16. državno tekmovanje v znanju računalništva (1992)

NALOGE ZA PRVO SKUPINO

1992.1.1

R: 6

Program za avtomatsko programiranje je iz primerov izluščil relacijo med argumentoma x in y ter vrednostjo funkcije in jo zapisal v obliki naslednjega programa. **Katero relacijo** se je naučil? Odgovor **utemelji**!

```
program KajStorim(Input, Output);
var
  x, y, a: 0..MaxInt;
  Kaj: boolean;
begin
  ReadLn(x, y);
  Kaj := true;
  while x <> 0 do begin
    a := x; x := y; y := a;
    y := y - 1;
    Kaj := not Kaj;
  end; {while}
  if Kaj then WriteLn('DA')
  else WriteLn('NE');
end. {KajStorim}
```

```
#include <stdio.h>

void main()
{
  unsigned int x, y, a, kaj;
  scanf("%u%u", &x, &y);
  kaj = 1;
  while (x != 0) {
    a = x; x = y; y = a;
    y--;
    kaj = !kaj;
  }
  if (kaj) printf("DA\n");
  else printf("NE\n");
}
```

1992.1.2

R: 6

Uslužbenci službe *Pomoč-informacije* sedijo vsak ob svojem telefonu in odgovarjajo na telefonske klice. Njihovi telefoni so povezani na lokalno telefonsko centralo, ki sprejema klice in jih dodeljuje posameznim (prostim) telefonistom.

Napiši postopek za upravljanje lokalne telefonske centrale, ki bo sprejemala klice iz telefonskega omrežja in jih dodeljevala telefonistom. Pri tem naj centrala pazi, da bodo vsi telefonisti enakomerno obremenjeni. Naslednji telefonski klic naj dodeli telefonistu, ki trenutno že najdlje počiva.

Lokalna telefonska centrala ima na razpolago naslednje podprograme:

Zvoni — vrne vrednost true, če telefonska centrala zaznava iz omrežja klic, ki še ni prevezan, sicer false.

Zveži(i) — centrala zveže klic iz omrežja z i-tim telefonistom.

Počiva(i) — vrne vrednost true, če je i-ti telefonist prost, sicer false.

R: 7 **1992.1.3** Mačka je igrivo razpoložena in lovi miško. Vsakič ko jo ujame, ji miška lahko uide takoj ali pa kasneje, ko se je mačka z njo v gobčku že nekaj časa potikala naokrog. **Napiši program**, ki bere koordinate, kjer sta se gibali mačka in miška, in izpiše, kolikokrat je miška pobegnila mački. Koordinate so pari celih števil.

R: 7 **1992.1.4** Na slavnostno večerjo kluba *Veseli bitki* je prišel tudi častni gost — računalničar K. Nuth. Častni gost ni nikogar poznal, njega pa so poznali vsi. Tik pred sladico je na veliki dogodek naravnost z letališča prihitel tuj novinar, ki se je hotel srečati s slavnim računalničarjem, ni pa znal niti besedice po naše. Med vožnjo z letališča se je naučil samo eno vprašanje: „Ali poznaš to osebo?“ Njegov načrt je bil preprost — stopil bo k vsakemu gostu in ga za vsakega ostalega gosta po vrsti vprašal, če ga morebiti pozna. Na koncu bo zagotovo vedel, kdo ne pozna nikogar.

Ko je prišel, je ves zgrožen ugotovil, da je povabljencev veliko preveč. Še preden bi uspel zbrati vse odgovore, bi bilo konec večerje. K sreči pa se da najti častnega gosta tudi z veliko manjkrat postavljenim vprašanjem. **V kakšnem vrstnem redu** bi ti spraševal, da bi čim hitreje našel častnega gosta?

NALOGE ZA DRUGO SKUPINO

R: 8 **1992.2.1** **Kaj izpiše naslednji program** za poljubno prebrano (nenegativno) število? Odgovor **utemelji**.

Opombi:

- Funkcija Xor izračuna „ekskluzivni ali“ po bitih dveh števil.
  $\mathsf{Xor}(0,0) = \mathsf{Xor}(1,1) = 1$; $\mathsf{Xor}(0,1) = \mathsf{Xor}(1,0) = 0$;
  $\mathsf{Xor}(10,9) = \mathsf{Xor}(001010_2, 001001_2) = 000011_2 = 3$.
- Funkcija Ord deluje tako:
  Ord(false) = 0, Ord(true) = 1.

```
program TheAnswer(Input, Output);
var
  s, j, m1, m2: 0..MaxInt;
begin
  ReadLn(s);
  m1 := 0; m2 := 0;
  for j := 7 downto 1 do begin
    m1 := 2 * m1 + Ord(Odd(s));
    m2 := 2 * m2
          + Ord(Odd(s) = Odd(j));
    s := s div 2;
  end;
```

```
#include <stdio.h>
void main()
{
  unsigned int s, j, m1, m2;
  scanf("%u", &s);
  m1 = m2 = 0;
  for (j = 7; j > 0; j--) {
    m1 = (m1 << 1) + (s & 1);
    m2 = (m2 << 1)
         + ((s & 1) == (j & 1));
    s >>= 1;
  }
```

```
  WriteLn(Xor(m1, m2));
end. {TheAnswer}
```

```
    printf("%u\n", m1 ↑ m2);
}
```

1992.2.2

R: 8

V Ministrstvu za raziskovanje rude in zapravljanje časa žigosata papirje dva uradnika, War in Bert. Vsak papir ožigosa najprej War, nato pa še Bert. Ker sta različno hitra, se lahko zgodi, da se naberejo papirji, ki jih je War že ožigosal, Bert pa še ne. Zato imajo v ministrstvu robota, ki skrbi za papirje na poti od Wara k Bertu. Robot je opremljen z roko, ki lahko drži en papir, in s skladiščem. V skladišču je prostora za natanko dva kupa papirjev, imenujmo ju A in B.

Robota upravljamo z naslednjimi podprogrami:

Nalozi(k: Kup) — naloži papir v roki na vrh kupa k,

Snemi(k: Kup) — vzame papir na vrhu kupa k v roko,

Prazen(k: Kup) — vrne true, če je kup k prazen, sicer false,

pri čemer ima k vrednost A, B ali M. Snemi(M) vzame papir z Warove mize, Nalozi(M) pa ga odloži na Bertovo mizo. Vrednost, ki jo vrne Prazen(M), ni definirana.

Ko War ožigosa papir, ga položi na mizo in pokliče proceduro ShraniPapir, ki shrani papir z njegove mize v skladišče, Bert pa pokliče proceduro VzemiPapir in dobi papir iz skladišča na svojo mizo.

Napiši podprograma ShraniPapir in VzemiPapir tako, da bo Bert žigosal papirje v enakem vrstnem redu, kot jih je žigosal War. Na kaj mora še paziti Bert?

1992.2.3

R: 10

Napiši podprogramsko funkcijo Val, ki dobi kot prvi parameter niz znakov (desetiški zapis števila) in ga pretvori v število, ki ga vrne kot tretji parameter. Dolžino vhodnega niza dobi kot drugi parameter. Vrednost funkcije naj bo true, če je pretvorba uspela, in false, če niz ne predstavlja legalnega števila.

```
const StrM = 100;
type string = packed array [1..StrM] of char;
function Val(Str: string; StrL: integer; var n: integer): boolean;
```

Legalno število je niz števk (znaki med '0' in '9'), pred katerim lahko stoji predznak (plus ali minus). Presledke pred in za številom naj podprogram ignorira. Niz mora predstavljati število z intervala MinInt ≤ n ≤ MaxInt.

Za predstavitev celih števil uporablja naš računalnik 16-bitne besede:

```
const
  MinInt = −32768;
  MaxInt = 32767;
type integer = MinInt..MaxInt;
```

Med preverjanjem in pretvorbo podprogram ne sme preseči obsega celih števil niti pri nelegalnih številih!

R: 11

1992.2.4

Robotski vrtalnik uporablja različno velike svedre, ki so shranjeni v luknjah na vrtljivem okroglem stojalu. Vrtalnik zna zamenjati sveder v vrtalnem stroju s svedrom, ki je v luknji ob vrtalnem stroju. Vrtalnik lahko vrti stojalo in tako doseže katerokoli luknjo.

Med delom se svedri pomešajo. **Napiši algoritem**, s katerim vrtalnik po opravljenem delu svedre uredi po velikosti.

Upoštevaj, da lahko vrtalnik opravlja samo naslednje preproste operacije:

1. Zavrti vrtljivo stojalo za eno luknjo v smeri urinega kazalca (podprogram Zavrti). V primerjavi z ostalimi tremi operacijami je premik stojala počasen.

2. Zamenja sveder, ki je vpet v vrtalnem stroju, s tistim, ki je v luknji ob vrtalnem stroju (podprogram Zamenjaj). Če je vrtalni stroj ali luknja prazna, se „zamenjata“ prazen prostor in sveder.

3. Podprogram StrojPrazen vrne true, če v vrtalnem stroju ni ničesar, in false, če je v njem vpet kak sveder.

4. Podprogram Primerjaj primerja velikost svedra v vrtalnem stroju s svedrom, ki je v luknji ob stroju. Vrne:

   - true, če je sveder v vrtalniku manjši od svedra v luknji in
   - false, če je sveder v vrtalniku večji od svedra v luknji.

   Vsi svedri so različno veliki. Prazna luknja/vrtalnik se pri primerjanju obnaša kot sveder velikosti 0 (je „manjša“ od vseh svedrov).

Pred začetkom urejanja je vrtalnik prazen in v vsaki luknji na stojalu je en sveder. Algoritem naj čim hitreje uredi svedre po velikosti od najmanjšega do največjega v smeri urinega kazalca.

NALOGE ZA TRETJO SKUPINO

R: 12

1992.3.1

Program za avtomatsko programiranje je iz primerov izluščil relacijo med argumentoma x in y ter vrednostjo funkcije in jo zapisal v obliki naslednjega programa. **Katero relacijo** se je naučil? Odgovor **utemelji**.

```
type pinteger = 0..MaxInt;

function Enostavna(x, y: pinteger): boolean;
begin
```

```
  if x = 0 then
    Enostavna := true
  else
    Enostavna := not Enostavna (y, x − 1);
end; {Enostavna}
```

```
int Enostavna(unsigned int x, unsigned int y)
{
  if (x == 0) return 1;
  else return !Enostavna(y, x − 1);
}
```

1992.3.2

R: 13

Pri lepem programiranju obsežnejše programe vedno razdelimo na več modulov. V enem modulu (odvisnem modulu) uporabljamo podatke in podprograme, ki so definirani v drugem modulu (podmodulu). Pred prevajanjem odvisnega modula mora prevajalnik prevesti vse njegove podmodule.

Napiši algoritem, ki bo izpisal pravilni vrstni red prevajanja modulov. Vhodni podatek algoritmu je ime glavnega modula. Pri tem lahko uporabljaš podprogram:

```
function Podmoduli(OdvisniModul: Ime): SeznamImen;
```

ki nam vrne seznam imen podmodulov, ki jih uporablja odvisni modul, katerega ime podamo kot parameter.

1992.3.3

R: 14

Napiši podprogram Najdaljse, ki v danem zaporedju celih števil poišče najdaljše strogo naraščajoče podzaporedje. Elemente podzaporedja izbiramo iz začetnega zaporedja od leve proti desni, pri tem pa lahko poljubno število originalnih elementov preskočimo.

Kadar imamo več enakovrednih rešitev, naj podprogram najde poljubno izmed njih.

Dva primera:

6 2 3 4 1	$\longrightarrow$	2 3 4	
1 1 3 2 6 8 1	$\longrightarrow$	1 3 6 8	ali
		1 2 6 8	

```
const N = ...;    { dolžina najdaljšega vhodnega zaporedja }
type Zaporedje = array [1..N] of integer;
procedure Najdaljse(A: Zaporedje;          { dano zaporedje }
                    DolA: integer;         { njegova dolžina }
                    var B: Zaporedje;      { vrne novo naraščajoče podzaporedje }
                    var DolB: integer);    { in njegovo dolžino }
```

R: 19 **1992.3.4** V grafičnem okolju (na primer X Windows, Macintosh, Amiga in MS Windows) so osnovni objekti okna. To so pravokotna območja na zaslonu; lahko se tudi prekrivajo. Del okna, ki leži pod kakim drugim oknom, ni viden. Uporabnik lahko okna premešča po zaslonu, jih zapira in jim spreminja velikost. Zato mora grafični vmesnik ob vsaki taki spremembi osvežiti vsebino zaslona. Če je kakšno okno (ali njegov del) postalo vidno, ga je treba na novo narisati.

Opiši postopek (algoritem), ki osveži vsebino danega okna:

algoritem OsveziOkno
  vhod: okno W
  učinek: osveži vsebino okna W

Tvoj algoritem lahko uporablja naslednje podprograme in podatke:

- Seznam vseh oken na zaslonu. Okna v seznamu so urejena glede na globino — okno, ki je višje (bolj spredaj) na zaslonu, nastopa prej v seznamu.
- Pravokotniki (in okna) na zaslonu so določeni s koordinatami levega zgornjega in desnega spodnjega oglišča. Privzemi, da je izhodišče koordinatnega sistema v levem zgornjem oglišču zaslona.
- Procedura Nariši(W: Okno; P: Pravokotnik) nariše na zaslon del okna W, ki je določen s pravokotnikom P. Pri tem mora biti pravokotnik P vsebovan v oknu W.

REŠITVE NALOG ZA PRVO SKUPINO

N: 1 **R1992.1.1** Program ugotavlja, ali velja relacija $x \leq y$. Prebrani števili izmenično zmanjšuje za 1, dokler ena od spremenljivk ni enaka 0. Kadarkoli je v spremenljivki Kaj vrednost true, je v x ustrezno zmanjšano drugo prebrano število. Zanka se konča, ko manjše od obeh prebranih števil zmanjšamo na 0. Iz spremenljivke Kaj izvemo, katero od obeh števil je to.

N: 1 **R1992.1.2** Informacijo o trenutni dejavnosti telefonistov hranimo v dveh seznamih. Seznam Zasedeni vsebuje številke vseh zasedenih telefonistov, seznam Prosti pa hrani številke vseh prostih telefonistov. Slednji je urejen tako, da so na začetku tisti telefonisti, ki so prosti že najdlje. To urejenost lahko zagotavljamo s tem, da jemljemo proste telefoniste vedno z začetka seznama; zasedene telefoniste, ki se jim je linija sprostila, pa dodamo vedno na konec.

Postopek je takle:

```
Zasedeni := prazen seznam; Prosti := seznam ⟨1, 2, ..., N⟩;
ponavljaj
  če Zvoni in Prosti ni prazen:
    e := prvi iz seznama Prosti; Zveži(e);
    Briši e iz seznama Prosti;
    Dodaj e na konec seznama Zasedeni;
  za vsak t v seznamu Zasedeni ponovi:
    če Počiva(t):
      Briši t iz seznama Zasedeni;
      Dodaj t na konec seznama Prosti;
```

R1992.1.3

N: 2

V spremenljivki Drzi si zapomnimo, ali je mačka v prejšnjem časovnem koraku držala miško. Če je Drzi = true in imata v naslednjem trenutku mačka in miška različen položaj, vemo, da je miška mački ušla in moramo povečati števec Usla. Drugače pa, če imata miška in mačka enak položaj, postavimo Drzi na true, da jo bomo lahko uporabili v naslednjem časovnem koraku.

```
program MackaInMiska(Input, Output);
var
  xMac, yMac, xMis, yMis: integer;
  Usla: integer;
  Drzi: boolean;
begin
  Drzi := false;
  Usla := 0;
  while not Eof do begin
    ReadLn(xMac, yMac, xMis, yMis);
    if (xMac = xMis) and (yMac = yMis) then Drzi := true
    else if Drzi then begin Drzi := false; Usla := Usla + 1 end;
  end; {while}
  WriteLn('Miška je ušla mački ', Usla, '-krat.');
end. {MackaInMiska}
```

R1992.1.4

N: 2

Problem slavne osebe (celebrity problem) ima res elegantno rešitev. Potrebujemo natanko eno vprašanje manj, kot je vseh povabljenih.

Povabljence oštevilčimo od 1 do N. Odgovor na vprašanje, ali a pozna b, nam da funkcija Pozna(a, b). Uporabimo še dve preprosti resnici. Če a pozna b, potem a gotovo ni častni gost, saj ta ne pozna nikogar. Če pa a ne pozna b, potem b gotovo ni častni gost, kajti le-tega vsi poznajo. Gosta 1 vprašamo, ali pozna gosta N. Če ga pozna, potem 1 gotovo ni častni gost in se premaknemo k naslednjemu. Sicer pa N gotovo ni častni gost in vprašamo gosta 1, ali pozna gosta $N - 1$. Postopek ponavljamo, dokler nam ne ostane samo še en gost — ta je iskani častni gost.

Oglejmo si še program:

```
program CastniGost(Input, Output);
const N = ...;
type GostT = 1..N;
var a, b: GostT;

  function Pozna(a, b: GostT): boolean; external;

begin
  a := 1;
  b := N;
  while a <> b do
    if Pozna(a, b) then a := a + 1 { Če a pozna b, potem a ni častni gost. }
    else b := b - 1;               { Če a ne pozna b, potem b ni častni gost. }
  WriteLn('Častni gost je oseba ', a, '.');
end. {CastniGost}
```

REŠITVE NALOG ZA DRUGO SKUPINO

N: 2

R1992.2.1

Program izpiše številko 42, ne glede na prebrani podatek. Zanka pretoči bite iz s v m1 v obratnem vrstnem redu; v m2 pa dobimo enice na tistih mestih binarnega zapisa, kjer se m1 razlikuje od števila $0101010_2 = 42$. Operacijo Xor si lahko razlagamo tudi kot operacijo, ki komplementira vse tiste bite prvega števila, kjer stoji na istoležnem binarnem mestu drugega števila enica. m2 torej v operaciji Xor obrne vse tiste bite v m1, ki ne ustrezajo predpisanemu vzorcu 0101010, tako da je rezultat operacije Xor(m1, m2) vedno 42.

N: 3

R1992.2.2

Pri nalogi gre pravzaprav za simulacijo vrste z dvema skladoma. Vrsta (*queue*) je podatkovna strukture oblike FIFO (*first in — first out*), ki deluje tako kot vrsta v banki — tisti, ki prej pridejo, so tudi prej na vrsti. Sklad (*stack*) pa je primer strukture LIFO (*last in — first out*) — kot če zložimo knjige v stolp. Snamemo jih lahko samo v vrstnem redu, nasprotnem od tistega, v katerem smo jih naložili.

Najprej zapišimo podprogram, ki ga bomo potrebovali nekoliko pozneje:

```
procedure Prekucni(OdKod, Kam: Kup);
{ Podprogram preloži vse, kar je na kupu OdKod, na kup Kam po en papir naenkrat.
  Pri tem se vrstni red papirjev ravno obrne. }
begin
  while not Prazen(OdKod) do begin
    Snemi(OdKod);
    Nalozi(Kam);
  end; {while}
end; {Prekucni}
```

Prvi način, ki nam pade na pamet, je verjetno ta, da papirje spravljamo tako, da jih nalagamo na kup A, pobiramo pa tako, da prekucnemo kup A na kup B, vzamemo vrhnjega ter prekucnemo kup B nazaj na kup A:

```
procedure ShraniPapir1;
begin
  Snemi(M); Nalozi(A);
end; {ShraniPapir1}

procedure VzemiPapir1;
begin
  Prekucni(A, B);
  Snemi(B); Nalozi(M);
  Prekucni(B, A);
end; {VzemiPapir1}
```

Velika slabost tega pristopa je zelo veliko število premikov, saj dvakrat premaknemo vse spravljene papirje za vsak papir, ki ga hočemo vzeti. Postopek lahko dvakrat izboljšamo tako, da prekucnemo kupček samo takrat, ko je to zares potrebno:

```
procedure ShraniPapir2;
begin
  Prekucni(B, A);
  Snemi(M); Nalozi(A);
end; {ShraniPapir2}

procedure VzemiPapir2;
begin
  Prekucni(A, B);
  Snemi(B); Nalozi(M);
end; {VzemiPapir2}
```

Ta metoda se obnese dobro, če spravljamo oz. jemljemo veliko papirjev naenkrat, za izmenično shranjevanje in jemanje pa je prav tako neučinkovita kot metoda 1. Daleč najbolje se odreže metoda 3, saj vsak papir preloži natanko enkrat:

```
procedure ShraniPapir3;
begin
  Snemi(M); Nalozi(A);
end; {ShraniPapir3}

procedure VzemiPapir3;
begin
  if Prazen(B) then Prekucni(A, B);
  Snemi(B); Nalozi(M);
end; {VzemiPapir3}
```

Bert mora tudi paziti, da ne poskuša vzeti papirja, kadar je skladišče prazno, saj ni nikjer določeno, kaj naredi procedura Snemi, če na kupu ni ničesar. Ali je skladišče prazno, se lahko prepriča s funkcijo:

```
function PraznoSkladisce: boolean;
begin
  PraznoSkladisce := Prazen(A) and Prazen(B);
end; {PraznoSkladisce}
```

N: 3

R1992.2.3

Najprej preskočimo presledke na začetku niza, če jih je kaj; nato pogledamo, če je prisoten predznak, in si ga zapomnimo v spremenljivki Sign. Pri branju števk se vrednost doslej prebranega dela števila nabira v n. Ko preberemo novo števko (z vrednostjo Dig), pomnožimo n z 10 in prištejemo Dig (če beremo negativno število, pa Dig odštejemo oz. ga pred prištevanjem pomnožimo z -1). Tako po vsaki dodatni števki n spet vsebuje vrednost števila, ki ga tvorijo doslej prebrane števke (in predznak). Preden dodamo novo števko, moramo še preveriti, da ne bo nova vrednost slučajno prevelika ali premajhna. Ker ne smemo prekoračiti vrednosti 32767, lahko, če je $n < 3276$, mirno pomnožimo n z 10 in prištejemo novo števko, saj bo rezultat v tem primeru največ 32759; če pa je $n = 3276$, bo $10n + d \leq 32767$, če je $d \leq 7$. Če je $n > 3276$, ga nikakor ne smemo pomnožiti z 10. Podobne pogoje lahko postavimo tudi pri negativnih številih. Ko se števke nehajo, lahko preberemo še nič ali več presledkov, če pa naletimo na kakšen drug znak, moramo javiti napako.

```
const
  StrM = 100;
  MinInt = −32768;
  MaxInt = 32767;
type
  integer = MinInt..MaxInt;
  string = packed array [1..StrM] of char;

function Val(Str: string; StrL: integer; var n: integer): boolean;
const
  MaxN = 3276 {MaxInt div 10}; ModMaxN = 7 {MaxInt mod 10};
  MinN = −3276 {−MinInt div 10}; ModMinN = 8 {−MinInt mod 10};
var
  j, Dig: integer;
  ok, Cont: boolean;
  Sign: (Plus, Minus);
begin
  ok := true; Sign := Plus; n := 0; j := 1;
  { Preskočimo presledke na začetku niza. }
  Cont := true;
```

```
  while Cont and (j <= StrL) do
    if Str[j] <> ' ' then Cont := false else j := j + 1;
  { Preberimo predznak (če je prisoten). }
  if Str[j] in ['+', '-'] then
    begin if Str[j] = '-' then Sign := Minus;    j := j + 1 end;
  { Berimo števke; pazimo, da ne bomo dobili prevelike ali premajhne vrednosti. }
  Cont := true;
  while Cont and (j <= StrL) do
    if not (Str[j] in ['0'..'9']) then Cont := false
    else begin
      Dig := Ord(Str[j]) - Ord('0');
      case Sign of
        Plus: if (n > MaxN) or ((n = MaxN) and (Dig > ModMaxN)) then
                                                          ok := false;
        Minus: begin
          if (n < MinN) or ((n = MinN) and (Dig > ModMinN)) then ok := false;
          Dig := -Dig;
        end;
      end; {case}
      if not ok then Cont := false
      else begin n := 10 * n + Dig; j := j + 1 end;
    end; {else, while}
  if Dig = -1 then ok := false; { Namesto števk je tu nek drug znak. }
  { Preskočimo presledke na koncu niza. }
  Cont := true;
  while Cont and (j <= StrL) do
    if Str[j] <> ' ' then Cont := false else j := j + 1;
  if j <= StrL then ok := false; { Poleg presledkov je za številko še nekaj drugega. }
  Val := ok;
end; {Val}
```

R1992.2.4

N: 4

Rešitev je dobro znani postopek za urejanje z mehurčki (*bubble sort*). Večina ostalih postopkov urejanja predpostavlja, da se lahko premikamo po tabeli, ki jo urejamo, v obe smeri ali pa tudi delamo še kakšne daljše skoke, vse to pa bi bilo pri našem mehanizmu premikanja (ki lahko vrti stojalo le v eno smer in vsakič le za eno luknjo) zelo potratno s časom.

Radi bi, da bi bili svedri na stojalu urejeni naraščajoče v smeri urinega kazalca; torej, ko se stojalo premakne za eno mesto v smeri urinega kazalca, vidimo mi zdaj pred vrtalnikom luknjo, ki je za eno mesto pred tisto, ki smo jo videli prej, zato bi morali videti zdaj (če bi imeli svedre že pravilno urejene) manjši sveder kot na prejšnjem mestu. Torej, če imamo nek sveder v vrtalniku, pa opažamo na stojalu same svedre, ki so večji od njega, ni nič narobe, če gremo mimo njih; ko pa pridemo mimo takega, ki je manjši od tistega v vrtalniku, bi bilo škoda, če bi se premaknili naprej, kajti če bomo tistega v vr-

talniku nekje kasneje odložili na stojalo, bo prišel v smeri urinega kazalca pred tistim manjšim, ki ga zdajle vidimo v vrtalniku. Torej v tem primeru raje zamenjajmo sveder v vrtalniku s tistim na stojalu in potem postopek nadaljujmo s tem manjšim svedrom, ki ga imamo zdaj po novem v vrtalniku. Ko pridemo na stojalu do praznega mesta, torej tja, kjer smo začeli, smo pravzaprav na začetku zaporedja; ker smo vsakič, ko smo naleteli na nek sveder, manjši od tistega, ki je bil trenutno v vrtalniku, izvedli zamenjavo, imamo zdaj v vrtalniku očitno najmanjšega izmed vseh svedrov, tako da ne kaže drugega, kot da ga odložimo na trenutno prazno mesto. Tako bo najmanjši sveder prišel na začetek zaporedja, kamor tudi sodi. Zdaj se lahko z praznim vrtalnikom premaknemo spet na naslednje mesto, poberemo tamkajšnji sveder in vse skupaj ponovimo. Če enkrat naredimo poln krog po stojalu, ne da bi pri tem kdaj nesli manjši sveder mimo večjega (z drugimi besedami: če po vsakem koraku naredimo zamenjavo), pomeni, da so svedri urejeni in se postopek lahko konča.

```
procedure Uredi;
var
  Konec: boolean;
begin
  repeat
    Konec := true;
    Zamenjaj;
    Zavrti;
    repeat
      while Primerjaj do begin
        Zavrti;
        Konec := false;
      end; {while}
      Zamenjaj;
      Zavrti;
    until StrojPrazen;
  until Konec;
end; {Uredi}
```

REŠITVE NALOG ZA TRETJO SKUPINO

N: 4

R1992.3.1

Funkcija Enostavna računa relacijo „manjše ali enako“. To se najhitreje vidi, če jo zapišemo kot rekurzivno matematično relacijo $f(x, y)$:

$$f(x,y) = \begin{cases} \mathsf{true}, & \text{če je } x = 0 \\ \neg f(y, x-1), & \text{sicer.} \end{cases}$$

Če sta x in y strogo večja od nič, smemo razviti dva koraka rekurzije:

$$f(x,y) = \neg f(y, x-1) = \neg\neg f(x-1, y-1) = f(x-1, y-1).$$

Za $x, y \geq k \geq 0$ smemo uporabiti zgornjo lastnost k-krat:

$$f(x,y) = f(x-1,y-1) = f(x-2,y-2) = \ldots = f(x-k,y-k).$$

Sedaj obravnavamo tri primere:

1. če je $x < y$, je $f(x,y) = f(x-x,y-x) = f(0,y) = \mathsf{true}$;
2. če je $x = y$, je $f(x,y) = f(x,x) = f(1,1) = f(0,0) = \mathsf{true}$;
3. če je $x > y$, je $f(x,y) = f(x-y,y-y) = f(x-y,0) = \neg f(0,x-y-1) = \neg\mathsf{true} = \mathsf{false}$.

Vidimo, da funkcija vrne vrednost true v prvem in drugem primeru ter false v tretjem. Torej je to res relacija $\leq$.

R1992.3.2

N: 5

Algoritem VrstniRed uporablja rekurzivni algoritem VrstniRed2. Zapis S $\oplus$ T pomeni „stakni seznama S in T“. Ko pripravljamo vrstni red prevajanja modulov, morajo biti vsi podmoduli, od katerih je nek modul odvisen, v tem seznamu pred njim. Če imajo ti tudi svoje pod-podmodule, morajo biti ti v seznamu še prej in tako dalje. VrstniRed2 zato pokliče najprej samega sebe za vse podmodule trenutnega modula, nato pa doda trenutni modul na konec zaporedja. Pri tem je koristno še voditi seznam odvisnih modulov („nadmodulov“), ki so pripeljali do trenutnega rekurzivnega klica: če bi se kakšen od njih pojavil kot podmodul, pomeni, da so odvisnosti med moduli ciklične in bi bilo dobro sporočiti to kot napako. Seveda moramo paziti tudi, da ne bi istega modula po večkrat dodali v zaporedje, kar bi se drugače čisto lahko zgodilo, če bi se (ali ta modul ali pa nek drug, ki je od njega odvisen) pojavil kot podmodul pri več drugih modulih.

algoritem VrstniRed2
vhod: GlavniModul: ime modula;
Odvisni: seznam modulov;
var Zaporedje: seznam modulov;
izhod: V seznam Zaporedje doda spisek modulov, ki jih je treba prevesti, preden prevedemo glavni modul. Nazadnje doda na konec seznama Zaporedje še modul GlavniModul. Pri tem upošteva, da so moduli iz seznama Odvisni odvisni od modula GlavniModul. Tako lahko odkrije ciklične klice podmodulov.
lokalni spremenljivki:
S: seznam;
M: ime modula;
postopek:
S := Podmoduli(GlavniModul);
ponavljaj za vsak modul M iz seznama S:

```
      če je modul M v seznamu Odvisni,
        so odvisnosti med moduli ciklične. Napaka — prekini izvajanje.
      če modula M še ni v seznamu Zaporedje,
        VrstniRed(M, Odvisni ⊕ ⟨M⟩, Zaporedje);
    Zaporedje := Zaporedje ⊕ ⟨GlavniModul⟩;
```

algoritem VrstniRed
vhod: GlavniModul: ime modula, ki ga želimo prevesti;
izhod: seznam modulov, ki določa vrstni red prevajanja;
lokalna spremenljivka: S: seznam modulov;
postopek:
  S := ⟨⟩;
  VrstniRed2(GlavniModul, ⟨⟩, S);
  vrni S;

N: 5

R1992.3.3

Recimo, da imamo vhodno zaporedje A z n elementi. Omejimo se za začetek na vprašanje, kako dolgo je najdaljše strogo naraščajoče (v nadaljevanju: s. n.) podzaporedje, kasneje pa se bomo ukvarjali še s tem, kako priti do elementov tega podzaporedja. Poskusimo pregledovati elemente zaporedja A od leve proti desni in v neki spremenljivki, recimo k, vzdrževati dolžino najdaljšega strogo naraščajočega podzaporedja v doslej pregledanem delu tabele A. Hitro se lahko prepričamo, da se, če tabeli A dodamo en nov element, vrednost k lahko samo poveča 1 ali pa ostane enaka.[1] Naš postopek bo torej nekako tak:

```
k := 0;
for i := 1 to n do
  if se v A[i] konča kakšno s. n. podzaporedje dolžine k + 1 then k := k + 1;
```

Če naj se v $A[i]$ konča kakšno s. n. podzaporedje dolžine $k+1$, mora za predzadnji člen tega podzaporedja (recimo mu $A[j]$) veljati naslednje troje: $j < i$; pri $A[j]$ se mora končati neko s. n. podzaporedje dolžine k; in $A[j] < A[i]$ (ker imamo opravka s strogo naraščajočim podzaporedjem). Koristno bi bilo torej vedeti, kateri je izmed doslej pregledanih elementov tabele A najmanjši tak (recimo mu v), pri katerem se konča neko s. n. podzaporedje dolžine k; če niti ta element ni manjši od $A[i]$, potem tudi nobeden izmed ostalih elementov,

[1] Naj bo $L[i]$ dolžina najdaljšega s. n. podzaporedja tabele $A[1..i]$. Mislimo si zdaj neko najdaljše s. n. podzaporedje tabele $A[1..i]$; če to podzaporedje ne vsebuje elementa $A[i]$, je hkrati že tudi podzaporedje tabele $A[1..i-1]$; če pa vsebuje element $A[i]$, mu ga lahko odvzamemo in dobimo neko s.n. podzaporedje tabele $A[1..i-1]$ (dolgo $L[i]-1$ elementov); v obeh primerih torej vidimo, da tabela $A[1..i-1]$ vsebuje neko s. n. podzaporedje dolžine vsaj $L[i]-1$, torej je $L[i-1] \geq L[i]-1$ oz. $L[i] \leq L[i-1]+1$. Ker je vsako s. n. podzaporedje tabele $A[1..i-1]$ hkrati tudi s. n. podzaporedje tabele $A[1..i]$, mora veljati seveda tudi $L[i-1] \leq L[i]$. Če oboje združimo, vidimo, da mora biti $L[i]$ enak eni od vrednosti $L[i-1]$ in $L[i-1]+1$.

pri katerih se konča kakšno s. n. podzaporedje dolžine k, ne bo manjši od $A[i]$ in bomo vedeli, da se v $A[i]$ ne končuje nobeno s. n. podzaporedje dolžine $k+1$.

```
k := 0; v := −∞;
for i := 1 to n do
  if v < A[i] then begin k := k + 1; v := A[i] end
  else . . . ;
```

Kaj moramo postaviti namesto treh pik? Če pride izvajanje programa do njih, pomeni, da se pri $A[i]$ ne konča nobeno s. n. podzaporedje dolžine $k+1$; je pa še vedno mogoče, da se pri njem konča vsaj kakšno s. n. podzaporedje dolžine k in v tem primeru je $A[i]$, ker je manjši ali enak v, najmanjši doslej odkriti element, pri katerem se konča kakšno s. n. podzaporedje dolžine n; zato si ga moramo v tem primeru zapomniti kot novo vrednost v.

```
k := 0; v := −∞;
for i := 1 to n do
  if v < A[i] then begin k := k + 1; v := A[i] end
  else if se v A[i] konča kakšno s. n. podzaporedje dolžine k then v := A[i];
```

Zdaj smo pri podobnem problemu kot prej: kako preveriti, če se v $A[i]$ konča kakšno s. n. podzaporedje dolžine k? Če bi že poznali najmanjšega izmed doslej pregledanih elementov, pri katerem se konča kakšno s. n. podzaporedje dolžine $k-1$ — recimo temu elementu v' — bi lahko podobno kot prej preverili, ali je $A[i] > v'$. Če je, se pri $A[i]$ res konča neko s. n. podzaporedje dolžine k, drugače pa ne; toda če se ne, se mogoče pri njem konča neko s. n. podzaporedje dolžine $k-1$ in je zato $A[i]$ kandidat za novo vrednost v'.

```
k := 0; v := −∞; v′ := −∞;
for i := 1 to n do
  if v < A[i] then begin k := k + 1; v := A[i] end
  else if v′ < A[i] then v := A[i]
  else if se v A[i] konča kakšno s. n. podzaporedje dolžine k − 1 then v′ := A[i];
```

Vidimo, da smo padli v zanko; potrebovali bi tudi vrednost v'', ki bi povedala najmanjši element, pri katerem se konča kakšno s. n. zaporedje dolžine $k-2$, itd., itd. Zato vpeljimo kar tabelo $v[1..n]$, v kateri je $v[j]$ vrednost najmanjšega doslej prebranega elementa tabele A, pri katerem se končuje kakšno s. n. podzaporedje dolžine j. Namesto zaporedja stavkov **if** pa bomo morali uporabiti zanko.

```
k := 1; v[1] := A[1];
for i := 2 to n do
  if v[k] < A[i] then begin k := k + 1; v[k] := A[i] end
  else begin
```

```
    j := k;
    while j > 1 do
      if v[j − 1] < A[i] then break else j := j − 1;
    v[j] := A[i];
  end; {if}
```

Majhna slabost tega postopka je v notranji zanki (**while**), ki pregleduje tabelo v, da bi našla največji j, pri katerem je $v[j] < A[i]$. Če imamo smolo, bo morala pri tem pogosto pregledati velik del tabele v; primer je zaporedje

$$A = \langle \tfrac{n}{2}+1, \tfrac{n}{2}+2, \ldots, n, 1, 2, 3, 4, \ldots, \tfrac{n}{2} \rangle.$$

Po pregledu prve polovice tega zaporedja bomo imeli $k = n/2$ in $v = \langle \frac{n}{2} + 1, \ldots, n \rangle$, ker pa so preostali elementi tabele A tako majhni, bo morala iti notranja zanka pri vsakem od njih po celi tabeli v, od desne proti levi vse do prvega elementa. Tako vidimo, da je časovna zahtevnost našega postopka v najslabšem primeru $O(n^2)$.

Tej težavi se lahko izognemo, če tabele v ne preiskujemo po vrsti. Če se pri nekem elementu tabele A konča neko zaporedje dolžine j, se konča pri njem seveda tudi neko zaporedje dolžine $j-1$, zato mora biti vrednost $v[j-1]$ manjša ali enaka $v[j]$. Vrednosti v tabeli v so torej urejene nepadajoče, zato lahko v njej za iskanje največjega elementa $v[j]$, ki je še manjši od $A[i]$, uporabimo kar bisekcijo. Pri tem postopku vzdržujemo spodnjo in zgornjo mejo za iskani indeks j, v vsaki iteraciji zanke pa eno od obeh mej premaknemo tako, da se razdalja med njima razpolovi. Zato pridemo do pravega indeksa že po $O(\log n)$ korakih, časovna zahtevnost celotnega postopka pa je tako $O(n \log n)$ namesto $O(n^2)$. Zanko **while** iz prejšnjega postopka moramo zamenjati z nečim takšnim:

```
p := 0; j := k;
while j − p > 1 do begin
  { Na tem mestu velja v[p] < A[i] ≤ v[j]. Za v[0] si mislimo, da je enak −∞. }
  h := (p + j) div 2;
  { Ker je p ≥ 0 in j − p ≥ 2, se ni treba bati, da bi bil h = 0. }
  if v[h] < A[i] then p := h else j := h;
end; {while}
```

Na koncu je $p = j-1$ in torej velja $v[j-1] < A[i] \le v[j]$, torej smo prišli ravno do tistega j, ki ga iščemo: najdaljše s. n. podzaporedje s koncem pri $A[i]$ je dolgo j elementov.

S tem, kar smo naredili doslej, znamo poiskati dolžino najdaljšega s. n. podzaporedja tabele A — to je kar vrednost k ob koncu postopka. Naloga pa zahteva tudi primer takega podzaporedja, ne le njegove dolžine. Do njega lahko pridemo na več načinov. Ena možnost je, da poleg tabele v uvedemo še eno tabelo, recimo w: če v $v[j]$ piše, kateri doslej prebrani element tabele A je najmanjši tak, da se pri njem končuje kakšno s. n. podzaporedje dolžine j,

naj v $w[j]$ piše indeks tega elementa znotraj tabele A. Potem, ko pri nekem novem elementu $A[i]$ ugotovimo, da se pri njem končuje neko s. n. podzaporedje dolžine j, vemo, da je element $A[w[j-1]]$ lahko njegov predhodnik v nekem takem podzaporedju. Te predhodnike si zapisujmo v neko tabelo, pa bomo iz nje na koncu rekonstruirali najdaljše naraščajoče zaporedje.

```
procedure Najdaljse(A: Zaporedje; DolA: integer;
                    var B: Zaporedje; var DolB: integer);
var v, w, Predhodnik: Zaporedje; i, j, h, p, k: integer;
begin
  k := 1; v[1] := A[1]; w[1] := 1; Predhodnik[1] := 0;
  for i := 2 to n do begin
    if v[k] < A[i] then begin k := k + 1; j := k end
    else begin
      p := 0; j := k;
      while j - p > 1 do begin
        h := (p + j) div 2;
        if v[h] < A[i] then p := h else j := h;
      end; {while}
    end; {if}
    v[j] := A[i]; w[j] := i;
    if j = 1 then Predhodnik[i] := 0
    else Predhodnik[i] := w[j - 1];
  end; {for}
  { Rekonstruirajmo najdaljše strogo naraščajoče zaporedje. }
  DolB := k; i := w[k];
  for j := k downto 1 do
    begin B[j] := i; i := Predhodnik[i] end;
end; {Najdaljse}
```

Eleganten način za rekonstrukcijo najdaljšega s. n. podzaporedja pa je tudi ta, da si za vsak i zapomnimo dolžino najdaljšega s. n. podzaporedja s koncem pri $A[i]$; recimo tej dolžini $L[i]$. Ko je ta tabela pripravljena, jo preglejmo od konca proti začetku; za zadnji element našega s. n. podzaporedja vzemimo kar največji i z lastnostjo $L[i] = k$. Če smo že našli j-ti člen naraščajočega zaporedja (recimo na indeksu i), je primeren $(j-1)$-vi člen kar največji i', za katerega velja, da je $i' < i$ in $L[i'] = L[i] - 1$. (Zagotovo obstaja kak tak i', saj smo lahko med pripravo tabele L do vrednosti $L[i]$ prišli le tako, da smo povečali za 1 vrednost $L[i']$ za nek $i' < i$.) Kako se prepričamo, da v tem primeru res velja $A[i'] < A[i]$? Recimo, da bi vendarle veljalo $A[i'] \geq A[i]$; torej pravi predhodnik i-ja v naraščajočem zaporedju, ki ga iščemo, ne more biti i', pač pa nek i''; zanj mora veljati $L[i''] = L[i] - 1$, ker pa takega elementa med i' in i ni (zaradi načina, kako smo prišli do i'), mora biti $i'' < i'$. Obenem je seveda tudi $A[i''] < A[i]$ (saj smo rekli, da je i'' predhodnik i-ja v s. n. podzaporedju) in zato tudi $A[i''] < A[i']$. Toda to pomeni, da

bi lahko naraščajoče zaporedje do i'' nadaljevali z elementom na indeksu i', tako da mora veljati $L[i'] > L[i'']$, to pa je v nasprotju z našimi dosedanjimi predpostavkami, po katerih sta si $L[i']$ in $L[i'']$ enaka (ker sta oba enaka $L[i] - 1$). Tako torej vidimo, da je $A[i']$ prav gotovo manjši od $A[i]$ (saj bi sicer prišli v protislovje) in zato primeren za njegovega predhodnika v naraščajočem zaporedju, ki ga skušamo rekonstruirati.

```
procedure Najdaljse2(A: Zaporedje; DolA: integer;
                     var B: Zaporedje; var DolB: integer);
var v, L: Zaporedje; i, j, h, p, k: integer;
begin
  k := 1; v[1] := A[1]; L[1] := 1; Predhodnik[1] := 0;
  for i := 2 to n do begin
    ... { Enak stavek if kot prej. }
    v[j] := A[i]; L[i] := j;
  end; {for}
  { Rekonstruirajmo najdaljše strogo naraščajoče zaporedje. }
  DolB := k; j := k; i := n;
  while j > 0 do begin
    if L[i] = j then begin B[j] := A[i]; j := j - 1 end
    else i := i - 1;
end; {Najdaljse2}
```

Tako porabimo eno tabelo manj, časovna zahtevnost rekonstrukcije najdaljšega s. n. podzaporedja je v obeh primerih enaka $O(n)$.[2]

---

[2] O problemu najdaljšega naraščajočega podzaporedja je precej literature ("longest increasing subsequence" ali "longest ascending subsequence" ali "longest upsequence"). Lepa razlaga tu opisanega postopka je v E. W. Dijkstra, *Some beautiful arguments using mathematical induction*, Acta Informatica, 13(1):1–8, January 1980. Če v našem zaporedju A nastopajo dolga strnjena naraščajoča podzaporedja, lahko postopek še pospešimo, če tabelo v popravljamo z neke vrste zlivanjem (R. B. K. Dewar, S. M. Merritt, M. Sharir: *Some modified algorithms for Dijkstra's longest upsequence problem*, Acta Informatica 18(1):1–15, November 1982).

Če so v našem zaporedju A same različne vrednosti, recimo cela števila z intervala $1, \ldots, u$, bodo tudi v tabeli v same različne vrednosti; zato lahko njene elemente namesto v tabeli hranimo v stratificiranem drevesu in vsaka iteracija glavne zanke traja le še $O(\log \log u)$ časa (namesto $O(\log n)$ kot doslej zaradi bisekcije po tabeli v). Več o stratificiranih drevesih najdemo v P. van Emde Boas, R. Kaas, E. Zijlstra: *Design and implementation of an efficient priority queue*, Math. Systems Theory, 10:99–127, 1977; P. van Emde Boas: *Preserving order in a forest in less than logarithmic time and linear space*, Inform. Process. Lett. 6(3):80–82, June 1977; K. Mehlhorn, S. Näher: *Bounded ordered dictionaries in $O(\log \log n)$ time and $O(n)$ space*, Inform. Process. Lett. 35(4):183–189, 7 August 1990.

Če kot zaporedja jemljemo permutacije n različnih števil, je povprečna dolžina najdaljšega naraščajočega podzaporedja približno $2\sqrt{n}$. Če bi nas zanimala poljubna monotona podzaporedja (torej tako naraščajoča kot padajoča), pa se izkaže, da ima najdaljše tako podzaporedje zagotovo vsaj $\sqrt{n}$ elementov (Erdös in Szekeres, 1935; preprost dokaz v P. Pritchard, *Another look at the "Longest Ascending Subsequence" problem*, Acta Informatica, 16(1):87–91, August 1981).

R1992.3.4

N: 6

Algoritem najprej najde vse dele okna W, ki so vidni. Hrani jih v seznamu pravokotnikov SeznP. Na začetku postavi v seznam kar cel pravokotnik W.pravokotnik. Potem po vrsti od seznama „odreže“ pravokotnike V.pravokotnik za vsa okna V, ki so pred oknom W. Na koncu seznam SeznP vsebuje natanko tiste dele okna W, ki so vidni. Za vsakega izmed njih izvede Nariši(W, P).

Napišimo vse to bolj podrobno. Uporabimo naslednje oznake:

V.pravokotnik	pravokotnik, s katerim je določeno okno V
V.predhodnik	okno, ki je neposredno pred oknom V
$P.x_1, P.y_1$	koordinati zgornjega levega oglišča pravokotnika P
$P.x_2, P.y_2$	koordinati zgornjega levega oglišča pravokotnika P
(x_1, x_2, y_1, y_2)	pravokotnik, določen s temi koordinatami; na zaslonu mu pripadajo točke (x, y) za $x_1 \le x < x_2$ in $y_1 \le y < y_2$.
$\langle a, b, c \rangle$	seznam z elementi a, b, c

algoritem OdrežiDvaPravokotnika
vhod: pravokotnika A in B
izhod: seznam pravokotnikov, ki jih dobimo, ko odrežemo B od A in ostanek razdelimo na manjše pravokotnike
lokalna spremenljivka: seznam pravokotnikov Sezn

1. Če je $B.x_2 \le A.x_1$ ali $A.x_2 \le B.x_1$ ali $B.y_2 \le A.y_1$ ali $A.y_2 \le B.y_1$, postavi Sezn := $\langle A \rangle$.
   Pojdi na korak 7.
2. Sezn := $\langle\rangle$;
3. Če je $B.y_1 > A.y_1$, dodaj v Sezn pravokotnik $(A.x_1, A.y_1, A.x_2, B.y_1)$;
4. Če je $B.y_2 < A.y_2$, dodaj v Sezn pravokotnik $(A.x_1, B.y_2, A.x_2, A.y_2)$;
5. Če je $B.x_1 > A.x_1$, dodaj v Sezn pravokotnik $(A.x_1, \max\{A.y_1, B.y_1\}, B.x_1, \min\{A.y_2, B.y_2\})$;
6. Če je $B.x_2 < A.x_2$, dodaj v Sezn pravokotnik $(B.x_2, \max\{A.y_1, B.y_1\}, A.x_2, \min\{A.y_2, B.y_2\})$;
7. Vrni Sezn.

algoritem Odreži
vhod: seznam pravokotnikov Sezn in pravokotnik A
izhod: seznam pravokotnikov, ki jih dobimo, ko odrežemo A od vsakega pravokotnika iz seznama Sezn
lokalna spremenljivka: seznam pravokotnikov Sezn2

1. Sezn2 := $\langle\rangle$;
2. ponavljaj za vse pravokotnike B iz seznama Sezn:
   dodaj seznamu Sezn2 seznam OdrežiDvaPravokotnika(B, A).
3. Vrni Sezn2.

algoritem OsveziOkno
vhod: okno W
izhod: osveži vsebino okna W na zaslonu
lokalni spremenljivki: seznam pravokotnikov SeznP, okno V

1. SeznP := ⟨W.pravokotnik⟩; V := W.predhodnik;
2. ponavljaj, dokler je V ≠ **nil**:
   SeznP := Odreži(SeznP, V.pravokotnik);
   V := V.predhodnik;
3. za vsak pravokotnik P iz seznama SeznP izvedi
   Nariši(W, P);