18. državno tekmovanje v znanju računalništva (1994)

NALOGE ZA PRVO SKUPINO

1994.1.1 Recimo, da je naš računalnik tak, kot so jih imeli pred mnogimi leti, in pozna samo števke od 0 do 9. Z njimi lahko seveda zapišemo tudi večja števila, vendar pri črkah že odpove.

Rešitev: str. 6

Naloga zahteva, da naš računalnik naučimo zapisovati tudi črke. Ker črk ne pozna, si bomo pomagali s trikom in črke zapisali s številkami. **Opiši postopek**, kako bi zapisoval črke s pomočjo števk in to tako, da bi porabil za neko besedilo kar najmanj števk.

1994.1.2 **Napiši podprogram** Pretvori, ki dobi kot parameter tabelo znakov in njeno dolžino in ju predela tako, da spremeni ubežna zaporedja v znake. Ubežno zaporedje se začne z znakom `'\'`, ki mu sledijo do 3 števke (znaki od `'0'` do `'9'`). Števke sestavljajo kodo znaka, s katerim moramo nadomestiti ubežno zaporedje. Če znaku `'\'` ne sledi števka, ga nadomestimo z znakom s kodo 0.

Rešitev: str. 7

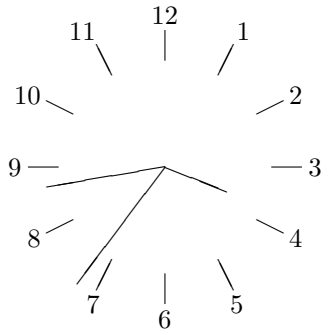
Primeri: `'abc\100efg'` → `'abcdefg'`; `'\1000'` → `'d0'`; `'A\66C'` → `'ABC'`.

Uporabljaj naslednje deklaracije:

```
const MaksDolzinaNiza = ...; { največja dolžina niza }
type NizT = array [1..MaksDolzinaNiza] of char;
procedure Pretvori(var Niz: NizT; var DolzinaNiza: integer);
```

1994.1.3 **Napiši program**, ki riše uro s kazalci, ki teče. Ura naj bi izgledala podobno kot televizijska ura. Na uri morajo biti sekundni, minutni in urni kazalec. Dimenzije zaslona so: $X = 1..1000$, $Y = 1..1000$. Na začetku je zaslon prazen. Na razpolago imaš naslednje podprograme:

Rešitev: str. 8



- **procedure** Cas(**var** Ura, Minuta, Sekunda: integer);
  Vrne trenutni čas.
- **procedure** PojdiNa(X, Y: integer);
  Pero na zaslonu postavi na točko (X, Y).
- **procedure** Naprej(K: real; D: integer);
  Pero premakne pod kotom K za dolžino D naprej. Način risanja pove, ali pero pri tem riše ali briše.

- **procedure** Risi; **procedure** Brisi;
  Način risanja peresa postavita na risanje oz. brisanje.

Rešitev: str. 10

1994.1.4

Sestavi funkcijski podprogram PodNiz(t, s), ki vrne *resnično* (true) natanko takrat, ko lahko dobimo besedo t iz besede s, če v besedi s prečrtamo nekaj (nič ali več) znakov. Lahko predpostavimo, da se beseda v tabeli konča s presledkom. Primera:

PodNiz('banana ', 'ljubljančanka ') = true
PodNiz('pero ', 'koper ') = false

Uporabljaj naslednje deklaracije:

```
const MaksDolzinaNiza = ...; { največja dolžina niza }
type NizT = array [1..MaksDolzinaNiza] of char;
function PodNiz(t, s: NizT): boolean;
```

NALOGE ZA DRUGO SKUPINO

Rešitev: str. 10

1994.2.1

Napiši funkcijski podprogram, ki za dano besedo vrne število zlogov v njej. Besedo vedno sestavlja vsaj en zlog. Vsak samoglasnik določa svoj zlog. Črko r, ki jo obdajata dva soglasnika, štejemo kot samoglasnik.[1] Primeri:

Stej('matematika ') = 5; Stej('z ') = 1; Stej('prstan ') = 2.

Uporabljaj naslednje deklaracije:

```
const MaksDolzinaNiza = ...; { največja dolžina niza }
type NizT = array [1..MaksDolzinaNiza] of char;
function Stej(Niz: NizT): integer;
```

Predpostaviš lahko, da je v tabeli Niz takoj za koncem besede vsaj en presledek (znak ' ').

Rešitev: str. 11

1994.2.2

Imamo števili $n > 0$ in $k \geq 0$. Sestavimo tabelo z n elementi, ki imajo vrednosti $-k, \ldots, k$. Če je vsota elementov tabele 0, imenujemo ta vektor *n-k-sestavljanka*.

Opiši postopek, ki prebere števili n in k ter izpiše vse n-k-sestavljanke. Postopek naj deluje čim hitreje. Primer vseh 3-2-sestavljank:

$(-2,0,2)$; $(-2,1,1)$; $(-2,2,0)$; $(-1,-1,2)$; $(-1,0,1)$;
$(-1,1,0)$; $(-1,2,-1)$; $(0,-2,2)$; $(0,-1,1)$; $(0,0,0)$;
$(0,1,-1)$; $(0,2,-2)$; $(1,-2,1)$; $(1,-1,0)$; $(1,0,-1)$;
$(1,1,-2)$; $(2,-2,0)$; $(2,-1,-1)$; $(2,0,-2)$.

[1] Malo bolj problematični so primeri, ko se beseda začne na r, temu pa sledi soglasnik. Na primer, *rjast* ima dva zloga in ne enega, *rjavolas* pa verjetno tri in ne štiri. Tvoj podprogram naj take r-je razglasi ali za samoglasnike ali pa za soglasnike, kakor se ti zdi pač bolj prikladno.

1994.2.3

Rešitev: str. 12

Z računalnikom krmilimo uro v zvoniku. Mehanizem skrbi za usklajeno pomikanje obeh kazalcev, naloga računalnika pa je, da vsako minuto ukaže mehanizmu, naj premakne kazalce za eno minuto, ter da ob vsaki polni uri odbije uro (število udarcev zvona mora biti enako uri: točno ob enih en udarec, ob dveh dva, ..., opoldne in opolnoči 12 udarcev; udarci naj si sledijo v razmiku treh sekund).

Na voljo imaš naslednja podprograma:

- PremakniKazalce ob vsakem klicu premakne kazalce za eno minuto naprej;
- UdariNaZvon sproži en udarec mehanizma na zvon.

Operacijski sistem računalnika vsako sekundo pokliče podprogram VsakoSekundo, ki upravlja z uro. **Napiši ta podprogram.**

Zaradi enostavnosti predpostavimo, da se program požene opolnoči (prvi klic tvojega podprograma bo že kar takoj opolnoči in že takoj je treba odbiti uro strahov) in da ob startu programa kazalci kažejo polnoč. Tvoj podprogram lahko uporablja globalne (ali lastne statične) spremenljivke, ki jim lahko tudi predpišeš začetno vrednost.

1994.2.4

Rešitev: str. 13

Napiši program ali podroben algoritem, ki pobarva grafični zaslon s črnimi pikami v naključnem vrstnem redu. Na voljo imaš naslednje parametre in pomožne podprograme:

- Velikost zaslona je XMax × YMax.
- Podprogram Pobarvaj(x, y) pobarva točko s koordinatami (x, y) s črno barvo.
- Funkcija Pobarvana(x, y) vrne true, če je točka (x, y) že pobarvana s črno barvo, sicer pa false.
- Funkcija Random(n) vrne naključno število med vključno 0 in n − 1.

Program mora vsako točko pobarvati natanko enkrat, ko so vse točke pobarvane črno, pa naj se ustavi. Privzeti smeš, da na začetku na zaslonu ni črnih točk. Točke mora barvati **enakomerno** in naključno. Program naj bo hiter in naj ne porabi veliko dodatnega pomnilnika.[2] Upoštevaj, da je tipična velikost zaslona 1280×1024.

NALOGE ZA TRETJO SKUPINO

1994.3.1

Rešitev: str. 17

Zaradi enostavnosti se domenimo, da „datoteka dolžine N“ pomeni niz bitov dolžine N (datoteke pri tej nalogi torej nimajo imen).

a) Program P za kompresijo podatkov je tak program, ki vsaki datoteki D priredi neko drugo datoteko $P(D)$. Seveda programu za kompresijo pripada tudi program $\bar{P}$ za dekompresijo podatkov, ki kompresirani datoteki $P(D)$ priredi prvotno datoteko $D = \bar{P}(P(D))$. **Ugotovi**, kakšnim **minimalnim** zahtevam morata zadostovati programa P in $\bar{P}$ za pravilno delovanje, to je, za vsako datoteko D mora veljati $\bar{P}(P(D)) = D$.

[2] Radi bi na primer, da bi program porabil veliko manj pomnilniških celic, kot pa je točk na zaslonu. Z enakomernostjo barvanja pa hočemo reči, naj bodo točke, ki jih program do posameznega trenutka pobarva, približno enakomerno razpršene po zaslonu (delež pobarvanih točk naj torej ne bo npr. na enem koncu zaslona znatno večji kot na drugem).

b) S programi za kompresijo podatkov skušamo prihraniti prostor na disku. Naj bo P program za kompresijo podatkov. Njegovo učinkovitost na datotekah, krajših od N, ocenimo takole: naj bo $\mathcal{D}$ množica *vseh* datotek, katerih dolžina ne presega N. Skupna dolžina vseh datotek v $\mathcal{D}$ naj bo M. Vsako datoteko iz $\mathcal{D}$ skomprimiramo s programom P in dobimo novo množico komprimiranih datotek $\mathcal{E}$. Njihova skupna dolžina naj bo m. Razmerju $(M - m)/M$ pravimo *učinkovitost programa P na datotekah, krajših od N*. **Ugotovi**, kakšna je najboljša možna učinkovitost programov za kompresijo.

c) Upoštevaje rezultat iz točke (b), **opiši** kak program za kompresijo, ki ima najboljšo možno učinkovitost.

d) Ali je definicija učinkovitosti programov za kompresijo iz točke (b) uporabna v praksi? **Odgovor** utemelji. Predlagaj, kako bi v praksi izmerili učinkovitost programa za kompresijo.

Rešitev: str. 19

1994.3.2

Na pošti so prejeli napravo, ki za posamezno pošiljko določi njeno poštnino (skupna vrednost znamk, ki morajo biti na pošiljki).

Napiši algoritem, ki bo iz danih vrednosti znamk in poštnine izpisal niz vrednosti znamk (lahko tudi več enakih), ki jih je potrebno uporabiti.

Če ne obstaja niz znamk, ki točno pokrije poštnino, potem je rešitev niz, katerega skupna vrednost znamk preseže poštnino za najmanjši možni znesek. V primeru, da različni nizi znamk pokrivajo enako poštnino, je rešitev niz, ki vsebuje najmanj znamk. Primeri:

če so vrednosti znamk: 2, 7, 14, 17, 22, 63, 98,

so rešitve nekaterih poštnin takšne:

72 — 2, 7, 63; 86 — 2, 7, 14, 63; 143 — 17, 63, 63; 5 — 2, 2, 2.

Rešitev: str. 22

1994.3.3

Vhod v garažno hišo je opremljen z velikimi drsnimi vrati, ki so namenjena vstopu in izstopu avtomobilov in pešcev v garažno hišo. Za prehod pešcev je dovolj, da se vrata odprejo do polovice širine, za avtomobil pa je potrebno odpreti vrata v celotni širini.

Pri izstopu iz objekta pešec pritisne tipko za odpiranje vrat; prisotnost avtomobila, ki želi odpeljati iz garažne hiše, pa začuti tipalo v asfaltu (ki ni občutljivo na pešce).

Pri vstopu imamo en sam način aktiviranja odpiranja vrat — s kontaktno ključavnico. Pri tem ne moremo ločiti vstopajočega pešca od avtomobila, zato vrata vedno odpremo v celotni širini.

Tik ob vratih je varovalno optično tipalo, ki v primeru ovire med vrati zagotavlja, da ostanejo vrata odprta oziroma da se ponovno razprejo do prvotne širine (pešec/avto). Popolnoma zaprtih vrat ni mogoče odpreti z aktiviranjem tega tipala.

Vrata naj ostanejo odprta 15 sekund, potem naj se samodejno zaprejo. Vsaka prisotnost ovire v tem času ali nov ukaz za odpiranje nastavi preostali čas odprtosti ponovno na 15 sekund.

Napiši program, ki bo ob upoštevanju tipal krmilil motor za pomik vrat. Delovanje naj bo smiselno tudi v primeru tesno si sledečih vstopov/izstopov pešcev in avtomobilov.

Na voljo imaš naslednje prodprograme:

- OdprtostVrat je funkcija, ki vrne realno število med 0 in 1, ki pove lego vrat:
  0 — zaprta vrata;
  0,5 ali več — vrata so dovolj odprta za prehod pešca;
  1 — popolnoma odprta vrata, primerna za vstop/izstop avtomobila;

- IzstopPesec je funkcija, ki vrne true, dokler je pritisnjena tipka za odpiranje vrat z notranje strani, sicer vrne false;

- IzstopAvto je funkcija, ki vrne true, dokler tipalo na notranji strani vrat čuti prisotnost avtomobila, ki čaka na izvoz (sicer vrne false);

- Vstop je funkcija, ki vrne true, če je aktivirana kontaktna ključavnica za odpiranje vrat z zunanje strani, sicer vrne false;

- Ovira je funkcija, ki vrne true, če je med vrati prisotna ovira (avto, pešec ali kaj drugega), sicer vrne false;

- Motor(Ukaz) je podprogram, ki upravlja z motorjem:
  Motor(Stoj) — izklopi motor, vrata se ustavijo v trenutni legi;
  Motor(Odpiraj) — vklopi motor, vrata se odpirajo (ali ostajajo odprta);
  Motor(Zapiraj) — vklopi motor, vrata se zapirajo (ali ostajajo zaprta);

- NastaviBudilko(s) je podprogram, s katerim nastavimo odštevalno uro na s sekund (glej podprogram PreostaliCas);

- PreostaliCas je funkcija, ki vrne preostali čas (v sekundah) od začetnega, ki smo ga nastavili s podprogramom NastaviBudilko(s). Če se je čas že iztekel, vrača 0.

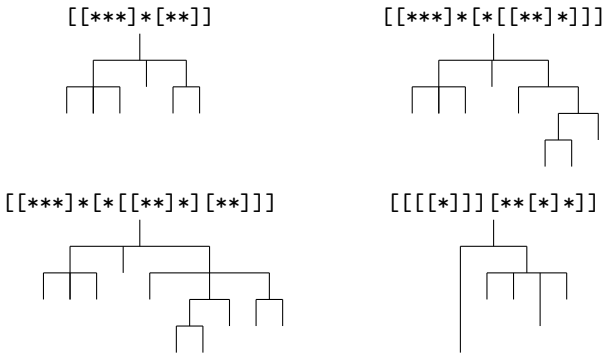
1994.3.4

Sestavi program, ki nariše drevo danega oklepajnega izraza tako, kot je prikazano v primerih. Oklepajni izraz je niz znakov `[`, `]` in `*`, pri katerem so oklepaji in zaklepaji pravilno gnezdeni. Predpostavi, da so vhodni nizi, s katerimi boš delal, pravilno oblikovani. Za risanje je na voljo podprogram Crta(x1, y1, x2, y2: real), ki nariše črto od točke (x1, y1) do točke (x2, y2).[3]

Rešitev: str. 23

Primeri:

[3] Mimogrede, še ena zanimiva naloga z risanjem dreves (v tekstovnem načinu) je "BUT we need a diagram" z ACMovega študentskega tekmovanja v programiranju za azijsko regijo (Tokio, 23. nov. 1998, problem F; #692 v zbirki na `online-judge.uva.es`).



REŠITVE NALOG ZA PRVO SKUPINO

Naloga: str. 1

R1994.1.1

Črke bomo desetiško kodirali (danes to delamo seveda binarno). Odločiti se moramo, ali bomo kodirali samo velike (25 črk) ali velike in male črke (50 črk). No, v obeh primerih je rešitev podobna.

Prva rešitev, ki nam pride na misel, je, da črke označimo s številkami od 1 do 25 (oziroma od 1 do 50). V tem primeru porabimo za n črk dolgo besedilo $2n$ števk. Temu bi lahko rekli dekadno kodiranje. Obstajajo pa še boljše rešitve in v teh je vsa umetnost kodiranja in hkrati tudi komprimiranja.

Uredimo vse črke po parih; tako dobimo $25 \times 25 = 625$ parov. Te lahko zapišemo s številkami od 1 do 625, to je s trimestnimi števili. V tem primeru porabimo za n črk dolgo besedilo $(n/2) \cdot 3 = 1{,}5\,n$ števk, kar je že lep prihranek v primerjavi s prvo rešitvijo.

Če uredimo črke po trojicah, dobimo $25 \times 25 \times 25 = 15625$ trojic. Le-te zapišemo s petmestnimi števili od 1 do 15625. Vendar, glej ga zlomka, za n črk dolgo besedilo bomo sedaj porabili $(n/3) \cdot 5 = 1{,}66\,n$ številk, kar pa je slabše kot v prvem primeru.

Če tako nadaljujemo, ugotovimo, da je črke še bolje urediti po peterkah, ki jih je 9 765 625 in jih torej lahko zapišemo s števili od 1 do 9 765 625. Za n črk dolgo besedilo bomo porabili $(n/5) \cdot 7 = 1{,}4\,n$ števk.

Ta razmislek lahko tudi posplošimo: če združujemo po k črk v skupine, je povprečna poraba prostora $f(k) = \lceil \log 25^k \rceil / k$ števk na vsako črko. Zgoraj smo ugotovili, da je $f(9) = 1{,}4$, vendar se dá pri večjih k najti še ugodnejša razmerja. Naslednjih nekaj k, pri katerih doseže $f(k)$ svojo najmanjšo vrednost doslej:

k	$f(k)$		približna vrednost
98	137/98	$\approx$	1,397959184
583	815/583	$\approx$	1,397941681
1068	1493/1068	$\approx$	1,397940075
14369	20087/14369	$\approx$	1,397940010
teoretična spodnja meja	$\log 25$	$\approx$	1,397940009

Iz definicije f in dejstva, da je $x \le \lceil x \rceil < x + 1$, sledi $\log 25 \le f(k) \le \log 25 + 1/k$; torej se lahko spodnji meji $\log 25$ približamo poljubno natančno, če smo le pripravljeni vzeti dovolj velik k. Ne moremo pa je čisto doseči, kajti to bi zahtevalo, da je neka potenca števila 25 hkrati tudi potenca števila 10, torej liha in soda obenem. Podobno lahko razmišljamo tudi, če moramo kodirati

poleg velikih tudi male črke (namesto 25 vzamemo 50). Vsekakor pa bi takšno kodiranje, če bi vzeli prevelik k, postalo nepraktično, saj bi zahtevalo preveč računanja z velikimi celimi števili.

Še ena možnost za varčevanje s prostorom pri kodiranju črk bi bila, da bi pogostejšim črkam (ali skupinam k črk) dodelili krajše kode (take z manj števkami), redkejšim pa daljše. To je koristno, če lahko (in običajno lahko) s krajšimi kodami pri pogostejših skupinah črk več pridobimo, kot pa bomo z daljšimi kodami pri redkejših skupinah črk izgubili. Paziti moramo le na to, da ne bo nobena koda podaljšek kakšne druge, saj bi bilo drugače tako kodirano besedilo težko (ali pa sploh nemogoče) nedvoumno dekodirati. Znan primer postopka, ki poišče takšne kode, se imenuje *Huffmanovo kodiranje*. Slabost takega kodiranja je, da moramo poznati pogostosti posameznih črk (ali skupin črk), te pa so odvisne od tega, na kakšnem besedilu (ali, še bolje: skupini besedil) smo jih merili; zato kod, ki je dober za eno besedilo, mogoče ni tako dober za neko drugo, v katerem so pogostosti črk drugačne (npr. ker je v drugem jeziku).

R1994.1.2

Naloga: str. 1

S številcem i se sprehodimo po danem nizu, številec j pa nam kaže, koliko znakov se je že nabralo v predelani različici niza. (Ker se niz ob predelovanju lahko le krajša, nikoli pa se ne podaljšuje, lahko odlagamo znake kar v isti niz, saj vemo, da bomo s tem vedno pisali čez tiste dele niza, ki smo jih že prebrali in nas ne zanimajo več.) Ko pri sprehajanju po nizu naletimo na znak `'\'`, preberemo še naslednjih nekaj števk in izluščimo celo število, ki ga predstavljajo, ter dodamo pripadajoči znak v predelani niz.

```
program UbezniZnaki;
const
  MaksDolzinaNiza = 10; { največja dolžina niza }
  MaksStevk = 3;        { največje število števk za znakom '\' }
type
  NizT = array [1..MaksDolzinaNiza] of Char;
var
  Niz: NizT;            { niz znakov za pretvorbo }
  DolzinaNiza: integer; { dolžina niza }

procedure Pretvori(var Niz: NizT; var DolzinaNiza: integer);
var
  i, j, k: integer;     { števci }
  c: integer;           { koda znaka }
begin
  i := 1; j := 0;
  while i <= DolzinaNiza do begin
    j := j + 1;
    if Niz[i] = '\' then begin
      i := i + 1; c := 0; k := 1;
      while (i <= DolzinaNiza) and (k <= MaksStevk) and
            (Niz[i] in ['0'..'9']) do begin
        c := c * 10 + Ord(Niz[i]) - Ord('0');
        i := i + 1; k := k + 1;
      end; {while};
      Niz[j] := Chr(c);
    end else begin
      Niz[j] := Niz[i]; i := i + 1;
    end; {if}
```

```
  end; {while}
  DolzinaNiza := j;
end; {Pretvori}

procedure IzpisiNiz(Niz: NizT; DolzinaNiza: integer);
var i: integer;
begin
  for i := 1 to DolzinaNiza do Write(Niz[i]);
  WriteLn;
end; {IzpisiNiz}

begin {UbezniZnaki}
  Niz := 'abc\100efg'; DolzinaNiza := 10;
  IzpisiNiz(Niz, DolzinaNiza); { izpiše: 'abc\100efg' }
  Pretvori(Niz, DolzinaNiza);
  IzpisiNiz(Niz, DolzinaNiza); { izpiše: 'abcdefg' }
end. {UbezniZnaki}
```

Naloga: str. 1

R1994.1.3

Vsako sekundo narišemo uro na zaslon, počakamo do konca te sekunde in nato uro zbrišemo; potem jo lahko narišemo s kazalci v novem položaju. Na začetku lahko narišemo še črtice na vsakih 30 stopinj in ob njih še številke za ure; dobro je, da so dovolj daleč od središča ure, tako da jih ob risanju in brisanju kazalcev ne bomo poškodovali. Spodnji program vsebuje tudi implementacijo podprogramov za delo z grafiko, ki jih omenja naloga (za Turbo Pascal).

```
program UraSKazalci;
uses Dos, Graph, Crt;
var
  PeroX, PeroY: integer;
  Ura, Minuta, Sekunda, NovaUra, NovaMinuta, NovaSekunda: integer;

procedure Cas(var Ura, Minuta, Sekunda: integer);
var WUra, WMinuta, WSekunda, WStotinka: word;
begin
  GetTime(WUra, WMinuta, WSekunda, WStotinka);
  Ura := WUra; Minuta := WMinuta; Sekunda := WSekunda;
end; {Cas}

procedure OdpriGraficniNacin;
var GDrv, GMode: integer;
begin
  GDrv := Detect; InitGraph(GDrv, GMode, 'c:\bp\bgi');
  if GraphResult <> grOk then Halt(1);
  PeroX := 1; PeroY := 1; SetColor(White);
end; {OdpriGraficniNacin}

procedure ZapriGraficniNacin;
begin
  CloseGraph;
end; {ZapriGraficniNacin}

procedure PojdiNa(X, Y: integer);
begin
  PeroX := X; PeroY := Y;
```

```
end; {PojdiNa}

procedure Naprej(K: real; D: integer);
var NoviPeroX, NoviPeroY: integer;
begin
  NoviPeroX := PeroX + Round(D * Cos(Pi * (K - 90) / 180));
  NoviPeroY := PeroY + Round(D * Sin(Pi * (K - 90) / 180));
  Line(PeroX, PeroY, NoviPeroX, NoviPeroY);
  PeroX := NoviPeroX; PeroY := NoviPeroY;
end; {Naprej}

procedure Pisi(Stevilo: integer);
var Niz: string;
begin
  Str(Stevilo, Niz);
  OutTextXY(Trunc(Round(PeroX - 0.5 * TextWidth(Niz))),
            Trunc(Round(PeroY - 0.5 * TextHeight(Niz))), Niz);
end; {Pisi}

procedure Risi;
  begin SetColor(White) end;

procedure Brisi;
  begin SetColor(Black) end;

var
  Sirina, Visina, X0, Y0, R: integer;
  KazU, KazM, KazS: integer;                    { dolžine kazalcev }
begin {UraSKazalci}
  OdpriGraficniNacin;
  Sirina := GetMaxX + 1; Visina := GetMaxY + 1;  { velikost zaslona }
  X0 := Sirina div 2; Y0 := Visina div 2;        { središče zaslona (in ure) }
  if X0 > Y0 then R := Y0 else R := X0;
  R := Trunc(Round(0.95 * R));                   { to bo zunanji polmer ure }
  { Določimo velikost kazalcev. }
  KazU := 10 * R div 20; KazM := 12 * R div 20; KazS := 13 * R div 20;
  for Ura := 1 to 12 do begin                    { Narišimo številke in črtice ob njih. }
    PojdiNa(X0, Y0);
    Brisi; Naprej(Ura * 30, 14 * R div 20);
    Risi; Naprej(Ura * 30, 2 * R div 20);
    Brisi; Naprej(Ura * 30, R div 20);
    Risi; Pisi(Ura);
  end; {for}
  Cas(Ura, Minuta, Sekunda);
  while not KeyPressed do begin
    Risi;                                        { Narišimo kazalce v trenutnem položaju. }
    PojdiNa(X0, Y0); Naprej((Ura + Minuta / 60) * 30, KazU);
    PojdiNa(X0, Y0); Naprej(Minuta * 6, KazM);
    PojdiNa(X0, Y0); Naprej(Sekunda * 6, KazS);
    repeat                                       { Počakajmo do konca te sekunde. }
      Cas(NovaUra, NovaMinuta, NovaSekunda);
    until NovaSekunda <> Sekunda;
    Brisi;                                       { Pobrišimo kazalce v starem položaju. }
    PojdiNa(X0, Y0); Naprej((Ura + Minuta / 60) * 30, KazU);
```

```
    PojdiNa(X0, Y0); Naprej(Minuta * 6, KazM);
    PojdiNa(X0, Y0); Naprej(Sekunda * 6, KazS);
    Ura := NovaUra; Minuta := NovaMinuta; Sekunda := NovaSekunda;
  end; {while}
  ZapriGraficniNacin;
end. {UraSKazalci}
```

Naloga: str. 2

R1994.1.4

Zahteva, naj bo mogoče dobiti t iz s z brisanjem nekaj črk, je enakovredna zahtevi, da se morajo v s-ju pojavljati vse črke iz t-ja (in to v pravem vrstnem redu), vmes pa so lahko še kakšne druge črke (ki bi jih morali iz s-ja zbrisati, da bi dobili t). Torej se lahko z zanko (števec i v spodnjem programu) premikamo po nizu s in si zapomnimo (števec j), koliko prvih črk niza t smo že zagledali. Vsakič, ko zagledamo naslednjo črko niza t, povečamo vrednost j. Če pridemo do konca s-ja prej kot do konca t-ja, vemo, da t ni s-jev podniz, sicer pa je.

```
program Banana;

const MaksDolzinaNiza = 15; { največja dolžina niza }
type NizT = array [1..MaksDolzinaNiza] of char;

function PodNiz(t, s: NizT): boolean;
var i, j: integer;
begin
  i := 1; j := 1;
  while (s[i] <> ' ') and (t[j] <> ' ') do begin
    if s[i] = t[j] then j := j + 1;
    i := i + 1;
  end; {while};
  PodNiz := t[j] = ' ';
end; {PodNiz}

begin
  WriteLn(PodNiz('banana ', 'ljubljančanka '));
  WriteLn(PodNiz('pero ', 'koper '));
end. {Banana}
```

REŠITVE NALOG ZA DRUGO SKUPINO

Naloga: str. 2

R1994.2.1

Spodnji podprogram si poenostavi delo s predpostavko, da se beseda, ki nas zanima, začenja šele v drugi celici tabele s, v prvi celici pa je presledek (pravzaprav je dovolj že, če ni samoglasnik); predpostavi tudi, da je za besedo v tabeli vsaj en presledek. To nam poenostavi preverjanje, kakšni črki stojita ob r-ju, v primerih, ko je r na začetku ali na koncu besede. Uporabljeni pogoj, da ob r-ju ne sme biti samoglasnikov, če naj nastane nov zlog, pomeni, da bomo tudi r na začetku besede razglasili za samoglasnik, če takoj za njim stoji soglasnik.

```
program Zlogi;
const MaksDolzinaNiza = 15; { največja dolzina niza }
type NizT = array [1..MaksDolzinaNiza] of char;

function Samoglasnik(Crka: char): boolean;
```

```
begin
  Samoglasnik := Crka in ['a', 'e', 'i', 'o', 'u'];
end; {Samoglasnik}

function StZlogov(s: NizT): integer;
var i, z: integer;
begin
  i := 2; z := 0;
  while s[i] <> ' ' do begin
    if Samoglasnik(s[i]) then z := z + 1
    else if (s[i] = 'r') and
            not (Samoglasnik(s[i - 1]) or Samoglasnik(s[i + 1])) then z := z + 1;
    i := i + 1;
  end; {while}
  if z = 0 then StZlogov := 1 else StZlogov := z;
end; {Stej}

begin {Zlogi}
  WriteLn(StZlogov(' matematika '));
  WriteLn(StZlogov(' z '));
  WriteLn(StZlogov(' prstan '));
end. {Zlogi}
```

R1994.2.2

Naloga: str. 2

Recimo, da smo določili v tabeli že prvih $m-1$ števil in dobili vsoto s; če bi zdaj za naslednje število vzeli i, bi bila vsota $s+i$. Ostane nam še $n-m$ števil, ki morajo biti vsa iz množice $\{-k,\ldots,k\}$, torej bo vsota vseh n števil na koncu vsaj $s+i-(n-m)k$ in kvečjemu $s+i+(n-m)k$. Mi pa bi radi, da bi bila vsota na koncu lahko 0. Torej mora veljati

$$s+i-(n-m)k \le 0 \le s+i+(n-m)k,$$

iz česar sledi pogoj

$$-(n-m)k-s \le i \le (n-m)k-s.$$

Drugih vrednosti i torej nima smisla postavljati na m-to mesto, ker v nadaljevanju nikakor ne bi mogli dobiti vsote 0. S to neenakostjo si lahko spodnji program pomaga, da se izogne nepotrebnim rekurzivnim klicem.

```
program Sestavljanka;

const MaxN = 20;
var n, k, i: integer;
    Tabela: array [1..MaxN] of integer;

function Min(a, b: integer): integer;
  begin if a < b then Min := a else Min := b end;

function Max(a, b: integer): integer;
  begin if a > b then Max := a else Max := b end;

procedure Pregled(Globina, Vsota: integer);
var
  Meja1, Meja2, i: integer;
begin
```

```
  if Globina = n + 1 then begin
    Write('(');
    for i := 1 to n do begin
      Write(Tabela[i]);
      if i < n then Write(',');
    end; {for}
    WriteLn(')');
  end else begin
    Meja1 := Max((Globina - n) * k - Vsota, -k);
    Meja2 := Min((n - Globina) * k - Vsota, k);
    for i := Meja1 to Meja2 do begin
      Tabela[Globina] := i;
      Pregled(Globina + 1, Vsota + i);
    end; {for}
  end; {if}
end; {Pregled}

begin {Sestavljanka}
  ReadLn(n, k);
  for i := -k to k do begin
    Tabela[1] := i;
    Pregled(2, i);
  end; {for}
end. {Sestavljanka}
```

Kot kaže spodnja tabela, število n-k-sestavljank kar hitro narašča:

	$n=$	0	1	2	3	4	5	6	7	8	9	10
$k=1$		1	1	3	7	19	51	141	393	1 107	3 139	8 953
2		1	1	5	19	85	381	1 751	8 135	38 165	180 325	856 945
3		1	1	7	37	231	1 451	9 331	60 691	398 567	2 636 263	17 538 157

Če jih hočemo le prešteti, ni treba, da vse naštejemo, tako kot to počne gornji program. Lahko pridemo kar do rekurzivne formule za število sestavljank, le problem moramo malo posplošiti: naj bo $f(n,k,s)$ število n-k-sestavljank z vsoto s. Razmislek, ki smo ga uporabili tudi pri pisanju programa, nam pokaže, da je

$$f(n,k,s) = \sum_{t=\max\{-(n-1)k,\,s-k\}}^{\min\{(n-1)k,\,s+k\}} f(n-1,k,t)$$

za $n > 0$; trivialna primera sta $f(0,k,0) = 1$ in $f(n,k,s) = 0$ za $|s| > nk$. Z metodo rodovnih funkcij se lahko prepričamo, da je $f(n,k,s)$ ravno koeficient pri členu x^{s+nk} v polinomu $(1 + x + x^2 + \ldots + x^{2k})^n$.

Naloga: str. 3

R1994.2.3

Program si lahko v globalnih spremenljivkah zapomni, koliko manjka do naslednje polne ure ali pa do naslednjega udarca na zvon in kolikokrat je še treba pozvoniti.

```
var
  DoPomika: integer value 60;      { sekund do pomika kazalcev }
  DoPolneUre: integer value 0;     { sekund do polne ure }
  DoUdarca: integer value 0;       { sekund do udarca na zvon }
  Udarcev: integer value 0;        { potrebno odbiti še toliko udarcev }
  Ura: integer value 11;           { koliko je ura }
```

```
  procedure PremakniKazalce; external;
  procedure UdariNaZvon; external;

procedure VsakoSekundo;
begin
  if DoPomika > 0 then DoPomika := DoPomika − 1
  else begin PremakniKazalce; DoPomika := 59 end;
  if DoPolneUre > 0 then DoPolneUre := DoPolneUre − 1
  else begin
    if Ura < 12 then Ura := Ura + 1 else Ura := 1;
    Udarcev := Ura; DoPolneUre := 3600 − 1;
  end; {if}
  if DoUdarca > 0 then DoUdarca := DoUdarca − 1
  else if Udarcev > 0 then begin
    UdariNaZvon; Udarcev := Udarcev − 1; DoUdarca := 2;
  end; {if}
end; {VsakoSekundo}
```

R1994.2.4

Naloga: str. 3

Če bi bil zaslon majhen, recimo velikosti 80×25, bi lahko uporabili naslednji algoritem. Točke na zaslonu uredimo po vrsti tako, da uredimo vrstice od zgoraj navzdol in točke v vrstici od leve proti desni. Nato izvajamo naslednje korake:

(1) i := število točk na zaslonu;
(2) r := Random(i);
(3) Po vrsti prebiraj točke na zaslonu.
    Ko naletiš na r-to nepobarvano točko, jo pobarvaj.
(4) i := i − 1;
(5) Če je i > 0, skoči na korak (2), sicer končaj.

Za velik zaslon se ta algoritem ne obnese, ker traja korak (3) predolgo. Pomagamo si tako, da si zapomnimo, koliko točk je treba še pobarvati v vsaki posamezni vrstici. V koraku (3) lahko torej zelo hitro določimo vrstico, v kateri je r-ta točka.

```
const XMax = 1280; YMax = 1024;

function Random(n: integer): integer; external;
procedure Pobarvaj(x, y: integer); external;
function Pobarvana(x, y: integer): boolean; external;

procedure PobarvajZaslon1;
var i, r, x, y: integer;
    Vrstica: array [0..YMax − 1] of integer;
begin
  for i := 0 to YMax − 1 do Vrstica[i] := XMax;
  for i := XMax * YMax downto 1 do begin
    r := Random(i); y := 0;
    while r >= Vrstica[y] do
      begin r := r − Vrstica[y]; y := y + 1 end;
    x := 0; Vrstica[y] := Vrstica[y] − 1;
    while r >= 0 do begin
      if Pobarvana(x, y) then r := r − 1;
      x := x + 1;
    end; {while}
```

```
    Pobarvaj(x − 1, y);
  end; {for}
end; {PobarvajZaslon1}
```

Časovna zahtevnost barvanja zaslona $X \times Y$ s tem postopkom je $O(XY(X+Y))$ (četrtina točk, torej $O(XY)$ točk, leži na spodnji desni četrtini zaslona in pri njih porabi prva zanka **while** $O(Y)$, druga pa $O(X)$ časa).

Pravzaprav bi bilo koristneje, če ne bi bili tako sistematični. Zelo preprost postopek bi bil, da bi naključno izbirali točke, dokler ne bi naleteli na kakšno nepobarvano; to bi potem pobarvali in nadaljevali na enak način. Vendar pa nas zdaj lahko zaskrbi, da bo mogoče potrebnih veliko takšnih naključnih poskusov, preden bomo naleteli na naslednjo nepobarvano točko, sploh proti koncu izvajanja postopka, ko bo že večina točk pobarvanih. Recimo, da je na celem zaslonu $n = X \cdot Y$ točk in da smo jih pobarvali že k; potem je verjetnost, da je neka naključno izbrana točka nepobarvana, enaka $p = (1 - k/n)$. Izkaže se, da bo v povprečju potrebnih $1/p = n/(n-k)$ naključnih poskusov, preden bomo zadeli kakšno nepobarvano točko.[4] Če to seštejemo po vseh točkah, imamo $s = \sum_{k=0}^{n-1} n/(n-k) = n \sum_{k=1}^{n} 1/k = nH_n$ poskusov. Vsota $H_n = \sum_{k=1}^{n} 1/k$ se imenuje „n-to harmonično število“ in velja $H_n \approx \ln n + \gamma$, pri čemer je $\gamma \approx 0{,}577$. Preden pobarvamo vse točke, potrebujemo torej $s \approx n \ln n$ operacij.

Ta postopek porabi največ časa proti koncu, ko je že večina točk pobarvanih in je zato potrebnih veliko naključnih poskusov, preden najde naslednjo nepobarvano. Recimo, da bi se ustavili takrat, ko ostane še a nepobarvanih točk; potem lahko prečešemo ves zaslon (kar zahteva še dodatnih $O(n)$ operacij), poiščemo te nepobarvane točke, jih vpišemo v neko tabelo, nato pa to tabelo premešamo in tako v naključnem vrstnem redu pobarvamo še te preostale točke. Barvanje do trenutka, ko ostane še a nepobarvanih točk, pa zdaj zahteva v povprečju $\sum_{k=0}^{n-a+1} n/(n-k) = n \sum_{k=a+1}^{n} 1/k = n(H_n - H_a) \approx n(\ln n - \ln a) = n \ln(n/a)$ poskusov. Če na primer vzamemo $a = \sqrt{n}$, imamo $n \ln \sqrt{n} = \frac{1}{2} n \ln n$, torej pol manj poskusov kot prej, pri tem pa je poraba pomnilnika narasla le za $O(\sqrt{n})$, ker moramo pač na koncu hraniti tabelo a nepobarvanih točk. Spodnji podprogram vzame za a širino zaslona, kar je ponavadi še več kot $\sqrt{n}$, saj je zaslon ponavadi širši kot višji. Njegova časovna zahtevnost je torej $\Theta(XY \ln Y)$.

```
procedure PobarvajZaslon2;
var i, r, x, y: integer;
    Ostale: array [1..XMax] of integer;
begin
  for i := 1 to XMax * YMax − XMax do begin
    repeat
      x := Random(XMax); y := Random(YMax);
    until not Pobarvana(x, y);
    Pobarvaj(x, y);
  end; {for}

  { Ostalo je še XMax nepobarvanih točk. Poglejmo, katere so to. }
  i := 0; for y := 0 to YMax − 1 do for x := 0 to XMax − 1 do
```

[4] Verjetnost, da bo potrebnih točno i poskusov, je $(1-p)^{i-1}p$ (ker nam mora $i-1$ poskusov spodleteti, za kar je verjetnost vsakič $1-p$, i-ti poskus pa mora uspeti, kar se zgodi z verjetnostjo p). Pričakovano število potrebnih poskusov je zato $\sum_{i=1}^{\infty} i((1-p)^{i-1}p)$, kar se z nekaj telovadbe izkaže za enako $1/p$. Več o tem najdemo v učbenikih verjetnostnega računa pod „geometrijska porazdelitev“; ali pa v MathWorld *s. v.* “Geometric Distribution”.

```
      if not Pobarvana(x, y) then
        begin i := i + 1; Ostale[i] := y * XMax + x end;
    { Pobarvajmo te preostale točke v naključnem vrstnem redu. }
    for i := XMax downto 1 do begin
      r := Random(i) + 1;
      Pobarvaj(Ostale[r] mod XMax, Ostale[r] div XMax);
      Ostale[r] := Ostale[i];
    end; {for}
end; {PobarvajZaslon2}
```

Če smo pripravljeni žrtvovati več pomnilnika, lahko prihranimo še nekaj časa. Če je na zaslonu n točk, potrebujemo $\lceil \lg n \rceil$ bitov pomnilnika za indeks vsake točke. Za tabelo a nepobarvanih točk potrebujemo torej približno $a \lg n$ bitov pomnilnika. Če smo pripravljeni porabiti n bitov pomnilnika, torej po en bit za vsako točko (kar je sicer glede na besedilo naloge najbrž že preveč), si lahko privoščimo $a = n/\lg n$. Pri barvanju prvih $n - a$ točk imamo zato le še $n \ln(n/a) = n \ln \lg a$ klicev funkcije Pobarvana.

Lahko pa bi se pri barvanju zaslona zgledovali tudi po generatorjih naključnih števil. Preprost način za generiranje psevdonaključnih števil je formula $x_{i+1} = (ax_i + 1) \bmod m$, pri čemer pa moramo primerno izbrati konstanti a in m. S to formulo bomo dobivali števila od 0 do $m - 1$; če oštevilčimo točke na zaslonu z $0, \ldots, n - 1$, lahko pri vsakem x_i pogledamo, če je manjši od n, in če je, pobarvamo točko x_i. Za m lahko vzamemo kakšno potenco števila 2 (lahko bi vzeli kar $2^{\lceil \lg n \rceil}$, torej najmanjšo potenco števila 2, ki je $\geq n$, vendar je bilo pri naših poskusih videti barvanje zaslona še bolj naključno, če smo vzeli dvakrat ali štirikrat tolikšen m), za a pa je potem koristno vzeti kakšno število oblike $8k+5$. To med drugim zagotavlja, da bomo od generatorja dobili vsa števila med 0 in $m - 1$, še preden se bo kakšno od njih ponovilo. Koristno je tudi, če a ni niti preblizu 1 niti preblizu m (sicer bi na primer majhnemu številu x_i sledilo vedno tudi majhno število x_{i+1}). Ni nujno, da vsak a, ki ustreza tem zahtevam, daje že tudi res dober generator psevdonaključnih števil, vendar pri našem problemu barvanja zaslona to niti ni tako zelo pomembno.[5] Lepo pri tem razmisleku je, da imamo le $O(m)$ računskih operacij, saj moramo spremljati le eno periodo našega psevdonaključnega generatorja; in če je $m = 2^{\lceil \lg n \rceil}$, je $m < 2n$, tako da je časovna zahtevnost našega algoritma le $O(n)$ in ne več $O(n \ln n)$ kot pri prejšnjem postopku.

```
procedure PobarvajZaslon3;
var StPobarvanih, x, y, r, a, m: integer;
begin
  { m := najmanjša potenca števila 2, ki je ≥ XMax * YMax. }
  m := 1; while m < XMax * YMax do m := m * 2;
  { Zdi se, da je pri večjem m barvanje zaslona videti še malo bolj
    naključno. Brez naslednjega stavka ima program na začetku barvanja
    nekatere stolpce rajši kot nekatere druge. }
  m := m * 4;
  { a := oblike 8k + 5, niti preblizu 1 niti preblizu m. }
  a := ((m div 2) div 8) * 8 + 5;
```

[5] Dober uvod v generiranje psevdonaključnih števil je Knuth, *The Art of Computer Programming*, 2. knjiga, 3. poglavje, še posebej razdelek 3.6, od koder smo povzeli tudi zgoraj navedene napotke. Glej tudi W. H. Press *et al.*, *Numerical Recipes in C*, 2. izd., § 7.1.

```
  { Načeloma je vseeno, s katerim r začnemo, saj bo imelo naše zaporedje }
  r := Random(XMax * YMax);                                { periodo m. }
  StPobarvanih := 0;
  while StPobarvanih < XMax * YMax do begin
    if r < XMax * YMax then begin
      Pobarvaj(r mod XMax, r div XMax);
      StPobarvanih := StPobarvanih + 1;
    end; {if}
    r := (a * r + 1) mod m;
  end; {while}
end; {PobarvajZaslon3}
```

Pri računanju `(a * r + 1) mod m` moramo biti pazljivi: vrednost `a * r` utegne biti prevelika za 32-bitna števila, tako da moramo ali uporabiti 64-bitne spremenljivke (če jih naš prevajalnik podpira) ali pa pisati svoje podprograme za računanje z velikimi celimi števili. (Na primer: pri zaslonu 1280×1024 bi gornji program dobil $m = 2^{23}$; a je blizu $m/2$ in če je r velik, blizu m, bo $a \cdot r$ lahko blizu 2^{45}.)

Še preprostejši (in tudi hitrejši) pa je naslednji postopek. Mislimo si, da bi točke na zaslonu oštevilčili po vrsticah in nato pri vsaki vrstici od leve proti desni z vrednostmi $0, \ldots, n-1$. Zahteva, naj vse točke pobarvamo v naključnem vrstnem redu, je pravzaprav enakovredna zahtevi, naj ta števila $0, \ldots, n-1$ zdaj naključno premešamo. To pa lahko enostavno naredimo tako, da nekajkrat ponovimo naslednja dva koraka: (1) vsak stolpec ciklično zamaknemo za neko naključno število točk navzdol („ciklično“ pomeni, da vsako točko, ki pade spodaj čez rob zaslona, postavimo nazaj na vrh, na začetek stolpca); (2) vsako vrstico ciklično zamaknemo za neko naključno število točk v desno. Prvi korak nam točko (x, y) premakne na $(x, (y + z_1[x]) \bmod Y)$, drugi pa (x, y) premakne na $((x + z_2[y]) \bmod X, y)$, pri čemer sta $z_1[0..X-1]$ in $z_2[0..Y-1]$ tabeli, ki nam povesta, kolikšen je zamik pri posamezem stolpcu in vrstici. Pri naših poskusih je bilo videti barvanje zaslona že čisto naključno, če smo takšno zamikanje izvedli dvakrat.

```
procedure PobarvajZaslon4;
const StZamikov = 2;
var ZamikX: array[1..StZamikov, 0..XMax − 1] of integer;
    ZamikY: array[1..StZamikov, 0..YMax − 1] of integer;
    x, y, xx, yy, i: integer;
begin
  for i := 1 to StZamikov do begin
    { V vsaki od tabel ZamikX[i] in ZamikY[i] pripravimo
      nek primeren naključen zamik. }
    for x := 0 to XMax − 1 do ZamikX[i, x] := Random(YMax);
    for y := 0 to YMax − 1 do ZamikY[i, y] := Random(XMax);
  end; {for i}
  for x := 0 to XMax − 1 do for y := 0 to YMax − 1 do begin
    xx := x; yy := y;
    for i := 1 to StZamikov do begin
      { Ciklično zamaknemo stolpec xx za ZamikX[i, xx] vrstic navzdol. }
      yy := yy + ZamikX[i, xx]; if yy >= YMax then yy := yy − YMax;
      { Ciklično zamaknemo vrstico yy za ZamikY[i, yy] stolpcev v desno. }
      xx := xx + ZamikY[i, yy]; if xx >= XMax then xx := xx − XMax;
    end; {for i}
```

```
      Pobarvaj(xx, yy);
    end; {for x, y}
end; {PobarvajZaslon4}
```

Lepo pri tem postopku je, da ne vsebuje toliko dragih računskih operacij, kot sta množenje in deljenje 64-bitnih števil pri prejšnji rešitvi.

Na oko je barvanje zaslona pri tem postopku čisto naključno, nič slabše kot pri prejšnjih rešitvah; je pa v enem pogledu ta rešitev vendarle slabša od njih. Če je na našem zaslonu n točk, obstaja $n!$ možnih vrstnih redov barvanja točk. Pri podprogramih PobarvajZaslon1 in PobarvajZaslon2 se hitro vidi, da ima vsak od teh $n!$ vrstnih redov enako verjetnost, da bo uporabljen[6] (če je funkcija Random(r) res pošten generator naključnih števil, torej če res z enako verjetnostjo vrne vsako od števil $0, \ldots, r-1$). Pri podprogramu PobarvajZaslon4 pa mnogi vrstni redi barvanja sploh niso možni. Pri vsakem od X stolpcev imamo Y možnosti za zamik tega stolpca, pri vsaki od Y vrstic pa X možnosti za zamik te vrstice; vsega skupaj torej z eno izvedbo zamikanja vrstic in zamikanja stolpcev dosežemo $X^Y Y^X$ različnih vrstnih redov. Po k zamikanjih (zgoraj smo predlagali $k = 2$) lahko dosežemo največ $(X^Y Y^X)^k$ različnih vrstnih redov (v resnici mogoče še manj, ker lahko različna zaporedja zamikov pripeljejo do istega vrstnega reda barvanja točk). Če pišemo $X = cY$ in upoštevamo, da je gotovo $c \ll Y$, je $(X^Y Y^X)^k = (c^Y Y^Y Y^{cY})^k \ll (Y^Y Y^Y Y^{cY})^k = Y^{(c+2)kY}$. Vseh možnih vrstnih redov pa je $n!$ za $n = XY = cY^2$; znano je (Stirlingova formula), da je $n! \approx n^n e^{-n}\sqrt{2\pi n} = n^n n^{-n/\ln n}\sqrt{2\pi n}$, to pa je naprej enako $(cY^2)^{cY^2(1-1/(cY^2))}\sqrt{2\pi c}Y$, kar je (če upoštevamo $c \geq 1$, $\sqrt{2\pi c} \geq 1$ in dejstvo, da pri dovolj velikih Y gotovo velja $1/(cY^2) \leq 1/2$) naprej $\geq (Y^2)^{cY^2(1-1/2)}Y = Y^{1+cY^2}$. Preden bi imelo število dosegljivih vrstnih redov, ki je $\leq Y^{(c+2)kY}$, sploh kakšne možnosti, da bi se približalo številu vseh vrstnih redov, ki je $\geq Y^{1+cY^2}$, bi morali torej imeti dovolj velik k, da bi bilo $(c+2)kY = 1 + cY^2$, torej $k = (1+cY^2)/((c+2)Y) \approx \frac{c}{c+2}Y$. Pri tolikšnem k bi bila časovna zahtevnost našega postopka že $O(kXY) = O(XY^2)$, torej pravzaprav nič boljša kot pri PobarvajZaslon1, prostorska zahtevnost (za tabeli ZamikX in ZamikY) pa $O(k(X+Y)) = O(XY + Y^2)$, kar je sploh odločno preveč.

Opisane postopke smo primerjali pri barvanju „zaslona“ 1280×1024 točk. Da risanje po zaslonu ne bi preveč vplivalo na čas izvajanja, program v resnici ni ničesar prikazoval na zaslonu, pač pa sta podprograma Pobarvaj in Pobarvana simulirala zaslon z globalno spremenljivko — tabelo 1280×1024 bitov. Rezultate prikazuje tabela na str. 18.

REŠITVE NALOG ZA TRETJO SKUPINO

R1994.3.1

Naloga: str. 3

(a) Edina zahteva, ki ji mora zadoščati program P, je, da priredi dvema različnima datotekama različni komprimirani datoteki. Z drugimi besedami, program P mora biti injektivna preslikava na množici vseh datotek. Program za dekompresijo $\bar{P}$ je pač inverz programa P.

[6] Naloga je pravzaprav zahtevala, naj zaslon barvamo enakomerno in naključno; med temi $n!$ vrstnimi redi pa je tudi nekaj takih, ki ga ne barvajo enakomerno, ampak npr. sistematično od zgoraj navzdol, torej točke, ki so v določenem trenutku že pobarvane, nikakor niso enakomerno razpršene po celem zaslonu. To je na nek način pomanjkljivost prejšnjih rešitev, vendar lahko do takšnih neenakomernih razporedov pride tudi PobarvajZaslon4; v praksi je verjetnost takšnega neenakomernega razporeda pri vseh opisanih rešitvah zanemarljivo majhna.

	Postopek	Časovna zahtevnost	Prostorska zahtevnost	Čas izvajanja	Št. klicev Pobarvana (mio.)
	PobarvajZaslon1	$O(XY(X+Y))$	$O(Y)$	69 s	419,2
	PobarvajZaslon2				
	— brez tabele Ostale	$O(XY\ln(XY))$	$O(1)$	23,1 s	19,66 (19,22⋆)
	— tabela velikosti Y	$O(XY\ln Y)$	$O(Y)$	10,8 s	10,35 (10,39⋆)
	— tabela $XY/\ln(XY)$	$O(XY\ln\ln(XY))$	$O(XY/\ln(XY))$	4,3 s	4,78 (4,78⋆)
	PobarvajZaslon3				
(†)	— $m = 2^{\lceil \lg(XY)\rceil}$	$O(XY)$	$O(1)$	6,5 s	0
(†)	— $m = 2\cdot 2^{\lceil \lg(XY)\rceil}$	$O(XY)$	$O(1)$	9,4 s	0
	— $m = 4\cdot 2^{\lceil \lg(XY)\rceil}$	$O(XY)$	$O(1)$	15,0 s	0
	PobarvajZaslon4				
(‡)	— StZamikov = 1	$O(XY)$	$O(X+Y)$	0,18 s	0
	— StZamikov = 2	$O(XY)$	$O(X+Y)$	0,26 s	0
	— StZamikov = 3	$O(XY)$	$O(X+Y)$	0,34 s	0
(‡)	sistematično barvanje zaslona (od zgoraj navzdol, od leve proti desni)				
		$O(XY)$	$O(1)$	0,10 s	0
	naivna rešitev: v tabeli pripravimo naključno permutacijo števil $0..(XY-1)$ (in torej porabimo preveč pomnilnika)				
		$O(XY)$	$O(XY)$	1,14 s	0

⋆ V oklepajih je pričakovano število klicev podprograma Pobarvana, torej $XY\ln(XY/a)$, če je a velikost tabele Ostale; če tabele nimamo, si mislimo $a = 1$.
† Barvanje ni videti dovolj naključno. ‡ Barvanje ni niti najmanj naključno.

Primerjava različnih rešitev naloge 1994.2.4 za $X = 1280$, $Y = 1024$.

Njegovo delovanje je nedefinirano, če ga poženemo nad datoteko, ki ni nastala s kompresijo kakšne datoteke s programom P.

(b) Naj bo k število vseh datotek, krajših od N.[7] Ker program za kompresijo P priredi različnim datotekam različne komprimirane datoteke, vsebuje množica $\mathcal{E}$ natančno k datotek. Torej je skupna dolžina m vseh datotek iz $\mathcal{E}$ najmanjša tedaj, ko množica $\mathcal{E}$ vsebuje prvih k najkrajših datotek. To pa so ravno vse datoteke iz $\mathcal{D}$. Tako smo ugotovili, da vedno velja $m \geq M$, torej je učinkovitost programa vedno manjša ali enaka 0.

(c) Najboljša učinkovitost programa ni večja od 0. Program za kompresijo P z najboljšo možno učinkovitostjo 0 je zato preprosto program, ki z datoteko ničesar ne naredi: $P(D) = D$. Pripadajoči program za dekompresijo $\bar{P}$ prav tako ničesar ne naredi: $\bar{P}(E) = E$.

(d) Seveda ima takole teoretično definirana učinkovitost programov za kompresijo v praksi majhno uporabno vrednost. V resnici nas ne zanima, kako dobro komprimira program vse možne datoteke; zanima nas učinkovitost kompresije za datoteke, s katerimi dejansko imamo opravka v praksi. Te datoteke pa so le majhen delček vseh možnih datotek. Pri datotekah iz vsakdanjega življenja se kažejo določene pravilnosti, ki jih programi za kompresijo uspešno izkoriščajo.

V praksi je pomembno, kakšne datoteke želimo komprimirati. Na primer, na datotekah, ki vsebujejo besedila, se bolje obnesejo eni programi za kompresijo, na datotekah, ki vsebujejo slike, pa drugi (tam se pogosto tudi odpovemo zahtevi po injektivnosti iz točke (a): ne moti nas, če se slika po kompresiji in dekompresiji malo spremeni, zato se lahko več podobnih slik skomprimira v enako zaporedje bitov). Zato je smiselno meriti učinkovitost programa za kompresijo

[7] Hitro se vidi, da je število dvojiških nizov, krajših od N, ravno $2^N - 1$ (če štejemo tudi prazni niz), njihova skupna dolžina pa je $M = 2 + (N-2)\cdot 2^N$, vendar to za nadaljevanje našega razmisleka pravzaprav ni pomembno.

glede na dani tip datotek (npr. besedila, programi ali bitne slike). Učinkovitost programa izmerimo statistično na zadosti velikem vzorcu datotek danega tipa. Tako dobimo dosti uporabnejšo oceno o tem, koliko prostora bomo dejansko prihranili s kompresijo.

R1994.3.2

Naloga: str. 4

Ker nas zanima med ugodnimi poštninami (takimi, ki čim manj presegajo zahtevno vrednost) taka z najmanj znamkami, bomo poštnine pregledovali po naraščajočem številu znamk. Na začetku vemo, da lahko z 0 znamkami sestavimo poštnino 0. Nato v vsakem koraku poskusimo vsako od poštnin, ki se jih je dalo sestaviti s k znamkami, dopolniti s še eno znamko; tako dobimo vse poštnine, ki se jih da sestaviti s $k+1$ znamkami. Na začetku bomo tako iz poštnine 0 dobili vse možne poštnine, ki se jih da dobiti z eno samo znamko; nato bomo iz teh dobili vse poštnine, ki se jih da dobiti z dvema znamkama; itd. Takemu preiskovanju pravimo *iskanje v širino*. Da ne bi po nepotrebnem izgubljali časa z neobetavnimi poštninami, si zapomnimo, za koliko presega zahtevano vrednost najboljša doslej najdena rešitev. Če kasneje odkrijemo kakšno poštnino, ki zahtevano vrednost presega za več kot toliko, jo lahko kar takoj pozabimo, saj vemo, da to zanesljivo ne bo najboljša rešitev. Na začetku vzamemo za ta presežek kar vrednost najdražje znamke, saj gotovo obstaja kakšna poštnina, ki presega zahtevano vrednost kvečjemu za toliko (sicer bi lahko iz nje vzeli kakšno znamko in dobili še boljšo rešitev). Še en pogoj, s katerim lahko zavržemo nekatere neobetavne rešitve, je naslednji: če smo neko vrednost poštnine uspeli sestaviti že nekoč prej z manj znamkami kot zdaj, nima smisla razvijati naprej trenutne poštnine z več znamkami, saj se bo dalo vse, kar lahko naredimo iz te poštnine, narediti tudi iz tiste prejšnje z enako vrednostjo in še manj znamkami. Strategija, ki izloča delne rešitve na opisani način, se imenuje *dinamično programiranje*.

```
program Znamke;

const NajvecjiNiz = 100;

type
  NizT = array [1..NajvecjiNiz] of integer;
  { PostnineT[1] je v bistvu seznam delnih poštnin.
    PostnineT[2, i] je vrednost zadnje znamke, ki smo jo dodali
    pri tvorjenju poštnine, da smo dobili poštnino PostnineT[1, i]. }
  PostnineT = array [1..2] of NizT;
  MnozicaT = set of 1..NajvecjiNiz;

var
  VrednostiZnamk : NizT;       { vrednosti znamk, iz katerih sestavljamo poštnino }
  NizZnamk: NizT;              { niz vrednosti znamk, ki tvorijo poštnino }
  Postnine: PostnineT;         { delne poštnine in zadnje dodane znamke }
  Postnina: integer;           { poštnina, ki jo želimo pokriti }
  Presezek: integer;           { najmanjši dosedanji presežek preko poštnine }
  OstanekPostnine: integer;    { poštnina brez zadnje znamke }
  PokriteVrednosti: MnozicaT; { poštnine, ki smo jih pokrili }

procedure BeriNizVrednosti(var Niz: NizT);
var i: integer;
begin
  i := 1;
```

```
  while not Eoln do
    begin Read(Niz[i]); i := i + 1 end;
  ReadLn;
  Niz[i] := −1;
end; {BeriNizVrednosti}

procedure Inicializacija;
var i : integer;
begin
  { Izračunajmo največji možni presežek: to je kar vrednost najdražje znamke,
    kajti gotovo se da sestaviti poštnino, ki želeno vrednost presega kvečjemu
    za vrednost ene znamke. }
  Presezek := VrednostiZnamk[1];
  i := 2;
  while VrednostiZnamk[i] <> −1 do begin
    if Presezek > VrednostiZnamk[i] then
      Presezek := VrednostiZnamk[i];
    i := i + 1;
  end; {while}
  { Na začetku ni pokrita še nobena poštnina. }
  PokriteVrednosti := [];
  Postnine[1, 1] := 0; Postnine[1, 2] := −1;
  NizZnamk[1] := −1;
end; {Inicializacija}

procedure NajdiNizZnamk(TrenPostnine: PostnineT; PokriteVrednosti: MnozicaT;
            var OstanekPostnine: integer; var NizZnamk: NizT; IndeksZnamke: integer);
var NovePostnine: PostnineT;
    TrenPostnina: integer;
    i, j, k: integer;
begin
  if (TrenPostnine[1, 1] <> −1) and (Presezek > 0) then begin
    i := 1; k := 1;
    while TrenPostnine[1, i] <> −1 do begin
      { Za vsako trenutno poštnino naredimo vse možne „naslednice“,
        ki jih lahko dobimo, če trenutni poštnini dodamo novo znamko. }
      j := 1;
      while VrednostiZnamk[j] <> −1 do begin
        { Izračunamo naslednico trenutne poštnine. }
        TrenPostnina := TrenPostnine[1, i] + VrednostiZnamk[j];
        if not (TrenPostnina in PokriteVrednosti) then begin
          { Trenutna poštnina še ni pokrita. }
          PokriteVrednosti := PokriteVrednosti + [TrenPostnina];
          if TrenPostnina < Postnina + Presezek then begin
            { Trenutna poštnina ne presega trenutno najboljše rešitve. }
            NovePostnine[1, k] := TrenPostnina;
            NovePostnine[2, k] := VrednostiZnamk[j];
            k := k + 1;
            if TrenPostnina >= Postnina then
              { Trenutna poštnina je izboljšala presežek. }
              Presezek := TrenPostnina − Postnina;
          end; {if}
        end; {if}
        j := j + 1;
```

```
        end; {while}
        i := i + 1;
      end; {while}
      NovePostnine[1, k] := −1;
      { Rekurzivni klic podprograma z novimi poštninami. }
      NajdiNizZnamk(NovePostnine, PokriteVrednosti, OstanekPostnine,
                    NizZnamk, IndeksZnamke + 1);
      if NovePostnine[1, 1] <> −1 then begin
        { Katera znamka, dodana v tem koraku, vodi do najboljše poštnine? }
        i := 1;
        while NovePostnine[1, i] <> OstanekPostnine do i := i + 1;
        NizZnamk[IndeksZnamke] := NovePostnine[2, i];
        OstanekPostnine := OstanekPostnine − NovePostnine[2, i];
      end else begin
        { Smo v najbolj zunanjem rekurzivnem klicu, torej smo našteli že }
        NizZnamk[IndeksZnamke] := −1;    { vse znamke v tej poštnini. }
      end; {if}
    end
    else begin
      { Prejšnji rekurzivni klic ni ustvaril nobene nove poštnine, torej se lahko
        naše iskanje konča; rekonstruirajmo znamke v najboljši najdeni poštnini. }
      OstanekPostnine := Postnina + Presezek;
      NizZnamk[IndeksZnamke] := −1;
    end; {if}
  end; {NajdiNizZnamk}

  procedure IzpisiNizZnamk(NizZnamk: NizT);
  var i : integer;
  begin
    i := 1;
    while NizZnamk[i] <> −1 do
      begin Write(NizZnamk[i], ' '); i := i + 1 end
    WriteLn;
  end; {IzpisiNizZnamk}

  begin {Znamke}
    BeriNizVrednosti(VrednostiZnamk);
    ReadLn(Postnina);
    Inicializacija;
    NajdiNizZnamk(Postnine, PokriteVrednosti, OstanekPostnine, NizZnamk, 1);
    IzpisiNizZnamk(NizZnamk);
  end. {Znamke}
```

V gornjem programu opravi glavnino dela podprogram NajdiNizZnamk. Ta v strukturi TrenPostnine dobi seznam poštnin, ki se jih je dalo sestaviti z IndeksZnamke − 1 znamkami (ne pa z manj). Za i-to izmed teh poštnin je v TrenPostnine[1, i] njena vrednost, v TrenPostnine[2, i] pa vrednost zadnje znamke, dodane k tej poštnini (slednje pride prav, da lahko točno rekonstruiramo skupino znamk, ki tvori neko poštnino). Konec seznama je označen s tem, da je TrenPostnine[1, i] = −1.

Podprogram NajdiNizZnamk poskuša vsaki od poštnin iz TrenPostnine na vse možne načine dodati še po eno znamko; če je nova poštnina obetavna (torej če je še nismo videli in če še ne poznamo kakšne boljše rešitve), jo doda v strukturo

NovePostnine (ki ima enako zgradbo kot TrenPostnine). Nove poštnine, ki se nam tako naberejo, uporabimo kot izhodišče za nov rekurzivni klic, v katerem bomo poskušali dodati še po eno znamko. Če nismo uspeli sestaviti nobene nove poštnine, se rekurzija neha, podprogram NajdiNizZnamk pa v tem primeru shrani vrednost najboljše doslej najdene poštnine v spremenljivko OstanekPostnine.

Klicatelj lahko preveri, katera je bila nazadnje dodana znamka v poštnino z vrednostjo OstanekPostnine in to znamko doda v tabelo NizZnamk, njeno vrednost pa odšteje od OstanekPostnine, tako da bodo lahko tudi višje ležeči rekurzivni klici ugotovili, kako se nadaljuje sestava najboljše poštnine.

Naloga: str. 4

R1994.3.3

V spremenljivki StanjeMotorja si bomo zapomnili, kaj smo motorju nazadnje naročili, v spremenljivki Ukaz pa, kaj bi radi od njega zdaj. Vrednost MinSirina pove, kako na široko bi radi odprli vrata. Na primer, če hoče kdo vstopiti ali pa hoče izstopiti avtomobil, moramo naročiti odpiranje vrat do polne širine; če hoče izstopiti pešec, pa je dovolj, če naročimo odpiranje do polovične širine (če pa je že naročeno odpiranje do polne širine, npr. ker je hotel tik pred tem nekdo vstopiti, nam ni treba spreminjati ničesar). Ko se vrata pri odpiranju dovolj odprejo, moramo motor ustaviti, enako pa tudi, ko se pri zapiranju zaprejo do konca. Ko se motor ustavi in vrata niso zaprta, nastavimo budilko, ponovno pa jo nastavimo tudi, če so vrata še vedno odprta in se pojavi ovira ali pa kakšna zahteva za vstop ali izstop. Ko se čas budilke izteče, začnemo vrata zapirati. Če zaznamo oviro, vrata pa niso čisto zaprta, jih moramo začeti spet odpirati (do stare širine, ki je še vedno v MinSirina — šele ko se čisto zaprejo, postavimo MinSirina na 0).

```
program Vrata;
type MotorT = (Stoj, Odpiraj, Zapiraj);
var
  MinSirina: real;
  Ukaz, StanjeMotorja: MotorT;
  UraTece: boolean;

  function OdprtostVrat: real; external;
  function IzstopPesec: boolean; external;
  function IzstopAvto: boolean; external;
  function Vstop: boolean; external;
  function Ovira: boolean; external;
  procedure Motor(Ukaz: MotorT); external;
  procedure NastaviBudilko(s: integer); external;
  function PreostaliCas: integer; external;

begin
  MinSirina := 0; UraTece := false;
  StanjeMotorja := Stoj; Motor(Stoj);
  repeat
    Ukaz := StanjeMotorja;
    if Vstop or IzstopAvto then
      begin Ukaz := Odpiraj; MinSirina := 1 end;
    if IzstopPesec and ((MinSirina = 0) or (Ukaz = Zapiraj)) then begin
      if MinSirina = 0 then MinSirina := 0.5;
      Ukaz := Odpiraj;
    end; {if}
    if Ovira and (OdprtostVrat > 0) then Ukaz := Odpiraj;
```

```
    if UraTece and (PreostaliCas = 0) then
      begin Ukaz := Zapiraj; UraTece := false end;

    if (Ukaz = Odpiraj) and (OdprtostVrat >= MinSirina) then Ukaz := Stoj;
    if (Ukaz = Zapiraj) and (OdprtostVrat = 0) then
      begin Ukaz := Stoj; MinSirina := 0 end;
    if Ukaz <> StanjeMotorja then begin
      Motor(Ukaz); StanjeMotorja := Ukaz;
      if (Ukaz = Stoj) and (OdprtostVrat > 0) then
        begin NastaviBudilko(15); UraTece := true end;
    end; {if}

    if StanjeMotorja <> Stoj then
      begin NastaviBudilko(0); UraTece := false end;
    if (StanjeMotorja = Stoj) and (OdprtostVrat > 0) and
       (Vstop or IzstopAvto or IzstopPesec or Ovira) then
      begin NastaviBudilko(15); UraTece := true end;
  until false;
end. {Vrata}
```

R1994.3.4

Naloga: str. 5

Nalogo lahko rešimo z rekurzijo. Oklepajni izraz je ali oblike $*$ ali pa $[A_1 A_2 \cdots A_k]$, pri čemer so $A_1, \ldots, A_k$ spet oklepajni izrazi. i-temu znaku $*$ našega oklepajskega izraza pripada na sliki navpična črtica enotske dolžine na x-koordinati i (oz. $i-1$, če hočemo, da se drevo začne na x-koordinati 0), y-koordinata pa je odvisna od globine gnezdenja (v koliko oklepajev je ovit opazovani znak $*$). Vsakemu podizrazu A_j pa pripada neko manjše poddrevo celotnega drevesa; y-koordinata, na kateri se nahaja vrh tega drevesa, je spet odvisna od globine gnezdenja, x-koordinata pa od tega, katere znake $*$ pokriva podizraz A_j. Da narišemo drevo izraza $[A_1 A_2 \cdots A_k]$, moramo narisati vsa poddrevesa za izraze $A_1, \ldots, A_k$ ter nato njihove vrhove povezati z vodoravno črto in na sredi nad njo še kratko navpično črto.

Podprogram RisiDrevo v spodnjem programu nariše zaporedje dreves za izraze $A_1, \ldots, A_k$ in prek parametrov LeviX in DesniX vrne x-koordinato vrha prvega ter zadnjega drevesa (torej drevesa za A_1 in drevesa za A_k). x-koordinato, na kateri je treba narisati naslednji znak $*$, lahko vzdržujemo prek globalne spremenljivke X, y-koordinato pa povečamo ob rekurzivnem klicu (parameter Globina). Rekurzivni klic izvedemo, čim zagledamo [; v okviru tega klica naj bi se izrisalo vse do pripadajočega zaklepaja], nato pa klicatelj prek vrednosti LeviPodX in DesniPodX tudi ve, po katerih koordinatah mora segati vodoravna črta nad poddrevesi.

```
program IzrazVDrevo;
uses Graph;

const MaksDolzinaNiza = 25; { največja dolžina niza }
type NizT = packed array [1..MaksDolzinaNiza] of char;
var Izraz: NizT;
    Z: integer;
    X, LeviX, DesniX : real;

procedure OdpriGraficniNacin;
var GDrv, GMode: integer;
begin
  GDrv := Detect;
```

```
  InitGraph(GDrv, GMode, 'c:\bp\bgi');
  if GraphResult <> grOk then Halt(1);
end; {OdpriGraficniNacin}

procedure ZapriGraficniNacin;
begin
  CloseGraph;
end; {ZapriGraficniNacin}

procedure Crta(X1, Y1, X2, Y2: real);
begin
  Line(Round(X1 * 50), Round(Y1 * 50),
       Round(X2 * 50), Round(Y2 * 50));
end; {Crta}

procedure RisiDrevo(Globina: integer; var LeviX, DesniX: real);
var
  LeviPodX, DesniPodX: real;
  Prvic: boolean;
begin
  Prvic := true;
  repeat
    if Izraz[Z] = '[' then begin
      Z := Z + 1; RisiDrevo(Globina + 1, LeviPodX, DesniPodX);
      Crta(LeviPodX, Globina, DesniPodX, Globina);
      DesniX := LeviPodX + (DesniPodX - LeviPodX) / 2;
      if Prvic then begin LeviX := DesniX; Prvic := false end;
      Crta(DesniX, Globina, DesniX, Globina - 1);
    end
    else if Izraz[Z] = '*' then begin
      Z := Z + 1;
      Crta(X, Globina, X, Globina - 1);
      DesniX := X;
      if Prvic then begin LeviX := DesniX; Prvic := false end;
      X := X + 1;
    end; {if}
  until (Izraz[Z] = ']') or (Izraz[Z] = ' ');
  Z := Z + 1;
end; {RisiDrevo}

procedure Demo(Niz: NizT);
begin
  ClearViewPort; Izraz := Niz; Z := 1; X := 1;
  RisiDrevo(1, LeviX, DesniX); ReadLn;
end; {Demo}

begin {IzrazVDrevo}
  OdpriGraficniNacin;
  Demo('[[***]*[**]] ');
  Demo('[[***]*[*[[**]*][**]]] ');
  Demo('[[***]*[*[[**]*]]] ');
  Demo('[[[[*]]][**[*]*]] ');
  ZapriGraficniNacin;
end. {IzrazVDrevo}
```

Za ljubitelje skladovnega računanja pa je tule še rešitev v postscriptu:[8]

```
/xKorak 5 25.4 div 72 mul def          % 5 mm
/yKorak 5 25.4 div −72 mul def         % negativna, da ne bodo drevesa rastla navzgor

% Razcepi niz na vrhu sklada: s razcepi → (prva črka s) (preostanek s)
/razcepi
{
  /s exch def                          % poberimo parameter s sklada
  s 0 1 getinterval                    % prva črka
  s 1   s length 1 sub   getinterval   % preostanek
} def

% Nariše poddrevo: globina xPrvegaListaTegaDrevesa opis drevo1 →
/drevo1                                % xPrvegaListaZaTemDrevesom ostanekOpisa xKorena
{
  10 dict begin       % Odprimo nov slovar za lokalne spremenljivke.
  % Poberimo parametre s sklada.
  razcepi   /ostanekOpisa exch def   /prviZnak exch def
  /xLista exch def   /globina exch def
  prviZnak ([) eq     % Smo pri listu ali pri notranjem vozlišču?
  {
    /prviOtrok true def
    {
      % Narišimo naslednje poddrevo.
      globina yKorak add   xLista   ostanekOpisa   drevo1
      % Zapomniti si moramo x-koordinato korena prvega in zadnjega poddrevesa.
      prviOtrok
      { /prviOtrok false def
        dup /xPrvegaPoddrevesa exch def } if
      /xZadnjegaPoddrevesa exch def
      % Tole sta spremenljivki, ki bi ju radi pravzaprav prenašali „po referenci“.
      /ostanekOpisa exch def
      /xLista exch def
      % Smo pregledali vsa poddrevesa?
      ostanekOpisa razcepi exch
      (]) eq { /ostanekOpisa exch def   exit } { pop } ifelse
    } loop
    % Narišimo vodoravno prečko.
    xPrvegaPoddrevesa   globina yKorak add   moveto
    xZadnjegaPoddrevesa   globina yKorak add   lineto
    % Narišimo koren.
    /xKorena   xPrvegaPoddrevesa xZadnjegaPoddrevesa add   2 div   def
    xKorena   globina yKorak add   moveto
    xKorena   globina   lineto
    % Odložimo na sklad podatke, ki jih bo uporabljal klicatelj.
    xLista ostanekOpisa xKorena
  }
```

[8] Za več o postscriptu glej npr. Glenn C. Reid, *Thinking in PostScript*, Addison-Wesley, 1990; Adobe Systems Inc., *PostScript Language Reference Manual*, 3. izd., Addison-Wesley, 1999.

```
  {
    % Opraviti imamo z listom — narišimo le navpično črtico.
    xLista globina moveto
    xLista globina yKorak add lineto
    % Odložimo na sklad podatke, ki jih bo uporabljal klicatelj.
    xLista xKorak add   ostanekOpisa   xLista
  }
  ifelse
  end % zaprimo slovar
} def

/drevo
{
  /opis exch def   /y exch def   /x exch def      % Poberimo parametre s sklada.
  /Courier findfont   12 scalefont   setfont      % Izpišimo opis drevesa.
  x   y 5 add   moveto
  opis show
  newpath   y x opis drevo1   stroke              % Narišimo drevo.
  pop pop pop                                     % Počistimo sklad.
} def

100 100 ([[***]*[**]]) drevo
100 200 ([[***]*[*[[**]*][**]]]) drevo
100 300 ([[***]*[*[[**]*]]]) drevo
100 400 ([[[[*]]][**[*]*]]) drevo

showpage
```