali tik za simboloma * ali /, kajti tak - je unaren). Vsak od nastalih podizrazov je sestavljen iz števil, ki so ločena z operatorjema * in /, tik pred kakšnim številom pa je lahko tudi unarni minus. Take z minusom pomnožimo z -1 in se minusa znebimo. Vrednost podizraza lahko zdaj izračunamo tako, da začnemo s prvim številom in ga nato z vsakim naslednjim številom pomnožimo ali pa delimo, odvisno od operatorja pred tem številom. Ko imamo vrednost vsakega podizraza, izračunamo vrednost celega izraza spet tako, da začnemo z vrednostjo prvega in ji nato prištevamo ali odštevamo vrednosti naslednjih, spet odvisno od tega, ali je pred nekim podizrazom + ali -.

Lahko pa smo še za odtenek bolj elegantni in vse skupaj združimo v en sam prehod skozi vhodne podatke. V ta namen imamo spodaj podprograme EvalExpr (za izračun celega izraza), EvalTerm (za izračun podizraza, dobljenega z množenjem in deljenjem) in EvalAtom (za izračun izraza v oklepajih, izraza z unarnim minusom ali pa izraza, ki je kar številska konstanta). Vsi pričakujejo kot parameter prvi osnovni simbol svojega dela izraza, ob koncu izvajanja pa vrnejo v njem prvi osnovni simbol za svojim delom izraza, kar bo potreboval klicatelj za nadaljevanje dela. Vrednosti izrazov računajo sproti. Na primer, EvalExpr pokliče EvalTerm, da bi dobil vrednost prvega podizraza. Če za tem podizrazom pride kaj drugega kot + ali -, lahko kar končamo (če je bil cel izraz pravilno zgrajen, lahko tam razen + ali - tako ali tako nastopa le še zaklepaj ali pa konec izraza), sicer pa preberemo naslednji izraz in njegovo vrednost prištejemo ali odštejemo ter tako nadaljujemo. Podobno dela EvalTerm, ki za vsak faktor v svojem podizrazu kliče EvalAtom.

```
function Eval(var f: text): real;

  const EOL = Chr(13);
```

```
var PutBack: char;

function GetCh: char;
begin
  if PutBack = Chr(0) then begin
    if Eoln(f) then begin PutBack := EOL; ReadLn(f) end
    else Read(f, PutBack);
  end; {if}
  GetCh := PutBack; PutBack := Chr(0);
end; {GetCh}

const EolT = 'e'; Num = 'n';
type TokenT = record t: char; x: real end;

procedure NextToken(var t: TokenT);
var c: char; u: real;
begin
  { Preskočimo presledke. }
  repeat c := GetCh until not (c in [' ', Chr(9)]);
  if c = EOL then { Prišli smo do konca izraza. }
    begin t.t := EolT; PutBack := c end
  else if c in ['+', '-', '/', '*', '(', ')'] then
    t.t := c { Operator ali oklepaj, ki je že sam zase osnovni simbol. }
  else if c in ['0'..'9', '.'] then begin
    { Prebrati moramo število. }
    t.t := Num; t.x := 0;
    while c in ['0'..'9'] do { Števke pred decimalno piko. }
      begin t.x := t.x * 10 + (Ord(c) - Ord('0')); c := GetCh end;
    if c = '.' then begin
      c := GetCh; u := 1; { Decimalna pika in mogoče še števke za njo. }
      while c in ['0'..'9'] do begin
        u := u * 0.1; t.x := t.x + u * (Ord(c) - Ord('0')); c := GetCh;
      end; {while}
    end; {if}
    PutBack := c; { Ta znak že ne spada več k našemu številu. }
  end else
    begin WriteLn('Nedovoljen znak: #', Ord(c)); t.t := EolT end;
end; {NextToken}

function EvalExpr(var t: TokenT): real; forward;

function EvalAtom(var t: TokenT): real;
begin
  if t.t = '(' then begin
    NextToken(t); EvalAtom := EvalExpr(t);
    if t.t <> ')' then WriteLn('Pričakoval bi zaklepaj, ne pa ', t.t, '.');
    NextToken(t);
  end else if t.t = Num then begin
    EvalAtom := t.x; NextToken(t);
  end else if t.t = '-' then begin
    NextToken(t); EvalAtom := -EvalAtom(t);
  end else WriteLn('Pričakoval bi oklepaj, unarni minus ',
                   'ali številko, ne pa ', t.t, '.');
end; {EvalAtom}

function EvalTerm(var t: TokenT): real;
```

```
  var x, y: real; op: char;
  begin
    x := EvalAtom(t);
    while t.t in ['*', '/'] do begin
      op := t.t; NextToken(t); y := EvalAtom(t);
      if op = '*' then x := x * y else x := x / y;
    end; {while}
    if not (t.t in ['+', '-', ')', EolT]) then WriteLn('Pričakoval bi ',
      'zaklepaj, + , - ali konec izraza, ne pa ', t.t, '.');
    EvalTerm := x;
  end; {EvalTerm}

  function EvalExpr(var t: TokenT): real;
  var x, y: real; op: char;
  begin
    x := EvalTerm(t);
    while t.t in ['+', '-'] do begin
      op := t.t; NextToken(t); y := EvalTerm(t);
      if op = '+' then x := x + y else x := x − y;
    end; {while}
    if not (t.t in [')', EolT]) then
      WriteLn('Pričakoval bi zaklepaj ali konec izraza, ne pa ', t.t, '.');
    EvalExpr := x;
  end; {EvalTerm}

var t: TokenT; x: real;
begin {Eval}
  PutBack := Chr(0); NextToken(t);
  if t.t = EolT then begin WriteLn('Datoteka je prazna!'); Eval := 0 end
  else begin
    x := EvalExpr(t); Eval := x;
    if t.t <> EolT then WriteLn('Izraz ni pravilne oblike! ',
                                'Zatakne se pri ', t.t, '.');
    WriteLn('Vrednost izraza: ', x:10:5);
  end; {if}
end; {Eval}
```

Za tip osnovnega simbola, ki predstavlja konec izraza, bi lahko namesto EolT uporabili tudi kar EOL, a potem bi bila sporočila o napakah, če nepričakovano zgodaj naletimo na konec izraza, videti bolj čudno (za „`ne pa`“ ne bi bilo videti ničesar, pač pa bi se pika izpisala na začetku vrstice).