22. državno tekmovanje v znanju računalništva (1998)

NALOGE ZA PRVO SKUPINO

1998.1.1

Kaj izpiše naslednji **program**? Odgovor primerno utemelji!

Rešitev: str. 8

```
program Ego;
const
  S: array [1..21] of string = (
    'program Ego;',
    'const',
    '  S: array[1..21] of string = (',
    '  );',
    'var',
    '  i, j: integer;',
    'begin',
    '  for i := 1 to 3 do',
    '    WriteLn(S[i]);',
    '  for i := 1 to 21 do begin',
    '    Write("    """, S[i], """");',
    '    if i < 21 then Write(",");',
    '    WriteLn;',
    '  end;',
    '  for i := 4 to 21 do begin',
    '    for j := 1 to Length(S[i]) do',
    '      if (i <> 18) or (j <> 23) then',
    '        if S[i, j] = """ then S[i, j] := """";',
    '    WriteLn(S[i]);',
    '  end;',
    'end.'
  );
var
  i, j: integer;
begin
  for i := 1 to 3 do
    WriteLn(S[i]);
  for i := 1 to 21 do begin
    Write('    ''', S[i], '''');
    if i < 21 then Write(',');
    WriteLn;
  end;
  for i := 4 to 21 do begin
    for j := 1 to Length(S[i]) do
      if (i <> 18) or (j <> 23) then
        if S[i, j] = '"' then S[i, j] := '''';
    WriteLn(S[i]);
  end;
end.
```

(*Opomba:* ni mišljeno, da bi tekmovalec odgovoril, da program ne izpiše ničesar, ker da se sploh ne prevede, saj poskuša občasno spreminjati vsebino tabele S, ta pa je deklarirana kot konstanta. V Turbo Pascalu konstante, ki so imele ob deklaraciji eksplicitno naveden tip, niso bile nič drugega kot spremenljivke z vnaprej določeno začetno vrednostjo, kasneje pa se jih je dalo spreminjati kot običajne spremenljivke.)

Rešitev: str. 9

1998.1.2

V urejevalniku besedil WORD BUSTER imamo neko besedilo, ki ga želimo spremeniti, kot je zahtevano na koncu naloge.

Urejevalnik pozna pojem *trenutnega položaja* (nahajališče kazalca oziroma kurzorja). Če je v vrstici n znakov, je trenutni položaj vedno med 1 in $n+1$: bolj desno od $(n+1)$-vega znaka trenutni položaj ne more biti, prav tako ne bolj levo od začetka vrstice. V spodnjih primerih je trenutni položaj označen s črtico pod ustreznim znakom, npr. `takole`.

Na razpolago imaš naslednje ukaze urejevalnika:

`BRISI_KONEC` — zbriše znake v vrstici od vključno znaka, na katerem se trenutno nahajamo, do konca vrstice; zbrisane znake shrani v začasni pomnilnik. Trenutni položaj se ne spremeni.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEF_`

`BRISI_ZACETEK` — zbriše znake v vrstici pred trenutnim položajem; znaki od vključno trenutnega položaja do konca vrstice se premaknejo na začetek vrstice, novi trenutni položaj je 1. Zbrisani znaki se shranijo v začasni pomnilnik.

Primer: `ABCDEFGHIJKLMN` ⟶ `GHIJKLMN`

`BRISI_ZNAK` — zbriše znak na trenutnem položaju, če nismo na koncu vrstice. Znaki od trenutnega položaja do konca vrstice se premaknejo za en znak v levo; zbrisani znak se shrani v začasni pomnilnik. Trenutni položaj se ne spremeni.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEFHIJKLMN`

`VRINI_ZBRISANO` — na trenutno mesto vrine znake iz začasnega pomnilnika, ki jih je tja shranil zadnji od ukazov `BRISI_ZACETEK` ali `BRISI_KONEC` ali `BRISI_ZNAK`. Znaki v prvotni vrstici od vključno znaka, na katerem se trenutno nahajamo, pa do konca vrstice, se premaknejo v desno za število vrinjenih znakov. Trenutni položaj se premakne skupaj s preostankom vrstice.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEFshranjenoGHIJKLMN`

`VRINI_PRESLEDEK` — na trenutno mesto vrine en presledek; trenutni položaj se premakne za eno mesto v desno skupaj s preostankom vrstice.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEF GHIJKLMN`

`DESNO` — premakne trenutni položaj za eno mesto v desno. Če se že nahajamo tik za zadnjim znakom v vrstici, se položaj ne spremeni.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEFGHIJKLMN`

`LEVO` — premakne trenutni položaj za eno mesto v levo. Če se že nahajamo na prvem mestu v vrstici, se položaj ne spremeni.

Primer: `ABCDEFGHIJKLMN` ⟶ `ABCDEFGHIJKLMN`

`NA_ZACETEK_NASLEDNJE_BESEDE` — premakne trenutni položaj na začetek naslednje besede ali na konec vrstice, če smo že pri zadnji besedi. Beseda je definirana kot strnjeno zaporedje znakov, ki niso presledki.

Primeri:
```
ABC DEFGHIJ   KLMN ⟶ ABC DEFGHIJ   KLMN
ABC DEFGHIJ   KLMN ⟶ ABC DEFGHIJ   KLMN
ABC DEFGHIJ _ KLMN ⟶ ABC DEFGHIJ   KLMN
```

`NA_KONEC_PREJSNJE_BESEDE` – premakne trenutni položaj na zadnji znak prejšnje besede, ali na začetek vrstice, če smo že pri prvi besedi.

Primeri:
```
ABC DEFGHIJ   KLMN ⟶ ABC DEFGHIJ   KLMN
ABC DEFGHIJ   KLMN ⟶ ABC DEFGHIJ   KLMN
ABC DEFGHIJ _ KLMN ⟶ ABC DEFGHIJ   KLMN
```

Pred vsakim ukazom lahko stoji število ponovitev. Na primer: „`10 DESNO`“ pomeni, naj se ukaz `DESNO` izvrši desetkrat. Za vse ukaze velja, da ne storijo nič, če svoje naloge ne morejo opraviti (npr. brisanje prazne vrstice, pomik levo od prvega znaka v vrstici in podobno).

V začetku je trenutni položaj na začetku vrstice, po končanem opravilu je lahko trenutni položaj kjerkoli v vrstici.

Primer: naslednje zaporedje ukazov prestavi prvo besedo v vrstici na konec vrstice:

```
NA_ZACETEK_NASLEDNJE_BESEDE, BRISI_ZACETEK, 1000 DESNO,
VRINI_PRESLEDEK, VRINI_ZBRISANO
```

Reši naslednji nalogi:

(a) V urejevalniku je besedilo, ki ga želimo desno poravnati tako, da na začetek vrstic vrinemo ustrezno število presledkov (vsi znaki so enako široki). Nobena vrstica danega besedila ni daljša od 70 znakov in ne vsebuje presledkov niti na začetku niti na koncu vrstic. Končno poravnano besedilo naj bo široko 80 znakov. **Napiši zaporedje ukazov**, ki trenutno vrstico prestavi (poravna) desno.

(b) V vsaki vrstici je zapisano neko število med nič in dvajset milijoni. **Napiši zaporedje ukazov**, ki število v trenutni vrstici naredi bolj čitljivo tako, da vrine presledek na vsaka tri mesta, kot kažejo primeri:

```
12345000  ⟶  12 345 000
567890    ⟶  567 890
43        ⟶  43
```

1998.1.3

Rešitev: str. 10

Podjetja vsako leto zaslužijo nekaj denarja. Ob koncu leta, ko je obračun, morajo prikazati dobiček. Zaradi hecnih zakonov se jim ne splača prikazati dobička, ki je večji kot 1000 cekinov. Naše podjetje *„Hlevi softwearskih ljubiteljev“* (HSL) si vsako leto izplača največji dobiček, ki ne presega 1000 cekinov, preostanek denarja pa prenese v naslednje leto. **Napiši program**, ki prebere, koliko denarja je zaslužilo podjetje, izpiše pa naj dobiček ob koncu leta in koliko denarja se je preneslo v naslednje leto. Pri naslednjem letu seveda upoštevaj tudi denar, ki se je prenesel iz prejšnjega leta. Primer: Če imamo na vhodu naslednje zneske zaslužkov (brez komentarjev):

`300`	prvo leto, začetek firme
`800`	drugo leto, vzpon
`1200`	tretje leto, vse cveti
`1500`	četrto leto, nič nas ne more ustaviti
`400`	peto leto, ops!
`100`	šesto leto, leto suhih krav
`50`	sedmo leto, životarjenje
`500`	osmo leto, morda pa le ni tako slabo
`1500`	deveto leto, povratek odpisanih

mora program izpisati naslednje zneske (brez komentarjev):

`300 0`	dobiček je 300, nič prenosa v naslednje leto
`800 0`	dobiček je 800, nič prenosa v naslednje leto
`1000 200`	dobiček je 1000 ($= 1200 - 200$), prenos je 200
`1000 700`	dobiček je 1000 ($= 1500 + 200 - 700$), prenos je 700
`1000 100`	dobiček je 1000 ($= 400 + 700 - 100$), prenos je 100
`200 0`	dobiček je 200 ($= 100 + 100$), prenosa ni
`50 0`	dobiček je 50, prenosa ni
`500 0`	dobiček je 500, prenosa ni
`1000 500`	dobiček je 1000 ($= 1500 - 500$), prenos je 500

Rešitev: str. 11

1998.1.4

Imamo merilno napravo, ki šteje dežne kapljice, ki padejo na merilno ploskev. S podprogramsko funkcijo StDeznihKapelj lahko odčitamo, koliko dežnih kapelj je padlo od prejšnjega klica te funkcije. Kadar le rahlo rosi, bi radi, da računalnik enkrat zapiska za vsako padlo kapljico in to takoj, ko jo merilna naprava zazna. Da ne bi bilo v nalivu piskanja preveč, postavimo dodatne pogoje:

- med dvema zaporednima piskoma mora miniti vsaj ena sekunda;
- če zaradi prejšnjega pogoja ni dovoljeno zapiskati takoj, lahko pisk zamuja, vendar ne več kot za 5 sekund;
- višek dežnih kapljic, ki jih zaradi gornjih dveh omejitev ni mogoče javiti, tiho spregledamo.

Napiši program, ki ob upoštevanju danih omejitev odčitava merilno napravo in piska. Pri tem naj ne troši procesorskega časa po nepotrebnem. Na voljo imaš naslednje podprograme:

function StDeznihKapelj(CakajNajvec: integer): integer;
: Ta podprogramska funkcija odčita število dežnih kapelj, padlih na merilno napravo od prejšnjega klica, in to število vrne kot funkcijsko vrednost. Podprogram se vrne takoj, če je na zalogi kakšna še neprešteta kaplja, ali pa takrat, ko nova kaplja pade — vendar na kapljo ne čaka dlje kot CakajNajvec tisočink sekunde (CakajNajvec je lahko tudi 0). Če je CakajNajvec negativno število, čakanje ni časovno omejeno.

procedure Zapiskaj;
: Sproži en pisk.

procedure Zaspi(Cakaj: integer);
Podprogram zaspi za Cakaj tisočink sekunde (v tem času lahko procesor izvaja druge programe) in se po tem času vrne.

NALOGE ZA DRUGO SKUPINO

1998.2.1

Rešitev: str. 12

Podane so celoštevilske koordinate n točk, ki ležijo v ravnini. Vsak par teh točk določa pravokotnik, ki ima s koordinatnima osema vzporedne stranice in ena točka leži v njegovem spodnjem levem oglišču, druga pa v zgornjem desnem oglišču.[1] **Napiši program**, ki učinkovito poišče število takih pravokotnikov, ki so kvadrati. Štej tudi izrojene kvadrate, pri katerih spodnje levo in zgornje desno oglišče sovpadata.

Število točk n je manjše od 10^5.

Območje, na katerem ležijo točke v ravnini, je znotraj kvadrata:

$$-10\,000 \leq x \leq 10\,000, \qquad -10\,000 \leq y \leq 10\,000$$

Na voljo imaš največ 1 234 567 zlogov (bytov) pomnilnika.

1998.2.2

Rešitev: str. 13

Kriptografija je veda, ki služi marsičemu — še najbolj jo poznamo iz filmov in knjig o dogajanjih med vojno ali o kakšnih tajnih agentih ali o čem podobnem. Prav nam pa pride lahko tudi v čisto vsakdanjem življenju, pri reševanju medsosedskih problemov. Predstavljajmo si na primer ulico z n ($n > 2$) sosedi, ki sicer vedno govorijo resnico, nočejo pa drug drugemu izdati, kakšno plačo ima kateri od njih. Ker bi se radi primerjali s prebivalci iz sosednje ulice, pa morajo vseeno na nek način izvedeti, koliko skupaj zaslužijo na mesec — pri tem seveda nočejo kršiti prej omenjene kaprice (da bi komurkoli izdali višino svoje plače). Razmislite in **opišite postopek** oz. postopke, kako bi to sosedi med seboj izvedli. Opišite tudi prednosti in slabosti naštetih postopkov.

1998.2.3

Rešitev: str. 13

Danih imate n pravokotnikov, podanih s koordinatami levega spodnjega in desnega zgornjega oglišča $\langle(x_1, y_1), (x_2, y_2)\rangle$. Koordinate so vse celoštevilske (tipa integer). **Opiši postopek**, ki izračuna skupno ploščino, ki jo prekrivajo pravokotniki. Pri tem seveda ne pozabite upoštevati, da se pravokotniki lahko tudi prekrivajo — v takih primerih prekrite ploščine ne smemo šteti dvo- ali večkratno. Program naj izračuna skupno ploščino s čim manj izvedenimi operacijami in s čim manj uporabljenega pomnilnika.

Podatke o pravokotnikih dobite z dvema pomožnima funkcijama:

function StPravok: integer;
Vrne n, število vseh pravokotnikov.

[1] Kaj pa, če imamo na primer točki $(0, 1)$ in $(1, 0)$? Tidve ležita v zgornjem levem in spodnjem desnem oglišču svojega pravokotnika. Gornje besedilo iz leta 1998 torej, če ga beremo dobesedno, obljublja, da se v vhodnih podatkih to ne bo zgodilo; na primer, če imamo točki (x, y) in (x', y'), pa je $x < x'$, bo gotovo veljalo tudi $y \leq y'$. Verjetnejša razlaga pa je, da so sestavljalci naloge pozabili omeniti, naj tekmovalčeva rešitev tiste pare točk (x, y) in (x', y'), za katere je $x < x'$ in $y > y'$, pri štetju kvadratov pač ignorira.

function DajKoord(i: integer; **var** x1, y1, x2, y2: integer);
V spremenljivke x1, y1, x2, y2 vrne koordinate i-tega pravokotnika. Spremenljivki x1 in y1 predstavljata levo zgornje, x2 in y2 pa desno spodnje oglišče.

Rešitev: str. 19

1998.2.4

V vsaki vrstici vhodne datoteke je zapisano eno ime (niz znakov) nekega računalnika v internetu; imena se ne ponavljajo. Program mora brati imena, za vsako ime poiskati njegov omrežni naslov (niz znakov), nato pa dobljene naslove zapisati v izhodno datoteko v nespremenjenem vrstnem redu.

Trajanje iskanja imenu pripadajočega naslova je zelo različno; večino časa pri tem izgubimo s čakanjem na tuje računalnike v omrežju. Zagotovljeno je, da dobimo podatek najkasneje v nekem določenem končnem času (v ilustracijo: podatek dobimo v 1 do 100 sekundah).

Da pospešimo delo, smo razvili paralelni del programa, ki je sposoben hkrati iskati do 64 naslovov, rezultate pa vrača takoj, ko so na voljo, torej v vrstnem redu, ki večinoma ni enak vrstnemu redu zastavljenih vprašanj.

Napiši postopek, ki bo bral imena, dodeljeval delo prostim paralelnim vejam programa, od njih pobiral rezultate in jih tako sproščal za novo nalogo ter rezultate takoj, ko bo možno, izpisoval v enakem vrstnem redu, kot so bili prebrani podatki (imena računalnikov). Upoštevaj, da je lahko vhodnih podatkov poljubno mnogo in da imaš na razpolago omejen pomnilnik, ki lahko hrani največ 1000 nizov (imen ali naslovov računalnikov). **Pojasni tudi**, kako si organiziral podatkovno strukturo, potrebno za urejanje rezultatov.

Na voljo imaš naslednja podprograma:

function Obdelaj(Ime: string): integer;
Podprogram poišče eno od 64 prostih vej programa in ji naloži novo opravilo (preslikavo imena v naslov). Podprogramska funkcija se vrne takoj in kot funkcijsko vrednost vrne zaporedno številko veje, ki je prevzela opravilo (število med 1 in 64). Če nobena veja ni prosta, vrne 0, opravila pa si ne zapomni.

function PoberiRezultat(**var** Naslov: string): integer;
Podprogram poišče eno od vej programa, ki je že opravila svojo nalogo (po potrebi počaka na prvo tako), ter vrne najdeni naslov, kot funkcijsko vrednost pa vrne številko veje, iz katere je prišel rezultat. S prevzemom rezultata se ta veja sprosti in čaka na novo opravilo.

NALOGE ZA TRETJO SKUPINO

Rešitev: str. 20

1998.3.1

Dan je načrt mestnih ulic — v bistvu graf, podan s seznami sosednjih križišč za posamezno križišče. V teh seznamih je vrstni red pomemben — sledijo si v smeri urinega kazalca. V posameznem križišču se srečajo tri ali štiri ulice (točke, kjer se srečata samo dve ulici, ne imenujemo križišče). Znane so tudi dolžine ulic. Mestni očetje so sprejeli odlok, da je odslej prepovedano zavijati na levo.

Sestavi oz. **opiši postopek**, ki bo poiskal najkrajšo pot, ki upošteva odlok, iz danega križišča v drugo dano križišče (oziroma ugotovil, da ta ne obstaja).

Lahko predpostavite, da imate že dano funkcijo, ki poišče najkrajšo pot med dvema križiščema, ki pa ne upošteva odloka. Ta funkcija obvlada tudi mestne načrte z enosmernimi ulicami, čeprav takih ulic v prvotnem mestnem načrtu ni. Naloga je torej lahko tudi: kako predelati dani načrt mestnih ulic tako, da boste lahko uporabili dano funkcijo in s tem odgovorili na osnovno vprašanje.

1998.3.2

Rešitev: str. 21

Pri podjetju Paranoid d. o. o. so izdelali elektronsko ključavnico, ki jo je moč odkleniti le s posebnim elektronskim ključem. Pri odklepanju preda ključavnica ključu vprašanje v obliki poljubnega 16-bitnega števila, na osnovi katerega ključ izračuna pripadajoč 128-bitni odgovor in ga vrne ključavnici. Če odgovor ustreza vprašanju, ključ odklene ključavnico.

Jedro elektronskega ključa je procesor Merlin, ki teče s taktom 571 MHz, v enem ciklu izvede en ukaz in ima poleg 16-bitnega programskega števca še dva 8-bitna registra. Poleg procesorja je v elektronskem ključu le še bralni pomnilnik (ROM), ki vsebuje program za izračun odgovorov.

Na računalnik imaš priklopljeno testno ključavnico, v katero lahko vložiš ključ. **Opiši postopek**, ki ugotovi, ali se program za izračun odgovorov pri poljubnem vprašanju vedno ustavi, na voljo pa imaš naslednji dve funkciji:

- **procedure** Vprasaj(Vprasanje: integer); — Preda ključu 16-bitno vprašanje. Ključ ob tem „resetira“ — če je dotlej kaj računal, ga prekine in ključ se začne takoj ukvarjati z novim vprašanjem.
- **function** Odgovor(**var** Odg: **array** [1..16] **of** byte): boolean — Vrne false, če ključ še računa, sicer pa vrne true in v Odg ključev odgovor na zadnje vprašanje.

1998.3.3

Rešitev: str. 21

Med n različnimi števili, zapisanimi v urejeni tabeli, bi radi poiskali tak par števil, da bo njuna vsota enaka 100. V spodnjem zaporedju je primeren par $10 + 90$.

2 7 10 14 24 31 35 44 45 60 67 69 75 90 95

Kako se lotiti dela? Če bi slepo preizkušali vse možne pare, jih seštevali in preverjali njihove vsote, bi za n števil potrebovali $\frac{1}{2}n(n-1)$ seštevanj — že pri tisoč številih bi bilo seštevanj skoraj pol milijona, pri milijon številih pa že pol bilijona...

Gre hitreje? Za začetek poskusi kar na roko poiskati še kak par z vsoto 100 v gornji tabeli, nato pa izumljeni postopek zapiši v obliki podprograma IzpisiPare(a: **array** [1..n] **of** integer), ki izpiše vse primerne pare. Če ti uspe nalogo rešiti z manj kot n seštevanji, si na pravi poti.

1998.3.4

Rešitev: str. 22

Ko je bila Valentina še majhna, se je rada igrala z binarnimi drevesi. Tvorila je drevesa s samimi različnimi velikimi črkami v vozliščih. Tole je primer takšnega drevesa:

```
      D
     / \
    B   E
   / \   \
  A   C   F
```

(*Opomba:* pri drevesih te vrste velja tudi v primerih, ko ima neko vozlišče enega samega otroka ali poddrevo, da je ta otrok ali levi ali pa desni. V gornjem drevesu je na primer `F` desni otrok vozlišča `E` in če bi bil `F` namesto tega levi otrok, to že ne bi bilo več enako drevo.)

Da bi ta drevesa shranila za kasneje, je za vsako drevo zapisala dva niza. Prvega je zapisala kot: levo poddrevo, koren, desno poddrevo. Drugega pa je zapisala po nivojih (najprej prvi nivo, nato drugi, ..., vsak nivo pa od leve proti desni). Za zgornji primer je dobila `ABCDEF` in `DBEACF`.

Valentina je upala, da bo iz takih parov nizov lahko rekonstruirala svoja drevesa. Ko je čez leta to želela storiti, je ugotovila, da je to možno, vendar le zato, ker je vsako drevo zapisala z dvema nizoma. Bilo pa je to delo precej zamudno. Zato te prosi, da bi ji napisal program, ki bi ji pri tem pomagal.[2]

Program naj prebere oba niza (kot smo ju opisali zgoraj), drevo pa naj izpiše kot niz oblike: levo poddrevo, desno poddrevo, koren. Naš zgornji primer bi torej zapisali `ACBFED`.

REŠITVE NALOG ZA PRVO SKUPINO

Naloga: str. 1

R1998.1.1

Program izpiše samega sebe.

V konstanti S je po en niz za vsako vrstico programa, razen za tiste, ki definirajo sámo konstanto S. Program najprej izpiše vrstice do začetka definicije konstante S, nato uporabi nize iz tabele S, da izpiše še vrstice, ki definirajo S (vsak niz mora oviti v narekovaje in dodati še zamik na začetku vrstic), nato pa lahko izpiše še preostanek programa. Tam, kjer so v programu enojni narekovaji (znak `'`), vsebujejo nizi v tabeli S dvojne narekovaje (znak `"`), pri izpisovanju pa jih zamenjamo z enojnimi. Izjema je eden, ki mora vendarle ostati (23. znak v S[18]).

Program, ki izpiše samega sebe, je običajno sestavljen iz dveh delov, A in B; A definira neko konstanto S, ki vsebuje kodo dela B; B pa izpiše najprej A (pri čemer si lahko pomaga z vsebino konstante S) in nato še samega sebe (tu pa v bistvu preprosto izpiše S).

Ostane le še vprašanje, kako A in B sestaviti in povezati skupaj v program, ki res spoštuje vsa pravila izbranega programskega jezika in je njegov izpis res natančno enak njegovi izvorni kodi. Ta del problema so v bistvu le tehnikalije, odvisne od sintakse in drugih pravil našega programskega jezika.

V našem primeru je A sestavljen iz vrstic 4 do 24, torej tistih z vsebino tabele S, ostalo pa je B. Tehnikalije so v pascalu povezane predvsem z narekovaji. Če B vsebuje kaj enojnih narekovajev, jih mora A vsebovati podvojene (na primer: niz `x'y` moramo v pascalski izvorni kodi zapisati kot `'x''y'`), vendar pa jih bo B v konstanti S potem videl le enojno. Zato mora B, ko izpisuje del A, vsak enojni narekovaj izpisati dvakrat. Možne so še druge rešitve; A bi lahko v konstanti S na mestih, kjer B vsebuje enojne narekovaje, definiral kak drug znak, ki se drugače v kodi ne pojavlja, npr. `"`. B bi moral potem pri izpisovanju dela A vse znake `"` zamenjati z enojnimi narekovaji; seveda pa s tem B-jeva koda že vsebuje en znak `"` (da lahko z njim primerja vsebino konstante S) in tega pri izpisovanju A-ja ne sme spremeniti v `'`; v ta namen bi lahko na primer pazil, v kateri vrstici in stolpcu se ta `"` pojavlja in tistega pač ne bi spremenil v `'` (pri programu v nalogi smo se odločili prav za to rešitev). Lahko pa bi B uporabil tudi ASCII kodo znaka `"`, tako da bi namesto znaka `"` vseboval B le nedolžni niz Chr(34).

[2] To je naloga H z lokalnega ACMovega študentskega tekmovanja univerze v Ulmu v programiranju (Ulm, 1997); #536 v zbirki na `online-judge.uva.es`.

Toda isti prijem bi lahko uporabili že na samem začetku; sploh ni nujno, da se v B-ju pojavlja znak `'`, saj ga lahko nadomestimo s `Chr(39)`, ko bomo izpisovali narekovaje v definiciji konstante `S`; za izpis presledkov pa uporabimo `Chr(32)`, da jih ne bo treba v izvorni kodi dela B navajati v narekovajih kot `' '`. Vendar je prijem s kodami ASCII po svoje manj eleganten, ker mora predpostaviti, da našega programa ne bodo uporabljali na računalniku s kakšnim zelo eksotičnim naborom znakov.

Če ima programski jezik ugodne lastnosti, je pisanje programov, ki izpišejo sami sebe, lahko precej preprostejše. V jeziku C smo lahko s pomočjo funkcije `printf` malo bolj jedrnati:

```
#include <stdio.h>
char*s="#include <stdio.h>%cchar*s=%c%s%c,%c*t=%c%s%c,%c*u=%c%s%c;%c",
*t="int main(){int n='%cn',q='%c%c',b='%c%c';%c   %sreturn 0;}%c",
*u="printf(s,n,q,s,q,n,q,t,q,n,q,u,q,n);printf(t,b,b,q,b,b,n,u,n);";
int main(){int n='\n',q='\"',b='\\';
   printf(s,n,q,s,q,n,q,t,q,n,q,u,q,n);printf(t,b,b,q,b,b,n,u,n);return 0;}
```

Delu A v tem primeru ustrezajo nizi `s`, `t` in `u`, ostalo pa je del B. S pomočjo spremenljivk `n`, `q` in `b` se lahko izognemo potrebi po tem, da bi se v morali v nizih `s`, `t` in `u` pojavljati nadležni znaki, zaradi katerih bi bila vsebina takega niza med tekom programa drugačna od zaporedja znakov v izvorni kodi, s katero je bil niz definiran.

Python pa ima funkcijo `repr(x)`, ki vrne[3] kot niz kar košček izvorne kode, s kakršnim bi bilo treba v programu inicializirati neko spremenljivko, da bi dobila vrednost `x`. Zato so nam prihranjene manipulacije z narekovaji in samega sebe izpiše že naslednji enovrstični program (ki pa ima, kot lahko opazimo, zelo podobno strukturo kot gornji pascalski program):

```
s = 's = %s; print s %% repr(s)'; print s % repr(s)
```

Do skrajnosti gredo v to smer kakšne tradicionalne oblike BASICa:

```
10 LIST
```

Še en razvpit primer pa je popolnoma prazen program (ki, odkrito povedano, res ne izpiše ničesar), ki je dobil leta 1994 na tekmovanju obfuscated C nagrado za najhujšo zlorabo pravil. :-)

Na spletu je precej zanimivih strani o programih, ki izpišejo samega sebe (v angleščini se takemu programu pogosto pravi *quine*, v spomin na logika Willarda Quinea). S podobnim razmislekom kot pri takih programih lahko sestavimo tudi na primer dva programa, ki izpišeta drug drugega, ipd.

R1998.1.2

Naloga: str. 2

(a) Na začetek vrstice najprej vrinimo 80 presledkov in tako poskrbimo, da bo cela vrstica vsebovala vsaj 80 znakov. Če potem obdržimo le zadnjih 80 znakov, bo to celotna vsebina prvotne vrstice in še toliko presledkov na začetku, da bo vrstica desno poravnana.

```
80 VRINI_PRESLEDEK, 70 DESNO, 80 LEVO, BRISI_ZACETEK
```

[3] Vsaj pri standardnih tipih; razredi, ki jih napiše uporabnik, lahko sami določijo, kaj naj vrne `repr` na primerkih tega razreda.

Nekateri urejevalniki imajo omejitve glede največjega dovoljenega števila znakov v vrstici. Gornja rešitev bi lahko v eni vrstici dobila do 150 znakov (največ 70 od prej in še 80 vrinjenih presledkov). Recimo, da naš urejevalnik ne dovoli toliko znakov in bi pri vrivanju presledkov na začetku vrstice začel rezati znake s konca vrstice, če bi le-ta postala predolga. V tem primeru je bolje vrivati presledke na koncu in jih nato premakniti na začetek. Recimo, da je vrstica na začetku dolga n znakov. Če vrinemo na koncu 80 presledkov, bo rezultat dolg vsaj 80 znakov; in če zdaj zbrišemo vse razen prvih 80 znakov, bomo dobili kar prvotno vrstico, podaljšano za $80 - n$ presledkov. Če zdaj te presledke premaknemo na začetek vrstice, bomo dobili ravno to, kar iščemo: prvotno vrstico, poravnano desno na dolžino 80.

```
70 DESNO, 80 VRINI_PRESLEDEK, 150 LEVO, 80 DESNO,
BRISI_KONEC, NA_KONEC_PREJSNJE_BESEDE, DESNO,
BRISI_KONEC, 70 LEVO, VRINI_ZBRISANO
```

(b) Premaknimo se na konec vrstice, nato pa za tri mesta v levo in vrinimo presledek; nato za štiri mesta v levo (mimo pravkar vrinjenega presledka in še naslednjih treh števk) in spet vrinimo presledek. To je že dovolj, če je število v tej vrstici res največ dvajset milijonov (saj ima tako le osem števk in ni treba vriniti več kot dveh presledkov); vendar pa smo v primeru, ko je število krajše, zdaj mogoče vrinili presledek ali dva preveč. Zato je pametno iti na začetek vrstice (kar zahteva še največ tri premike v levo) in nato pobrisati vse do začetka prve besede — s tem se znebimo morebitnih odvečnih presledkov. Toda „pobrisati vse do začetka prve besede“ ni čisto trivialno: če pred prvo besedo (v našem primeru: pred prvim sklopom števk) ni nobenega presledka, bi bili na začetku vrstice obenem že tudi na začetku prve besede in ukaz `NA_ZACETEK_NASLEDNJE_BESEDE` bi nas prestavil na začetek *druge* besede. Zato je bolje pred tem vriniti na začetek vrstice še en presledek, se postaviti nanj in nato skočiti na začetek naslednje besede (kar bo zdaj zagotovo res pomenilo na začetek prve besede).

```
8 DESNO,
3 LEVO, VRINI_PRESLEDEK,
4 LEVO, VRINI_PRESLEDEK,
3 LEVO, VRINI_PRESLEDEK, LEVO,
NA_ZACETEK_NASLEDNJE_BESEDE, BRISI_ZACETEK
```

Naloga: str. 3

R1998.1.3

Program mora le slediti navodilom naloge. Prenos iz prejšnjega leta hranimo v spremenljivki Prenos (na začetku dobi vrednost 0), nato pa vsako leto prištejemo dobiček tistega leta in tisto, kar sega čez 1000, razglasimo za prenos v prihodnje leto.

```
program UstvarjalnoKnjigovodstvo;
var Prenos, Dobicek: integer;
begin
  Prenos := 0;
  while not Eof do begin
    ReadLn(Dobicek);
    Prenos := Prenos + Dobicek;
    if Prenos > 1000 then Dobicek := 1000 else Dobicek := Prenos;
    Prenos := Prenos − Dobicek;
```

```
    WriteLn(Dobicek, ' ', Prenos);
  end; {while}
end. {UstvarjalnoKnjigovodstvo}
```

R1998.1.4

Naloga: str. 4

V spremenljivki NaZalogi bomo hranili število dežnih kapelj, ki smo jih že odčitali z merilne naprave, vendar zanje še nismo zapiskali (niti se nismo odločili, da jih bomo ignorirali). Če je NaZalogi $=$ 0, lahko funkciji StDeznihKapelj naročimo, naj čaka na naslednjo kapljo, sicer pa ji naročimo, naj se vrne takoj in s tem le preverimo, če je padlo od zadnjega klica še kaj novih kapelj. Nato, če je NaZalogi večja od 0, jo zmanjšamo za 1, zapiskamo in počakamo eno sekundo.

Zahtevo, naj kapljice, ki bi morale na svoj pisk čakati več kot pet sekund, tiho ignoriramo, lahko upoštevamo na naslednji način. Recimo, da bi vsako kapljico, za katero na novo izvemo ob trenutnem klicu funkcije StDeznihKapelj, dodali na nek seznam (pravzaprav vrsto), kjer bi čakala, da izvedemo pisk zanjo. Če je kapljica v trenutku, ko jo na seznam dodamo, n-ta po vrsti, bo njen pisk prišel na vrsto čez $n-1$ sekund (za prvo kapljico bomo zapiskali takoj, za drugo čez eno sekundo in tako naprej). In ker smo jo na seznam ravnokar dodali, vemo, da je padla nekje med predzadnjim in zadnjim klicem funkcije StDeznihKapelj, torej pred največ eno sekundo, saj lahko med dvema klicema te funkcije v našem programu mine največ ena sekunda (če smo po prejšnjem klicu zapiskali in nato zaspali za toliko časa). Torej, če je $n \leq 5$, bo prišel njen pisk na vrsto čez največ štiri sekunde, padla pa je pred največ eno sekundo, torej pisk še ne bo prepozen; če pa je $n > 5$, bo prišel pisk na vrsto čez pet ali več sekund, torej bo v vsakem primeru prepozen (tudi pri $n = 6$, saj mora od padca do trenutka, ko bomo mi res izvedli prvi pisk, tudi miniti vsaj neka majhna količina časa, kar bo skupaj s petimi sekundami do šestega piska res pomenilo strogo več kot pet sekund od padca te kapljice do njenega piska). Torej lahko iz seznama vržemo vse kapljice razen prvih petih. No, ker pa naš program ene kapljice prav nič ne razlikuje od druge, zanje ni treba res imeti seznama, ampak je dovolj, če si zapomnimo, koliko bi jih v tem seznamu bilo; ravno to pa nam pove spremenljivka NaZalogi. To moramo torej zmanjšati na 5, če je po klicu StDeznihKapelj zrastla čez 5.

```
program Dez;
var NaZalogi: integer;
begin
  NaZalogi := 0;
  while true do begin
    if NaZalogi > 0
      then NaZalogi := NaZalogi + StDeznihKapelj(0)
      else NaZalogi := NaZalogi + StDeznihKapelj(−1);
    if NaZalogi > 5 then NaZalogi := 5;
    if NaZalogi > 0 then
      begin Zapiskaj; NaZalogi := NaZalogi − 1; Zaspi(1000) end;
  end; {while}
end. {Dez}
```

REŠITVE NALOG ZA DRUGO SKUPINO

Naloga: str. 5

R1998.2.1

Naj bosta (x, y) in (x', y') dve točki, ki ležita v spodnjem levem in zgornjem desnem oglišču nekega kvadrata s stranico a. Potem mora veljati $x' = x+a$ in $y' = y+a$, torej $x'-y' = (x+a)-(y+a) = x-y$. Taki dve točki torej prepoznamo po tem, da imata obe enako razliko med x- in y-koordinato; z drugimi besedami, če bi skozi vsako točko poslali premico, ki oklepa z absciso kot 45°, bi bila to obakrat ena in ista premica.

Zato lahko ravnamo takole: pripravimo si tabelo $a[-20\,000..20\,000]$, katere elementi bodo cela števila; na začetku postavimo vse elemente na nič. Z elementom $a[i]$ bomo šteli, koliko točk leži na premici $x - y = i$. Imejmo še nek števec, ki pove, koliko kvadratov smo že našli (recimo mu S). Potem za vsako točko (x, y) izračunamo $x - y$ in vemo, da smo doslej odkrili $a[x - y]$ točk z enako razliko med koordinatama in da naša nova točka tvori po en kvadrat z vsako od njih. (Seveda lahko naša nova točka tvori kvadrate tudi s kakšnimi točkami, ki jih bomo prebrali šele kasneje, ampak tiste kvadrate bomo pač šteli takrat, pri tistih točkah, tako da nam ni treba skrbeti, da bi kakšnega spregledali.) Torej povečajmo S za $a[x - y]$, nato pa povečajmo $a[x - y]$ za 1, da v mislih dodamo pravkar prebrano točko med tiste s takšno razliko koordinat.

Na koncu bo S ravno število iskanih kvadratov. Lepo pri tem postopku je tudi to, da nam ni treba v pomnilniku hraniti koordinat vseh točk. Količina porabljenega časa je $O(n)$ — premo sorazmerna s številom točk (če predpostavimo, da je zaloga vrednosti naših koordinat, torej v našem primeru od 0 do 20 000, vnaprej določena in omejena).

```
const StTock = ...;
var
  x, y: array [1..StTock] of integer;
  a: array [−20000..20000] of integer;
  j, s: integer;
begin
  ... { preberi koordinate }
  for j := −20000 to 20000 do a[j] := 0;
  s := 0;
  for j := 1 to StTock do begin
    i := x[j] − y[j];
    s := s + a[i];
    a[i] := a[i] + 1;
  end; {for}
  WriteLn('Število kvadratov: ', s);
end.
```

Malo slabša različica te rešitve bi bila, da ne bi sproti povečevali števca S, ampak bi se na koncu sprehodili po vseh elementih tabele a. Vrednost $a[i]$ nam pove, da smo videli $a[i]$ točk z razliko koordinat $x - y = i$; toliko točk pa tvori $a[i] \cdot (a[i] - 1)/2$ kvadratov. To moramo sešteti po vseh i, pa dobimo skupno število vseh kvadratov. Slabost v primerjavi s prejšnjim postopkom je ta, da se moramo zdaj še enkrat sprehajati po celi tabeli a.

Preprostejša, a časovno veliko bolj potratna rešitev pa je, da pogledamo vse pare točk in za vsak par preverimo, ali točki ležita v ogliščih kvadrata (torej: ali je $x'-x = y'-y$). Zdaj moramo imeti koordinate vseh točk v pomnilniku, kar pri gornji rešitvi ni nujno, količina opravljenega dela pa je sorazmerna s kvadratom števila točk (če je točk n, moramo pregledati $n(n-1)/2$ parov točk).

```
const StTock = ...;
var
  x, y: array [1..StTock] of integer;
  i, j, s: integer;
begin
  ... { preberi koordinate }
  s := 0;
  for i := 1 to StTock − 1 do
    for j := i + 1 to StTock do
      if x[j] − x[i] = y[j] − y[i] then
        s := s + 1;
  WriteLn('Število kvadratov: ', s);
end.
```

R1998.2.2

Naloga: str. 5

Sosedje naj si med seboj podajajo kalkulator (ali pa tablo) z dosedanjo vsoto svojih plač. Prvi naj si skrivaj izbere neko veliko naključno število, ga vtipka v kalkulator (in si ga zapomni) ter mu prišteje svojo plačo. Nato poda kalkulator naprej; vsakdo prišteje svojo plačo trenutni vsoti in poda kalkulator spet naslednjemu sosedu. Ko to naredijo vsi, naj prvi spet odšteje tisto svoje začetno število. Ostala bo ravno vsota njihovih plač, pri tem pa nobeden od njih ni točno vedel, kakšna je vsota plač tistih, ki so vnesli svoje plače že pred njim, saj niso vedeli, kakšno je tisto začetno število, ki si ga je izmislil prvi sosed. No, paziti morajo le še na to, da si lahko kalkulator zapomni zadnjo izvedeno operacijo, tako da naj po prištevanju svoje plače mogoče prištejejo še 0 ali izvedejo kakšno drugo nekoristno operacijo.

Morebitna slabost opisanega postopka je, da se lahko dva soseda zarotita proti nekemu tretjemu in izvesta njegovo plačo. Morata le poskrbeti, da se bosta v vrstnem redu računanja znašla tik pred in tik za njim; potem lahko primerjata vrednost, ki je bila v kalkulatorju, preden ga je tisti tretji sosed dobil, in vrednost, ko je dal kalkulator naprej; razlika med njima je ravno njegova plača. Da bi takšne zarote otežili, bi lahko na primer rekli, naj si vsakdo izmisli več čudnih števil, ki se bodo seštela ravno v njegovo plačo. Potem bi si sosedje kalkulator podajali večkrat, po možnosti vsakič v drugačnem in naključno izbranem vrstnem redu. Vsakdo bi, ko bi dobil kalkulator, prištel naslednje izmed svojih števil; tako bi morali sosedje, če bi hoteli izvedeti njegovo plačo, vedeti za vsak krog posebej, kaj je prištel takrat. Zato za zaroto zdaj nista dovolj le dva, ampak bi se je morali udeležiti vsi, ki so bili v kakšnem od krogov tik pred ali tik za njim.

R1998.2.3

Naloga: str. 5

Pomagali si bomo s tehniko „pometanja“ ali preleta ravnine (*plane sweep*). Označimo unijo naših pravokotnikov z U. Spomnimo se, da imajo vsi naši pravokotniki stranice, vzporedne s koordinatnima osema; recimo, da bi ravnino razrezali na vodoravne pasove pri vseh tistih y-koordinatah, kjer ima kateri od naših pravokotnikov svojo zgornjo ali spodnjo stranico. Kot vidimo iz slike na strani 15, velja zdaj za vsakega od nastalih pasov naslednje: če pogledamo katerokoli vodoravno premico znotraj tega pasu, pokriva U na tej premici vedno ene in iste x-koordinate; recimo, da imajo ti intervali pokritih x-koordinat skupno dolžino d, pas, v katerem to opazujemo, pa omejujeta premici $y = c$ in $y = c'$. Iz tega sledi, da ima presek lika U in opazovanega pasu ploščino $d \cdot (c' - c)$. Če bi zdaj podobno naredili z vsemi pasovi in dobljene ploščine sešteli, bi dobili ravno ploščino celega lika U. Zdaj imamo tak postopek:

A1 Naj ima i-ti pravokotnik spodnji levi kot (x_{i1}, y_{i1}) in zgornji desni kot (x_{i2}, y_{i2}). Postavimo $S := 0$; s prištevanjem bo S na koncu postala ploščina celega lika U.

A2 Ravnino bo treba prerezati pri vseh y_{i1} in vseh y_{i2}. Pripravimo si torej nek seznam trojic, ki bo vseboval za vsak i trojici $\langle y_{i1}, i, 1\rangle$ in $\langle y_{i2}, i, 2\rangle$; vse te trojice uredimo po naraščajoči y-koordinati. Drugi dve komponenti nam povesta, od katerega pravokotnika in katerega roba (zgornjega ali spodnjega) je določena trojica prišla; to bo prišlo kasneje še prav. Oštevilčimo dobljene y-koordinate v naraščajočem vrstnem redu kot y_j, $j = 1, \ldots, 2n$.

A3 (Ta korak izvedemo po enkrat za vsak j od 1 do $2n - 1$.) Ogledali si bomo presek U in pasu $y_j \leq y \leq y_{j+1}$. Naj bo T unija intervalov $[x_{i1}, x_{i2}]$ po vseh tistih i, za katere je $y_{i1} \leq y_j$ in $y_{j+1} \leq y_{i2}$. Naj bo d skupna dolžina te unije intervalov. Povečajmo S za $(y_{j+1} - y_j) \cdot d$.

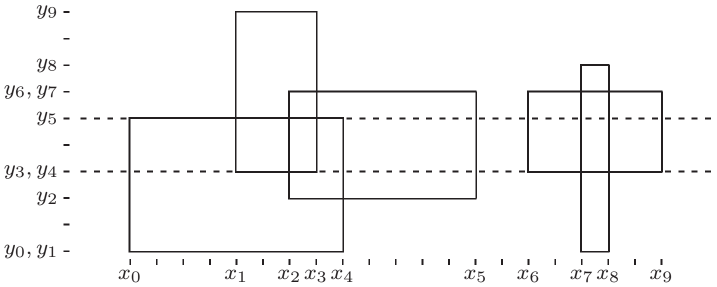
A4 Na koncu tega postopka je S ravno ploščina celega lika U.

Oglejmo si zdaj podrobneje delo z unijo intervalov v točki A3. Recimo, da bi T vsebovala intervale $[1, 3]$, $[2, 5]$, $[0, 6]$ in $[8, 18]$; dolžina unije teh štirih intervalov je $d = 16$, saj pokrivajo vsega skupaj x-koordinate od 0 do 6 in od 8 do 18. Nekatere podintervale, na primer $[1, 5]$, sicer pokriva po več intervalov iz T, vendar je za skupno dolžino pomembno le, ali je nek podinterval sploh pokrit ali ne, če je pokrit večkratno, pa ga moramo v dolžino unije še vedno šteti le enkrat. Navedeni štirje intervali pravzaprav razrežejo os x na disjunktne podintervale: $[0, 1]$ (ki ga pokriva en sam interval iz T, namreč $[0, 6]$); $[1, 2]$ (ki ga pokrivata dva, namreč $[0, 6]$ in $[1, 3]$); $[2, 3]$ (ki ga pokrivajo trije, poleg prejšnjih dveh še $[2, 5]$); $[3, 5]$ (ki ga pokrivata dva, $[0, 6]$ in $[2, 5]$); $[5, 6]$ (ki ga pokriva samo $[0, 6]$); $[6, 8]$ (ki ga ne pokriva nobeden od intervalov iz T); in $[8, 18]$ (ki ga pokriva samo $[8, 18]$). Če bi torej za dano množico intervalov preučili, kako nam razrežejo os x na disjunktne podintervale, bi morali potem le še sešteti dolžine tistih podintervalov, ki jih pokriva vsaj en interval iz T. Nov disjunktni podinterval se začne pri vsaki taki x-koordinati, kjer ima kakšen od intervalov v T krajišče (ali levo ali pa desno). To bi lahko naredili s postopkom, podobnim tistemu iz točke A2 zgoraj:

B1 Recimo, da imamo v T intervale $[x_{i1}, x_{i2}]$ za nekaj vrednosti i. Pripravimo seznam vseh parov $\langle x_{i1}, 1\rangle$ in $\langle x_{i2}, 2\rangle$ za vse intervale $[x_{i1}, x_{i2}]$ v T. Uredimo te pare po naraščajoči x-koordinati. Recimo, da tako urejene oštevilčimo kot $\langle x_k, t_k\rangle$ za $k = 1, \ldots, 2m$ (če je m število intervalov v T). Postavimo $d := 0$, $c := 0$. V d bomo zbirali skupno dolžino unije intervalov, v c pa bomo za trenutni podinterval vzdrževali podatek, koliko intervalov iz T ga pokriva.

B2 Ponovimo koraka B3 in B4 za vsak k od 1 do $2m - 1$:

B3 Zdaj gledamo interval od x_k do x_{k+1}. Če je $t_k = 1$, je x_k levo krajišče nekega intervala in moramo c povečati za 1, ker podinterval $[x_k, x_{k+1}]$



Primer naloge s petimi pravokotniki. Oznake $x_0, \ldots, x_9$ predstavljajo x-koordinate njihovih levih in desnih stranic v naraščajočem vrstnem redu, podobno pa $y_0, \ldots, y_9$ za y-koordinate njihovih spodnjih in zgornjih stranic.

Če se omejimo na pas ravnine med premicama $y = y_4$ in $y = y_5$, vidimo, da pripadajo našim pravokotnikom tu naslednji intervali x-koordinat: $[x_0, x_4]$, $[x_1, x_3]$, $[x_2, x_5]$, $[x_6, x_9]$ in $[x_7, x_8]$, torej po eden za vsak pravokotnik, ki sega v ta pas. Seveda se ti intervali prekrivajo in je njihova unija preprosto $[x_0, x_5] \cup [x_6, x_9]$. Skupna dolžina te unije je $13 + 5 = 18$ enot, širina pasu pa je 2 enoti, tako da ta pas k površini unije opazovanih pravokotnikov prispeva 36 enot. Podobno bi za pas od y_1 do y_2 ugotovili, da prispeva $9 \times 2 = 18$ enot, pas od y_2 do y_3 prispeva $14 \times 1 = 14$ enot, pas od y_5 do y_6 prispeva 14 enot, pas od y_7 do y_8 štiri enote in pas od y_8 do y_9 še šest enot. Tako vidimo, da je ploščina unije teh petih pravokotnikov enaka $18+14+36+14+4+6 = 92$ enot.

pokriva en T-jev interval več kot podinterval $[x_{k-1}, x_k]$ pred njim. Če pa je $t_k = 2$, je x_k desno krajišče nekega intervala in moramo c zmanjšati za 1. ter jih vse skupaj uredili po naraščajoči x-koordinati.

B4 Če je zdaj $c > 0$, je podinterval $[x_k, x_{k+1}]$ pokrit z vsaj enim od intervalov iz T, zato povečajmo d za $x_{k+1} - x_k$. Če pa je $c = 0$, pustimo d pri miru.

B5 Na koncu tega postopka je v d ravno dolžina unije vseh intervalov iz T.

Pokazati je mogoče, da zahteva urejanje m elementov (če uporabimo dovolj učinkovit algoritem) v najslabšem primeru čas $O(m \log m)$. Če bi torej uporabili ta postopek v točki A3 zgornjega algoritma, bi potrebovali za tisto točko v najslabšem primeru $O(n \log n)$ časa (čas, potreben za to, da ugotovimo, kateri pravokotniki sploh segajo v trenutni pas, je le $O(n)$ in bi se izgubil v času urejanja v točki B1), izvesti pa jo moramo $O(n)$-krat, tako da bi celoten postopek lahko pri n pravokotnikih porabil do $O(n^2 \log n)$ časa. Na srečo pa lahko to še precej izboljšamo.

Opazimo namreč lahko, da sta si množici intervalov pokritih x-koordinat za dva sosednja pasova zelo podobni. Ker smo rezali na pasove pri vsaki višini, kjer ima kateri od pravokotnikov svoj zgornji ali pa spodnji rob, se dva sosednja pasova glede pokrivanja x-koordinat razlikujeta le na enega od naslednjih dveh načinov: če je med njima spodnji rob i-tega pravokotnika, je pokrit v zgornjem od opazovanih dveh pasov tudi interval $[x_{i1}, x_{i2}]$, ki v spodnjem mogoče še ni bil; in če je med njima zgornji rob i-tega pravokotnika, interval $[x_{i1}, x_{i2}]$ v zgornjem pasu ni več pokrit, v spodnjem pa je še bil (no, lahko je tudi v zgornjem pasu še vedno pokrit, deloma ali pa celo v celoti, če ga pokrivajo kakšni drugi intervali). Torej, če bi imeli množico pokritih intervalov T predstavljeno na primeren način, bi jo morali ob prehodu z enega pasu na naslednjega le še malo popraviti, ne bi pa je bilo treba v celoti sestavljati na novo.

Drugo koristno opažanje je, da so krajišča intervalov, s katerimi imamo v množici T opraviti, vedno take x-koordinate, pri katerih ima eden od naših n pravokotnikov svoj levi ali desni rob; vseh možnih krajišč je torej največ $2n$, čeprav se v posameznem pasu pri marsikaterem od njih mogoče ne zgodi nič zanimivega, če tistega pravokotnika v tem pasu pač ni. Recimo, da imamo v našem primeru opravka z $n = 5$ pravokotniki in jim po x-koordinatah ustrezajo intervali $[1, 3]$, $[2, 5]$, $[0, 6]$, $[8, 18]$ in $[15, 20]$. Najfinejše razbitje na podintervale, s katerim bi utegnili imeti pri množici T opravka, je torej razbitje na $\langle 0, 1, 2, 3, 5, 6, 8, 15, 18, 20\rangle$. Tu imamo vseh $2n = 10$ krajišč, ki nam opisujejo skupno devet podintervalov. Vse, kar moramo v danem trenutku vedeti o množici T, da lahko izračunamo dolžino unije pokritih intervalov, je podatek o tem, katere od teh 9 podintervalov pokriva vsaj eden od intervalov, ki so trenutno v T. Ko dodamo v T nek nov interval, se pokritost nekaterih podintervalov poveča za 1, ko pa iz T nek interval zbrišemo, se pokritost nekaterih zmanjša za 1. Algoritma A in B lahko torej zamenjamo s takšnim postopkom:

C1 Naj ima i-ti pravokotnik spodnji levi kot (x_{i1}, y_{i1}) in zgornji desni kot (x_{i2}, y_{i2}). Postavimo $S := 0$; s prištevanjem bo S na koncu postala ploščina celega lika U.

C2 Pripravimo si seznam trojic $\langle y_{i1}, i, 1\rangle$ in $\langle y_{i2}, i, 2\rangle$ za vse i in jih uredimo po naraščajoči y-koordinati; oštevilčimo dobljene y-koordinate v naraščajočem vrstnem redu kot y_j, $j = 0, \ldots, 2n - 1$.

C3 Pripravimo si seznam trojic $\langle x_{i1}, i, 1\rangle$ in $\langle x_{i2}, i, 2\rangle$ za vse i in jih uredimo po naraščajoči x-koordinati; oštevilčimo dobljene x-koordinate v naraščajočem vrstnem redu kot x_j, $j = 0, \ldots, 2n-1$. Za vsak pravokotnik si lahko zdaj tudi zapišemo, kateri v tem vrstnem redu sta x-koordinati, ki predstavljata njegov levi in desni rob; naj bo torej u_{i1} položaj koordinate x_{i1} v tem vrstnem redu, u_{i2} pa položaj koordinate x_{i2}.

C4 T naj bo množica trenutno pokritih intervalov; na začetku je prazna.

C5 (Ta korak izvedemo po enkrat za vsak j od 0 do $2n-1$.) Če je y_j spodnja stranica i-tega pravokotnika, dodaj $[x_{u_{i1}}, x_{u_{i2}}]$ v T; sicer pa je y_j zgornja stranica i-tega pravokotnika in zbriši $[x_{u_{i1}}, x_{u_{i2}}]$ iz T. Naj bo zdaj d skupna dolžina unije intervalov v T. Povečajmo S za $(y_{j+1} - y_j) \cdot d$.

C6 Na koncu tega postopka je S ravno ploščina celega lika U.

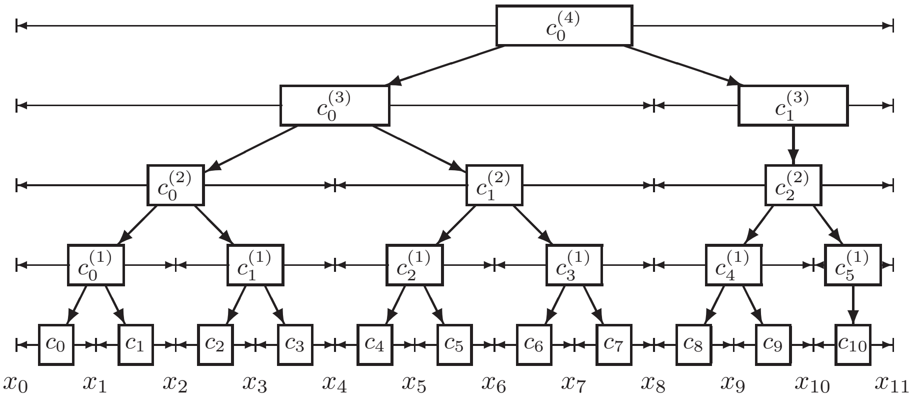
Preprosta oblika podatkovne strukture, s katero bi lahko predstavili množico T, bi bila tabela $2n - 1$ števil, v kateri bi j-to število (recimo mu c_j) povedalo, kolikokrat je pokrit interval $[x_j, x_{j+1}]$. Operacije na njej bi izvajali takole:

Inicializacija podatkovne strukture T:
    for $j := 1$ **to** $2n - 1$ **do** $c_j := 0$;
    $d := 0$
Dodajanje intervala $[x_u, x_v]$ v T:
    for $j := u$ **to** $v - 1$ **do**
        $c_j := c_j + 1$;
        if $c_j = 1$ **then** $d := d + (x_{j+1} - x_j)$
Brisanje intervala $[x_u, x_v]$ iz T:

for $j := u$ **to** $v - 1$ **do**
  $c_j := c_j - 1$;
  if $c_j = 0$ **then** $d := d - (x_{j+1} - x_j)$

Lahko se zgodi, da je interval, ki ga dodajamo, zelo širok in moramo zato spremeniti veliko elementov tabele c. V najslabšem primeru bo dodajanje ali brisanje intervala trajalo $O(n)$ časa. Ker moramo po eno tako operacijo izvesti v vsaki izvedbi koraka C5, ta pa se izvede $O(n)$-krat, je skupna zahtevnost algoritma $O(n^2)$ (v primerjavi s tem je čas $O(n \log n)$ za urejanje v točkah C2 in C3 nepomemben). To je že malo bolje od prejšnjega algoritma, zares dobro pa še vedno ni.

Še nadaljnjo pomembno izboljšavo pa dobimo, če nad tabelo c postavimo še drevesasto strukturo z več manjšimi tabelami. Vpeljimo tabelo $c^{(1)}$ z n elementi; vsak od njih bo predstavljal pokritost dveh sosednjih podintervalov: element $c_j^{(1)}$ se bo nanašal na podinterval $[x_{2j}, x_{2(j+1)}]$. (Če n ni večkratnik števila 2, si mislimo, da zadnji element tabele $c^{(1)}$ predstavlja le zadnji podinterval, ne pa dveh skupaj. Podobno ravnamo tudi pri kasnejših tabelah.) Podobno vpeljimo $c^{(2)}$ z $\lceil n/2 \rceil$ elementi, pri čemer $c_j^{(2)}$ predstavlja podinterval $[x_{4j}, x_{4(j+1)}]$. Tako nadaljujemo do tabele $c^{(K)}$ za $K = \lceil \lg 2n \rceil$; ta ima en sam element, ki predstavlja celoten interval $[x_0, x_{2n-1}]$. Takšni drevesasti strukturi pravijo „drevo segmentov“ (*segment tree*).



Primer drevesa segmentov za $n = 6$ intervalov s skupno 12 krajišči $x_0, \ldots, x_{11}$, ki določajo 11 osnovnih podintervalov. Vodoravne puščice pri vsakem vozlišču drevesa kažejo, kateri interval x-koordinat predstavlja to vozlišče.

Namen te podatkovne strukture je, da v primerih, ko moramo dodati ali brisati nek interval $[x_u, x_v]$ in je v veliko večji od u, ne bo treba spreminjati elementov c_j za vsak posamezen j od u do $v - 1$, ampak bo dovolj že, če bomo spremenili peščico primerno izbranih elementov v višje ležečih tabelah, saj vsak od njih zaleže za več elementov prvotne tabele c_j. Števec $c_j^{(k)}$ tako predstavlja interval $[x_{2^k \cdot j}, x_{2^k \cdot (j+1)}]$. Doslej nam je vrednost d predstavljala skupno dolžino unije vseh pokritih intervalov; zdaj pa bomo pri vsakem vozlišču drevesa hranili ločeno vrednost $d_j^{(k)}$, ki pomeni skupno dolžino unije vseh pokritih intervalov v poddrevesu, ki se začenja pri tem vozlišču, če se delamo, kot da ni nobeden od prednikov tega vozlišča pokrit že sam po sebi. Tisto, kar smo doslej imenovali d, je tako pravzaprav $d_0^{(K)}$.

Dodajanje intervala $[x_u, x_v]$ *v* $c_j^{(k)}$*:*

1 če je $u \le 2^k \cdot j$ in $2^k \cdot (j+1) \le v$:
2   povečaj $c_j^{(k)}$ za 1
3 sicer:
4   če je $u < 2^k \cdot (j + 1/2)$: dodaj $[x_u, x_v]$ v $c_{2j}^{(k-1)}$.
5   če je $v \ge 2^k \cdot (j + 1/2)$: dodaj $[x_u, x_v]$ v $c_{2j+1}^{(k-1)}$.
6 popravi $d_j^{(k)}$:
7   če je $c_j^{(k)} > 1$, naj bo $d_j^{(k)} \leftarrow x_{2^k \cdot (j+1)} - x_{2^k \cdot j}$ (interval je pokrit v celoti)
8   sicer, če je $k = 0$, naj bo $d_j^{(k)} \leftarrow 0$ (nepokrit list)
9   sicer naj bo $d_j^{(k)} \leftarrow d_{2j}^{(k-1)} + d_{2j+1}^{(k-1)}$ (seštejmo pokritost podintervalov)

Mi kot uporabniki drevesa moramo ta rekurzivni postopek za dodajanje pognati pri korenu drevesa, torej pri $c_0^{(K)}$. Rekurzija se izteče, ko pade k na 0; element $c_j^{(0)}$ je pravzaprav kar c_j. Pogoj v vrstici 1 je v tem primeru zagotovo izpolnjen, saj smo pri razbitju na osnovne podintervale, ki jih predstavljajo elementi c_j, že upoštevali leve in desne stranice vseh pravokotnikov, torej vrednosti x_u in x_v, s katerima imamo zdajle opravka, ne moreta razbiti osnovnega podintervala $[x_j, x_{j+1}]$, ki ga predstavlja števec c_j; če je do klica sploh prišlo, to že zanesljivo pomeni, da je ta interval v celoti vsebovan v $[x_u, x_v]$. V vrsticah 4 in 5 izvedemo rekurzivna klica istega podprograma. Po spremembah, ki se zgodijo v vrsticah 1–5, pa moramo v vrsticah 6–9 še popraviti vrednost $d_j^{(k)}$.

Brisanje je povsem podobno, le da moramo v vrstici 2 vrednost $c_j^{(k)}$ zmanjšati za 1 namesto povečati za 1. Inicializacija podatkovne strukture je preprosto v tem, da pripravimo vse tabele in postavimo njihove elemente (c-je in d-je) na 0.

Koliko dela imamo pri dodajanju nekega novega intervala $[x_u, x_v]$ v T? Opazimo lahko, da ob vsakem klicu podprograma za dodajanje $[x_u, x_v]$ v $c_j^{(k)}$ (ki predstavlja interval $[x_L, x_D]$ za $L = 2^k \cdot j$, $D = 2^k \cdot (j+1)$) velja $x_u < x_D$ in $x_v > x_L$. Zdaj lahko ločimo klice štirih vrst glede na to, ali $[x_u, x_v]$ vsebuje x_L, x_D, nobeno ali obe: (A) $x_u \le x_L < x_v < x_D$; (B) $x_L < x_u < x_D \le x_v$; (C) $x_L < x_u < x_v < x_D$; (D) $x_u \le x_L < x_D \le x_v$. Če premislimo, kakšni rekurzivni klici utegnejo nastati iz klica posamezne vrste, jih lahko zapišemo takole: $A \to A|DA|D$, $B \to B|BD|D$, $C \to D|C|BA|BD|DA$, $D \to \varepsilon$. S tem mislimo, da lahko iz klica tipa A nastane ali en klic tipa A ali pa en klic tipa D ali pa en klic tipa D in en klic tipa A, ipd. Iz klica tipa D ne nastane noben rekurzivni klic, kar smo ponazorili z ε. Če imamo zdaj seznam klicev, ki se izvedejo na nivoju k, lahko do klicev za en nivo nižje $(k-1)$ pridemo tako, da vsako črko v seznamu zamenjamo z desno stranjo enega od gornjih pravil. Na primer, če so se na nekem nivoju izvedli klici ADB, se lahko na naslednjem izvedejo $ADDB$ ali pa DDB ali pa AB ipd. Iz oblike pravil vidimo, da (ne glede na to, kakšnega tipa je bil začetni klic za dodajanje v koren) na nobenem nivoju ne bo več kot en klic tipa A, en tipa B in en tipa C; klici tipa D pa lahko nastanejo le iz klicev tipov A, B in C s prejšnjega nivoja in torej tudi ne morejo biti več kot trije. Zato na nobenem nivoju prav gotovo ne more biti več kot šest klicev. Vsak klic pa sam po sebi zahteva konstantno veliko dela, tako da je vsega skupaj le $O(K) = O(\lg n)$ dela. Vsaka od $O(n)$ iteracij točke C5 porabi torej $O(\lg n)$ časa, inicializacija vseh tabel $c^{(k)}$ le $O(n)$, urejanje v točkah C2 in C3 pa še vedno $O(n \lg n)$ kot doslej. Celoten postopek torej porabi $O(n \lg n)$

časa. Pokazati se da, da asimptotično boljšega algoritma ni.[4]

R1998.2.4

Naloga: str. 6

Naloga zahteva, naj izpisujemo odkrite naslove v enakem vrstnem redu, v kakršnem smo prebirali imena računalnikov iz vhodne datoteke. To pomeni, da če procesor, ki je iskal naslov $(n+1)$-vega računalnika iz vhodne datoteke, konča svoje delo prej kot tisti, ki je iskal naslov n-tega, bomo morali čakati še na tega slednjega, preden bomo lahko izpisali naslova obeh računalnikov. Seveda je neugodno, če bi moral procesor, ki je hitro našel naslov $(n+1)$-vega, zdaj čakati brez dela, da bo nek drug procesor prej našel še naslov n-tega; bolje je, če lahko procesorju, ki je našel naslov $(n+1)$-vega računalnika, takoj dodelimo v iskanje neko novo ime, naslov, ki ga je našel, pa si nekje zapomnimo. Pri tem moramo upoštevati, da imamo za te naslove na voljo omejeno mnogo pomnilnika; če na primer procesor, ki išče naslov n-tega računalnika, zavlačuje toliko časa, da smo medtem že našli naslove za tisoč naslednjih računalnikov, nam ne preostane drugega, kot da ostane nekaj procesorjev brez dela, medtem ko čakamo še na naslov n-tega računalnika, saj več kot toliko že najdenih naslovov ne moremo hraniti v pomnilniku.

Spodnji program lahko hrani največ ShranjenNaslovM naslovov računalnikov. Zanje uporablja tabelo ShranjenNaslov, ki deluje kot krožni medpomnilnik; naslovi so v njej shranjeni v takšnem vrstnem redu, v kakršnem smo prebrali imena računalnikov iz vhodne datoteke, veljavni elementi te tabele pa so na indeksih od IndNajnizjeZapSt do pred (ZapSt **mod** ShranjenNaslovM) + 1. Za vsako ime, ki ga damo v iskanje kakšnemu procesorju, si zapomnimo, kam v tej tabeli bo treba na koncu shraniti njegov naslov. Ko izvemo naslov za indeks IndNajnizjeZapSt, ga lahko izpišemo, mogoče pa tudi še nekaj naslednjih naslovov, če smo te našli že prej.

```
program Naslovi(Input, Output);
const
  StProcesorjev = 64;
  ShranjenNaslovM = 900; { dolžina tabele ShranjenNaslov }
type
  Niz = packed array [1..64] of char;
var
  ImeRac: Niz;
  { tabela urejenih, še neizpisanih naslovov }
  ShranjenNaslov: array [1..ShranjenNaslovM] of Niz;
  { zaporedna številka, ki je v obdelavi na posameznem procesorju }
  PripadajocaZapSt: array [1..StProcesorjev] of integer;
  { indeks, kjer je shranjen naslov s trenutno najnižjo zaporedno številko }
  IndNajnizjeZapSt: integer;
  { največja zaporedna številka, ki jo je možno trenutno hraniti v tabeli ShranjenNaslov }
  NajvecjaZapSt: integer;
  ZapSt: integer;                 { zaporedna številka prebranega podatka }
  vObdelavi: integer;             { število imen, ki so v paralelni obdelavi }
```

[4] Več o tovrstnih algoritmih se najde v kakšni knjigi o računski geometriji. Na primer: Franco P. Preparata, Michael Ian Shamos: *Computational Geometry: An Introduction*, 2nd ed., Springer, 1988, kjer govori o našem problemu razdelek 8.4, o drevesu segmentov pa razdelek 1.2.3.1. Še en primer naloge, ki zahteva unijo pravokotnikov, je naloga A ("Atlantis") z ACMovega študentskega tekmovanja v programiranju za "Mid-Central Europe" (Freiburg, 19. nov. 2000); opis naloge s testnimi primeri je npr. na `http://www.acm.inf.ethz.ch/ProblemSetArchive/B_EU_MCRC/2000/`.

```
   j: integer;

   function Obdelaj(Ime: Niz): integer; external;
   function PoberiRezultat(var Naslov: Niz): integer; external;

procedure IzpisiKarMores;
{ Vzame en rezultat iz obdelave, ga uvrsti na pravo mesto v tabelo ShranjenNaslov
  in izpiše, kolikor se je nabralo urejenega dela rezultatov v tej tabeli. }
var
   j, Kam: integer;
   Naslov: Niz;
begin
   j := PoberiRezultat(Naslov); vObdelavi := vObdelavi − 1;
   Kam := (PripadajocaZapSt[j] − 1) mod ShranjenNaslovM + 1;
   ShranjenNaslov[Kam] := Naslov;
   while ShranjenNaslov[IndNajnizjeZapSt] <> '' do begin
      WriteLn(ShranjenNaslov[IndNajnizjeZapSt]);
      ShranjenNaslov[IndNajnizjeZapSt] := '';
      IndNajnizjeZapSt := IndNajnizjeZapSt mod ShranjenNaslovM + 1;
      NajvecjaZapSt := NajvecjaZapSt + 1;
   end; {while}
end; {IzpisiKarMores}

begin {Naslovi}
   ZapSt := 0; vObdelavi := 0;
   IndNajnizjeZapSt := 1; NajvecjaZapSt := ShranjenNaslovM;
   for j := 1 to ShranjenNaslovM do ShranjenNaslov[j] := '';
   while not Eof do begin
      ReadLn(ImeRac); ZapSt := ZapSt + 1;
      while (vObdelavi >= StProcesorjev) or (ZapSt > NajvecjaZapSt) do
         IzpisiKarMores;
      j := Obdelaj(ImeRac);
      PripadajocaZapSt[j] := ZapSt; vObdelavi := vObdelavi + 1;
   end; {while}
   while vObdelavi > 0 do IzpisiKarMores;
end. {Naslovi}
```

REŠITVE NALOG ZA TRETJO SKUPINO

Naloga: str. 6

R1998.3.1

Funkcija za iskanje najkrajših poti, ki smo jo dobili podano, ne upošteva odloka o tem, kako je dovoljeno v posameznem križišču zavijati. Z drugimi besedami, če se nekaj ulic stika v določenem križišču, bo podana funkcija zadovoljna že s potmi, ki poljubno zavijajo z ene ulice na drugo. Če jo hočemo uporabiti za iskanje poti v skladu z novim odlokom, moramo torej graf predelati tako, da v križiščih sploh ne bo tistih ulic, na katere ne smemo zavijati. Toda v obstoječem grafu je to, na katere ulice je v določenem križišču dovoljeno zaviti, odvisno od tega, iz katere smeri smo v križišče prišli. Torej je zdaj pametno iz vsakega križišča, v katerem se stika k ulic, narediti k ločenih križišč, v katerem bo v vsakega vodila po ena ulica, iz nje pa le tiste, na katere smemo zaviti, če pridemo v to križišče po tej ulici.

Naj bo V množica križišč prvotnega grafa, E pa množica ulic. Naj bo $N(u)$ množica križišč, s katerimi je prek svojih ulic neposredno povezano križišče u.

Množico križišč novega grafa definirajmo takole:

$$V' := \{u_v : u \in V, v \in N(u)\}.$$

Novo križišče u_v ponazarja križišče u, v katerega smo se pripeljali iz smeri v. Nasledniki tega križišča naj bodo w_u za vse tiste $w \in N(u)$, za katere zavoj iz smeri $v \to u$ v smer $u \to w$ ni zavoj v levo. Pomembno je, da je novi graf usmerjen; z drugimi besedami, po vsaki povezavi gremo lahko le v eno smer.[5] Dolžina povezave med u_v in w_u naj bo enaka kot dolžina povezave med u in w v prvotnem grafu.

Zdaj lahko poiščemo najkrajšo pot od u do v v prvotnem grafu tako, da poiščemo v novem grafu najkrajše poti od vseh u_w do vseh v_x za vse $w \in N(u)$, $x \in N(v)$ ter vzamemo najkrajšo od vseh teh poti (tu smo seveda predpostavili, da je vseeno, v kakšni smeri začnemo voziti od križišča u in iz katere smeri pridemo na koncu v križišče v).

R1998.3.2

Naloga: str. 7

Edine količine, ki se pri izračunu šifre lahko spreminjajo, so vrednosti programskega števca in obeh registrov. To pomeni, da je program za izračun odgovora v vsakem trenutku v enem od 2^{32} stanj. Takoj, ko se eno od stanj ponovi, lahko zaključimo, da se je program ujel v neskončno zanko. Oziroma: če program uspešno izračuna odgovor, ga izračuna prej kot v 2^{32} korakih.

Procesor Merlin izvrši 2^{32} ukazov v dobrih 7,52 s. To pomeni, da zadostuje, če po vsakem vprašanju počakamo 8 s. Če po tem času ne dobimo odgovora, se je program pri danem vprašanju ujel v neskončno zanko. To moramo ponoviti za vsa možna vprašanja, kar pomeni, da lahko opravimo test v slabih šestih dnevih.

R1998.3.3

Naloga: str. 7

Oglejmo si najprej, kako ročno preiskati tabelo iz naloge. Po tabeli se bomo pomikali z dvema indeksoma, „levi“ bo šel od leve proti desni, „desni“ mu bo prihajal nasproti z druge strani. V začetku kaže levi na 2, desni na 95. Vsota je 97, kar je premalo, zato jo povečamo tako, da pomaknemo levi indeks za eno mesto naprej. $7 + 95$ je že preveč — vsota se zmanjša, če pomaknemo za eno mesto desni indeks. $7 + 90$ je spet premalo, zato premaknemo levi indeks in pridemo do prve rešitve, $10 + 90$. Nadaljujemo tako, da premaknemo oba indeksa. $14 + 75$ je premalo, prestavimo levega. $24 + 75$ je (za las) premalo in ko spet premaknemo levi indeks, dobimo $31 + 75$. Ker je to več kot 100, prestavimo desnega in dobimo drugo rešitev, $31 + 69$. Igro nadaljujemo, dokler se indeksa ne srečata.

Koliko seštevanj potrebujemo pri takem postopku reševanja? Razdalja med indeksoma je v začetku $n-1$, v vsakem koraku se zmanjša za 1, kadar odkrijemo par, pa celo za 2. Seštevanj je natančno $n - 1 -$ (*število odkritih parov*).

```
type Tabela = array [1..n] of integer;

procedure IzpisiPare(a: Tabela);
var Levi, Desni: integer;
```

[5] Povezava od u_v do w_u namreč pomeni, da lahko v prvotnem grafu, če pridemo v u iz v, nadaljujemo pot v w. Če pa bi šli po povezavi v nasprotni smeri, od w_u do u_v, bi se čisto neupravičeno delali, kot da lahko v prvotnem grafu, če v w pridemo iz u, nadaljujemo pot nazaj v u in smo nato na istem, kot če bi v u prišli iz v.

```
    Vsota: real;
begin
  Levi := 1; Desni := n;
  while Levi < Desni do begin
    Vsota := a[Levi] + a[Desni];
    if Vsota = 100 then WriteLn(a[Levi], ' ', a[Desni]);
    if Vsota <= 100 then Levi := Levi + 1;
    if Vsota >= 100 then Desni := Desni - 1
  end; {while}
end; {IzpisiPare}
```

Lahko se tudi čisto natančno prepričamo, da naš postopek res ne spregleda nobenega para. Na začetku vsake ponovitve zanke **while** namreč velja naslednji pogoj (takemu pogoju pravimo običajno *zančna invarianta*): program je izpisal od parov z vsoto 100 že vse tiste (a_i, a_j), ki imajo $i < l$ ali pa $j > d$. (Pisali bomo l namesto Levi in d namesto Desni.) Res, na začetku ta pogoj velja, saj je $l = 1$, $d = n$ in takih parov sploh ni. Recimo, da velja na začetku neke opazovane ponovitve glavne zanke. Če je $a_l + a_d \leq 100$, se bo l povečal za 1 in omenjeni pogoj bo po novem pokrival tudi pare z $i = l$, $j \leq d$, ki jih prej ni; toda ti pari, razen za $j = d$, imajo vsoto gotovo manjšo od $a_l + a_d$ (saj je $j < d$, v tabeli pa so sama različna števila) in zato tudi manjšo od 100; par $i = l$, $j = d$ pa smo tako ali tako posebej pregledali in ga izpisali, če je imel vsoto enako 100. Nato se bo, če je $a_l + a_d \geq 100$, d zmanjšal za 1 in omenjeni pogoj bo po novem pokrival tudi pare z $j = d$, $i \geq l$, ki jih doslej ni; toda ti pari, razen za $i = l$ (ki ga pregledamo posebej), imajo vsoto večjo od $a_l + a_d$ in zato tudi večjo od 100. Vidimo torej, da s povečevanjem l-ja in zmanjševanjem d-ja gotovo ne bomo spregledali nobenega para z vsoto 100.

Naloga: str. 7

R1998.3.4

Reševanja se lahko lotimo rekurzivno. Preberimo prvo črko iz drugega niza (zapisa vozlišč po nivojih); ta mora predstavljati koren, saj je to edino vozlišče na prvem nivoju. Poiščimo to črko v prvem nizu (zapisu vozlišč po načelu „levo poddrevo, koren, desno poddrevo“); vse, kar je levo od nje, se mora torej nanašati na levo poddrevo, vse, kar je desno od nje, pa na desno poddrevo. Zdaj torej vemo, katere črke so v levem poddrevesu; koren levega poddrevesa je potem tista, ki se prva med njimi pojavlja v zapisu po nivojih. Podobno lahko izvemo za koren desnega poddrevesa in ko imamo enkrat koren nekega poddrevesa, že tudi vemo, kaj je v njegovem levem pod-poddrevesu, kaj pa v desnem; itd. Izhodni niz želene oblike bomo dobili, če po obeh rekurzivnih klicih (za levo in desno poddrevo) izpišemo še črko iz korena.

Mimogrede, sprehodu po drevesu, v katerem najprej obiščemo levo poddrevo, nato koren, nazadnje pa še desno poddrevo, pravijo v angleščini *in-order traversal*; če koren obiščemo najprej, je to *pre-order*, če nazadnje, pa *post-order traversal*. V slovenščini jim včasih pravijo *vmesni*, *premi* in *obratni* obhod.

```
function RekonstruirajDrevo(VmesniObhod, NivojskiObhod: string): string;
var ObratniObhod: string;

  procedure Poddrevo(Prvi, Zadnji: integer);
  var C, Koren: char; iKorena, jKorena, i, j: integer;
  begin
    { Pregledati moramo poddrevo z vozlišči VmesniObhod[i]
```

```
      za i = Prvi, Prvi + 1, ..., Zadnji − 1, Zadnji. }
    if Zadnji < Prvi then exit; { Prazno poddrevo. }
    for i := Prvi to Zadnji do begin
      C := VmesniObhod[i];
      { Kje v nivojskem obhodu se pojavlja trenutno vozlišče C? }
      j := 1; while NivojskiObhod[j] <> C do j := j + 1;
      { Zapomniti si hočemo tisto vozlišče trenutnega poddrevesa, ki se v nivojskem
        obhodu pojavlja najbolj zgodaj — to namreč pomeni, da leži najvišje v
        drevesu, torej je to koren našega poddrevesa. }
      if (i = Prvi) or (j < jKorena) then
        begin Koren := C; iKorena := i; jKorena := j end;
    end; {for}
    { Zdaj imamo koren in torej tudi vemo, katera vozlišča so v levem
      in katera v desnem poddrevesu. Njuna opisa bomo z rekurzivnima
      klicema dodali v niz ObratniObhod, na koncu pa mu bomo pritaknili
      še črko iz korena trenutnega drevesa. }
    Poddrevo(Prvi, iKorena − 1);
    Poddrevo(iKorena + 1, Zadnji);
    ObratniObhod := ObratniObhod + Koren;
  end; {Poddrevo};

begin {RekonstruirajDrevo}
  ObratniObhod := ''; Poddrevo(1, Length(VmesniObhod));
  RekonstruirajDrevo := ObratniObhod;
end; {RekonstruirajDrevo}
```