15. republiško tekmovanje v znanju računalništva (1991)

NALOGE ZA PRVO SKUPINO

1991.1.1

R: 9

Mojster Turing je izumil prav čuden stroj. Stroj namreč nima običajnega pomnilnika. Dela z magnetnim trakom, s katerega lahko bere, pa tudi piše lahko po njem. To počne s posebno bralno-pisalno glavo, ki jo premika naprej in nazaj po posameznih zapisih. Zapisi imajo lahko vrednost „0“, „1“ ali „P“, kar pomeni prazno. Na začetku dela je glava na začetku traku (trak ima začetek, konca pa ne — v tisto smer je neskončno dolg).

Stroj upravljaš z naslednjimi ukazi:

PremakniGlavo(Smer: SmerT) premakne glavo v smer Smer (naprej ali nazaj);

PreberiZnak: ZnakT prebere znak s traku, pri čemer se glava ne premakne;

IzpisiZnak(Znak: ZnakT) napiše znak na trak, pri čemer se glava ne premakne.

Turing je pospravljal svojo sobo in našel naslednji program. Žal se ne more spomniti, **kaj program naredi**. Ali mu lahko pomagaš?

```
program TuringovStroj(Input, Output);

  type SmerT = (eNazaj, eNaprej);
       ZnakT = (e0, e1, eP);
  procedure PremakniGlavo(Smer: SmerT); external;
  function PreberiZnak: ZnakT; external;
  procedure IzpisiZnak(Znak: ZnakT); external;

begin
  while PreberiZnak <> eP do
    if PreberiZnak = e0 then begin
      repeat PremakniGlavo(eNaprej)
      until (PreberiZnak = e1) or (PreberiZnak = eP);
      if PreberiZnak = eP then begin
        PremakniGlavo(eNazaj);
        IzpisiZnak(eP);
      end else begin
        PremakniGlavo(eNazaj);
        IzpisiZnak(e1);
        PremakniGlavo(eNaprej);
      end; {if}
```

```
    end else begin
      repeat PremakniGlavo(eNaprej)
      until (PreberiZnak = e0) or (PreberiZnak = eP);
      if PreberiZnak = eP then begin
        PremakniGlavo(eNazaj);
        IzpisiZnak(eP);
      end else begin
        PremakniGlavo(eNazaj);
        IzpisiZnak(e0);
        PremakniGlavo(eNaprej);
      end; {if}
    end; {if}
end. {TuringovStroj}
```

R: 9

1991.1.2

Program za igranje šaha (med človekom in računalnikom) bi radi dopolnili v študijski šahovski program, ki bi omogočal igralcu vračanje svojih potez in vlečenje drugačnih. Program naj bi hranil zadnjih deset šahistovih potez. Šahist naj bi imel tudi možnost, da odigrane in nato vrnjene poteze ponovno odigra, ne da bi moral iste poteze ponovno vpisovati. Ponoviti je možno le poteze, ki so bile tik pred tem vrnjene — po vpisani novi (drugačni) potezi ponovitev ni več možna.

```
type UkazT = (Vrni, Ponovi, Vleci, Konec);
     PotezaT = ...;
```

Šahistovo naslednjo željo ugotovi podprogram:

```
procedure BeriUkaz(var Ukaz: UkazT; var Poteza: PotezaT); external;
```

pri tem je Ukaz lahko:

Vrni	zahtevano je vračanje poteze, če je še kakšna shranjena;
Ponovi	zahtevana je ponovitev poteze, če je bila pred tem vrnjena;
Vleci	igranje nove poteze; pri tem je v spremenljivki Poteza zapisana nova šahistova poteza;
Konec	konec partije.

Pri ukazih Vrni in Ponovi vrednost spremenljivke Poteza ni definirana.

Na voljo imaš še dva podprograma:

```
procedure IgrajPotezo(Poteza: PotezaT); external;
```

odigra podano šahistovo potezo (in računalnikovo protipotezo),

```
procedure VrniPotezo(Poteza: PotezaT); external;
```

pa opravi obratno potezo od podane poteze — vrne šahovnico v stanje pred odigrano navedeno šahistovo potezo. Podprogram deluje pravilno le v primeru, da vračamo poteze v obratnem vrstnem redu, kot so bile pred tem odigrane.

Napiši program!

1991.1.3

R: 11

Napiši program, ki prepiše izvorno kodo pascalskega programa z vhoda na izhod in pri tem uporabi za izpis rezerviranih besed in komentarjev pisave, različne od pisave preostalega besedila. Rezervirane besede so tista zaporedja črk, za katera funkcija RezervBeseda vrne true. Komentarji se prično z znakom „{“ in končajo z „}“, vmes pa lahko nastopajo poljubni znaki razen znaka „}“. V besedilu lahko nastopajo tudi nizi, ki se prično in končajo z znakom „’“, vmes pa se lahko pojavi poljuben znak razen „’“.

Na voljo imaš naslednje podprograme:

PreberiZnak(c) vrne naslednji znak z vhoda v spremenljivko c.

IzpisiZnak(c) izpiše znak c na izhod.

SpremeniPisavo(Pisava) spremeni obliko izpisa na izhodu. Spremenljivka Pisava lahko zavzame vrednosti:

- Krepko (za izpis rezerviranih besed),
- Nagnjeno (za izpis komentarjev),
- Normalno (za izpis vsega ostalega besedila).

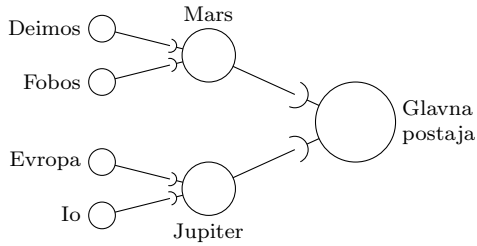
RezervBeseda(Niz, NizL) vrne vrednost true, če je niz dolžine NizL rezervirana beseda, in false, če ni.[1]

Kot približen primer izpisa lahko služi program iz prve naloge.

1991.1.4

R: 12

V osončju so medplanetarno pomembne odločitve sprejete, če zanje glasujejo predstavniki vseh štirih naseljenih lunic. Med glasovanjem sprejme vesoljska postaja *Mars* glasova z lun *Deimos* in *Fobos* ter pošlje v glavno postajo skupen glas obeh lun (DA, če sta obe ZA, sicer NE). Enako stori postaja *Jupiter* z glasovoma z lun *Evropa* in *Io*.



[1] Mišljeno je, da je parameter Niz tipa **packed array** [1..MaxDolz] **of** char (ne pa npr. string) in zato potrebuje eksplicitен podatek o dolžini besede.

V glavni postaji iz obeh prejetih glasov na enak način ugotovijo, ali je bila odločitev sprejeta (če *Mars* in *Jupiter* oba pošljeta glas DA, je odločitev sprejeta, sicer pa ne). Nekoč se je zgodilo, da se je na eni od treh postaj pokvaril eden od sprejemnikov, tako da je bilo videti, da sprejema glas ZA, ne glede na to, kaj je v resnici sprejemal. Da bi takšno napako v bodoče pravočasno odkrili, so sklenili, da bodo odslej pred pravimi glasovanji opravili nekaj poskusnih glasovanj s predpisanimi izjavami (glasovi DA in NE). Napiši **zaporedje poskusnih glasovanj** (za vsako glasovanje 4 predpisane izjave z lun), s katerimi je vedno mogoče odkriti, kateri od šestih sprejemnikov se je pokvaril, ali pa dejstvo, da so vsi brezhibni. (Zanima nas le okvara, pri kateri natanko en sprejemnik trdi, da sprejema DA ne glede na to, kar v resnici sprejema; z drugačnimi okvarami se tu ne ubadamo.)

NALOGE ZA DRUGO SKUPINO

R: 14

1991.2.1

Na nekaj primerih **izračunaj**, kaj program izpiše, in **razloži** način delovanja (postopek)!

```
program KajIzpisem(Input, Output);
type
  Stevilo = 0..255;

var
  a, b: Stevilo;
  i, k, Ham: integer;

  procedure Podprogram(p1: Stevilo; var p2: Stevilo; i: integer);
  var
    b1: integer;
    b2: integer;
  begin
    b1 := (p1 div i) mod 2;
    b2 := (p2 div i) mod 2;
    p2 := (p2 mod i) + Abs(b2 - b1) * i + (p2 div (2 * i)) * (2 * i);
  end; {Podprogram}

begin
  ReadLn(a, b);
  i := 1;
  for k := 1 to 8 do begin
    Podprogram(a, b, i);
    i := 2 * i;
  end; {for}
  Ham := 0;
  for k := 8 downto 1 do begin
    i := i div 2;
```

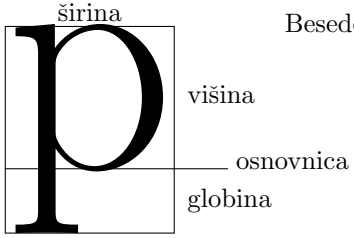
```
    if Odd(b div i) then Ham := Ham + 1;    { Odd(n) pove, če je število n liho. }
  end; {for}
  WriteLn(Ham);
end.
```

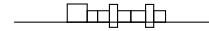
1991.2.2

Program za stavljenje besedil TEX obravnava vsako črko, kot da je zaprta v pravokotnik. Črke ene vrstice so nanizane na črto, ki jo imenujemo *osnovnica*. R: 15

Vsak pravokotnik opišemo z

- globino, to je razdalja med osnovnico in spodnjim robom pravokotnika;
- višino, to je razdalja med osnovnico in zgornjim robom pravokotnika;
- širino.



Besedo „Urejanje“ TEX vidi kot: 

Vrstice nanizanih pravokotnikov TEX razmakne na primerno razdaljo in jih pripravi za izpis. Včasih pa želimo vrstice stisniti čimbolj skupaj, vendar tako, da se pravokotniki ne prekrivajo. **Napiši podprogram**, ki izračuna najmanjšo razdaljo med dvema nepraznima vrstama (razdaljo med njunima osnovnicama), pri kateri se pravokotniki ne prekrivajo.[2] Podatki naj bodo predstavljeni v obliki:

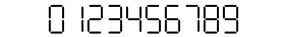
```
const VrstaM = ...;    { Največje dovoljeno število pravokotnikov v vrsti. }
type SkatlaT = record
                 Visina, Globina, Sirina: integer;
               end;
     VrstaT = array [1..VrstaM] of SkatlaT;
```

Argumenti, ki jih dobi podprogram, so:

- Vrsta1, Vrsta2 tipa VrstaT in
- Dolzina1, Dolzina2 (število pravokotnikov v vrsti) tipa integer.

[2]Zanimivo (le malo težjo) različico te naloge dobimo, če ne zahtevamo, da se obe vrstici začneta pri isti x-koordinati, ampak dovolimo, da se ena vrstica zamika malo v levo ali desno glede na drugo.

R: 20 **1991.2.3** Želimo narediti program, ki bo bral podatke z izpolnjenih formularjev. Številke na formularjih so natisnjene s posebnim tiskalnikom, ki jih izpisuje tako, kot to počne večina kalkulatorjev.



Program za razpoznavanje podatkov najprej vsako prebrano številko razbije na sedem delov (ki ustrezajo sedmim segmentom, iz katerih so sestavljene številke). Nato se za vsakega od teh delov odloči, ali je na formularju tam črnilo ali praznina (to stori z nekim nam neznanim postopkom). Pri tem se včasih tudi zmoti. Zato kliče naš program, ki ta delno pravilni opis primerja z opisi vseh številk in pove, kateri je najbolj podoben. Če se ne more odločiti, naj vrne presledek. **Napiši podprogram** za primerjanje znakov.

Oblika znakov (številk), ki se lahko pojavijo v formularju, je shranjena v tabeli OblikeZnakov.

```
type Znak = array [1..7] of boolean;
var OblikeZnakov: array ['0'..'9'] of Znak;
```

Naš podprogram pa dobi en argument tipa Znak.

R: 21 **1991.2.4** Med dvema računalnikoma želimo prenašati poljubna zaporedja bitov, grupiranih v pakete. Na voljo imamo serijsko linijo (prenaša se en bit naenkrat), dolžine posameznih paketov ne poznamo vnaprej in tudi nimamo dovolj pomnilnika, da bi lahko shranili ves paket.

Med paketi dodamo določemo (fiksno dogovorjeno) zaporedje bitov, ki ga bo sprejemni računalnik vedno razpoznal kot mejo med paketi.

Predlagaj postopek, ki bo omogočal prenos poljubnega zaporedja bitov in ki bo omogočal, da na sprejemni strani zanesljivo določimo začetek in konec vsakega paketa. Postopek lahko seveda dodaja svoje bite tudi med bite paketa, vendar naj bo skupno število dodanih bitov majhno v primerjavi s številom koristnih bitov v paketu.

NALOGE ZA TRETJO SKUPINO

R: 22 **1991.3.1** Ugotovi, **kaj počne** naslednji podprogram. **Preizkusi** ga najprej na primeru.

```
type byte = 0..255; { osembitna nepredznačena števila }
function KajStorim(a, b: byte): byte;
var
  Stevec: byte;
  Rezultat: byte;
  Premik: byte;
```

```
begin
  Rezultat := 0;
  Premik := 0;
  for Stevec := 1 to 8 do begin
    Premik := 2 * Premik + (a div 128);
    a := (a mod 128) * 2;
    Rezultat := 2 * Rezultat;
    if b <= Premik then begin
      Premik := Premik − b;
      Rezultat := Rezultat + 1;
    end; {if}
  end; {for}
  KajStorim := Rezultat;
end; {KajStorim}
```

1991.3.2

Napiši podprogram Parse, ki prebere izraz in ga izpiše z oklepaji glede na prioriteto operatorjev. Izraz sestavljajo simboli ⟨*operator*⟩, ⟨*člen*⟩ in ⟨*konec*⟩ (simbol za konec izraza). Izraz je zaporedje simbolov: R: 23

$$\langle člen\rangle\ \langle operator\rangle\ \langle člen\rangle\ \langle operator\rangle \cdots \langle člen\rangle\ \langle konec\rangle$$

Na razpolago imamo naštevni tip SymDef, ki definira simbole:

```
type SymDef = (sClen, sOp, sKonec);
```

Podprogram GetSym vpiše v spremenljivko Sym tip naslednjega simbola v zaporedju. Če ima spremenljivka Sym vrednost sClen, se v spremenljivki Ch po vrnitvi iz GetSym nahaja znak, ki označuje člen; če ima spremenljivka Sym vrednost sOp, se v spremenljivki OpPri nahaja število, ki označuje prioriteto operatorja (nižja številka označuje višjo prioriteto), v spremenljivki Ch pa znak, ki označuje operator. Če bi bilo možno postaviti oklepaje na več enakovrednih načinov (operatorji z enako prioriteto), potem so vse inačice enako pravilne — izberemo poljubno.

Primer: če v izrazih uporabljamo operatorje s prioritetami:[3]

$+$	$-$	$*$	$\uparrow$
3	3	2	1

[3]Iz prikazanega primera je videti, kot da so prioritete vedno majhna naravna števila. Vendar pa, če smo pri sestavljanju rešitve previdni, gre tudi brez te dodatne omejitve — za rešitev je dovolj že, če znamo prioritete samo primerjati med sabo in povedati, katera je višja in katera nižja.

Mimogrede, zanimiv problem dobimo tudi, če zahtevo te naloge obrnemo in poskušamo odstraniti iz izraza vse odvečne oklepaje, ne da bi mu pri tem spremenili pomen. Gl. npr. nalogo E z ACMovega srednjeevropskega študentskega tekmovanja v programiranju CERC 2000 (Praga, 11. nov. 2000).

mora podprogram Parse za vsakega od naslednjih vhodnih izrazov izpisati takšen izhodni izraz:

vhodni izraz	izhodni izraz
$a + b$	$(a + b)$
$a + b * c$	$(a + (b * c))$
$a * b + c$	$((a * b) + c)$
$a + b \uparrow c * d$	$(a + ((b \uparrow c) * d))$
$a + b + c$	$((a + b) + c)$ ali pa $(a + (b + c))$

Uporabljaj naslednje zgoraj opisane spremenljivke in podprograme:

```
var Sym: SymDef;
    Ch: char;
    OpPri: integer;

procedure GetSym; external;
```

R: 25

1991.3.3

Napiši podprogram, ki dobi na vhodu izvorni niz znakov in vzorec ("wild-card") ter njuni dolžini. Podprogram naj vrne true, če izvorni niz ustreza vzorcu. Vzorec sestavljajo znaki, ki morajo biti enaki znakom v izvornem nizu; znaka `*` in `?` v vzorcu imata poseben pomen: `*` pomeni nič, enega ali poljubno znakov v izvornem nizu; `?` pomeni poljuben znak. Predpostaviš lahko, da se znaka `*` in `?` ne pojavita v izvornem nizu znakov.

Primeri:

`'2124123124'`	ustreza	`'*123*124*'`
`'1234567'`	ustreza	`'12*34*7'`
`'111222333'`	ustreza	`'1*2*3'`
`'12321'`	ne ustreza	`'123?*2'`

R: 27

1991.3.4

Pri prenašanju podatkov med dvema računalnikoma lahko prihaja do motenj na komunikacijski zvezi. Da kljub temu zagotovimo zanesljiv prenos, običajno podatke (npr. zaporedje znakov) grupiramo v bloke (npr. po 256 znakov), vsak blok pa opremimo z dodatno informacijo, ki omogoča prejemnemu računalniku ugotoviti morebitne napake v bloku. Prejemni računalnik lahko oddajnemu potrdi pravilen sprejem vsakega bloka ali pa zahteva njegovo ponovitev.

Pravilom (algoritmu), po katerih se ravnata oba računalnika pri medsebojni komunikaciji in opisu blokov (dolžina bloka, dodatna informacija) pravimo *protokol*.

Če je oddaljenost med računalnikoma relativno majhna in hitrost prenašanja podatkov med njima ne posebno velika, potem se sprotno potrjevanje (ali zahtevanje ponovitve) vsakega bloka obnese.

Radi bi prenašali podatke od računalnika v Evropi do računalnika v Ameriki. Pošta nam je zagotovila hitro komunikacijsko zvezo (približno milijon bitov na sekundo) prek geostacionarnega telekomunikacijskega satelita (oddaljenega od površine Zemlje slabih 36 000 km čas potovanja elektromagnetnega valovanja do satelita in nazaj je približno 1/4 sekunde).

Ali bo naš preprosti protokol še vedno učinkovit? Zakaj? **Predlagaj učinkovitejši protokol!** Premisli tudi, ali zanesljivost prenosa (verjetnost napake / znak) vpliva na optimalno dolžino blokov.

REŠITVE NALOG ZA PRVO SKUPINO

R1991.1.1

N: 1

Program za opisani stroj prestavi elemente traku (ničle in enice) za eno mesto proti začetku traku, prvi element pa zavrže. To opravilo izvede tako, da poišče meje med zaporedji ničel in zaporedji enic ter prestavi le ustrezen element na meji. Ko naleti na „P“, napiše presledek tudi prek zadnjega nepraznega elementa, s čimer skrajša pomaknjeno zaporedje za ena.

Če bi bilo na traku ob začetku izvajanja programa več zaporedij ničel in/ali enic, vmes pa presledki, bi program pobrisal prvi znak le iz prve skupine ne-presledkov (ter to skupino premaknil za eno mesto proti začetku), ostale skupine pa bi ostale nedotaknjene. Med prvo in drugo skupino ne-presledkov bi bil po novem en presledek več kot ob začetku izvajanja programa.

Turingov stroj se imenuje po matematiku Alanu Turingu, ki ga je predlagal leta 1936 kot formalizacijo, s katero bi si lahko pomagali pri razmišljanju o tem, kaj je mogoče izračunati „efektivno“ (strojno, mehansko, algoritemsko). Obstaja še več različic Turingovega stroja, ki se od tu opisanega ločijo npr. po tem, da dovolijo več vrst zapisov (ne le „0“, „1“ in „P“ kot v naši nalogi), da imajo trak neskončen v obe smeri, da imajo več trakov ipd. Izkaže pa se, da takšne razširitve ne vplivajo bistveno na zmogljivosti stroja. Dobro je, da je stroj čim preprostejši, ker je potem o njegovem delovanju lažje razmišljati z matematičnimi orodji. Zato Turingov stroj tudi nima pomnilnika z neposrednim dostopom, ampak lahko podatke hrani le na traku (in zato tudi program, podan v naši nalogi, ne uporablja spremenljivk); edina izjema je ta, da ima stroj vedno pri roki podatek o tem, kateri stavek programa se trenutno izvaja.[4]

R1991.1.2

N: 2

Pomagali si bomo s tabelo potez (Poteza), ki jo bomo uporabljali kot neke vrste sklad (na njem je PotezaL potez). V spremenljivki Zadnja si zapomnimo, do kod smo se že vrnili z vračanjem potez. Če je treba potegniti novo potezo, vse za zadnjo pozabimo (PotezaL :=

[4] Več o Turingovih strojih najdemo v učbenikih teoretičnega računalništva, npr. M. Sipser: *Introduction to the Theory of Computation*, 1996; J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, 1979.

Zadnja) in nato dodamo novo potezo na vrh sklada. Vračanje in ponavljanje izvedemo preprosto tako, da zmanjšujemo in povečujemo spremenljivko Zadnja (in pazimo, da ostane na intervalu 1..PotezaL).

```
program Vracanje(Input, Output);
type
  UkazT = (Vrni, Ponovi, Vleci, Konec);
  PotezaT = record ... end;

  procedure BeriUkaz(var Ukaz: UkazT; var Poteza: PotezaT); external;
  procedure IgrajPotezo(Poteza: PotezaT); external;
  procedure VrniPotezo(Poteza: PotezaT); external;

const PotezaM = 10; { število shranjenih potez }
var
  Poteza: array [1..PotezaM] of PotezaT; { shranjene poteze }
  PotezaL: integer;      { število shranjenih potez }
  Zadnja: integer;       { številka zadnje veljavne poteze v shrambi }
  { Pri vlečenju novih potez velja Zadnja = PotezaL,
    sicer pa se lahko spremenljivka Zadnja vrača po shranjenih potezah. }
  Ukaz: UkazT;           { šahistov ukaz }
  p: PotezaT;            { šahistova poteza, če je Ukaz = Vleci }
  i: integer;
begin
  PotezaL := 0; Zadnja := 0;
  repeat
    BeriUkaz(Ukaz, p);
    case Ukaz of
      Vrni:
        if Zadnja > 0 then
          begin VrniPotezo(Poteza[Zadnja]); Zadnja := Zadnja - 1 end;
      Ponovi:
        if Zadnja < PotezaL then
          begin Zadnja := Zadnja + 1; IgrajPotezo(Poteza[Zadnja]) end;
      Vleci:
        begin
          PotezaL := Zadnja; { Zavržemo morebitne še shranjene novejše poteze. }
          if PotezaL < PotezaM then { V shrambi je prostor. }
            PotezaL := PotezaL + 1
          else { Ni prostora, zavržemo najstarejšo shranjeno potezo. }
            for i := 2 to PotezaL do Poteza[i - 1] := Poteza[i];
          Poteza[PotezaL] := p; Zadnja := PotezaL; IgrajPotezo(p);
        end;
    end; {case}
  until Ukaz = Konec;
end. {Vracanje}
```

R1991.1.3

N: 3

Vhodne podatke beremo znak za znakom. Z zastavicama Niz in Komentar si zapomnimo, če smo trenutno v nizu ali v komentarju; ti dve zastavici spreminjamo, ko pridemo do znaka za začetek ali konec niza ali komentarja. Na začetku in koncu komentarja tudi spremenimo pisavo. Ko smo v nizu ali komentarju, prebrane znake nespremenjene tudi izpisujemo. Če nismo niti v nizu niti v komentarju, pa naletimo na zaporedje črk, jih dodajamo v niz Beseda in jih še ne izpisujemo; ko se črke končajo, preverimo, ali je nastala beseda rezervirana in jo v tem primeru izpišemo krepko, sicer pa jo izpišemo v normalni pisavi. V vsakem primeru niz Beseda potem spraznimo, da bo pripravljen na naslednje zaporedje črk, ko bomo spet prišli do kakšnega. Za praznjenje niza Beseda poskrbimo tudi, ko pridemo do konca vhoda, saj je lahko takrat v nizu Beseda še nekaj črk, ki jih še nismo izpisali (res pa je, da pri sintaktično pravilnih pascalskih programih do tega najbrž ne bo prišlo, saj se končajo s piko (za besedo **end**) in mogoče še kakšnim komentarjem). Paziti moramo še na možnost, da je neko zaporedje črk predolgo za tabelo Beseda; takšno gotovo ne bo rezervirana beseda (saj se da vnaprej ugotoviti, kako dolga je najdaljša rezervirana beseda), zato jo lahko izpisujemo kar sproti.

```
program Izpis(Input, Output);
const MaxDolz = 30; { Največja možna dolžina rezervirane besede. }
type
  PisavaT = (Normalno, Nagnjeno, Krepko);
  BesedaT = packed array [1..MaxDolz] of char;
var
  c: char;
  Beseda: BesedaT;
  BesedaL: integer;
  i: integer;
  Niz, Komentar, Rez: boolean;

  procedure PreberiZnak(var c: char); external;
  procedure IzpisiZnak(c: char); external;
  procedure SpremeniPisavo(Pisava: PisavaT); external;
  function RezervBeseda(b: BesedaT; bL: integer): boolean; external;

  { Ta podprogram kličemo, ko pridemo do konca besede. Če je to zelo dolga beseda,
    smo jo izpisovali že sproti in je zdaj ni treba, sicer pa moramo preveriti,
    če je to mogoče ena od rezerviranih besed. }
  procedure KonecBesede;
  begin
    if BesedaL <= MaxDolz then begin
      Rez := RezervBeseda(Beseda, BesedaL);
      if Rez then SpremeniPisavo(Krepko);
      for i := 1 to BesedaL do IzpisiZnak(Beseda[i]);
      if Rez then SpremeniPisavo(Normalno);
```

```
    end; {if}
    BesedaL := 0;
  end; {KonecBesede}

begin {Izpis}
  BesedaL := 0; Niz := false; Komentar := false; SpremeniPisavo(Normalno);
  while not Eof do begin
    PreberiZnak(c);
    if (BesedaL > 0) and not (c in ['A'..'Z', 'a'..'z']) then
      KonecBesede;
    if Niz then begin
      if c = '''' then Niz := false;
      IzpisiZnak(c);
    end else if Komentar then begin
      IzpisiZnak(c);
      if c = '}' then
        begin Komentar := false; SpremeniPisavo(Normalno) end;
    end else if c = '{' then begin
      Komentar := true; SpremeniPisavo(Nagnjeno); IzpisiZnak(c);
    end else if c = '''' then begin
      IzpisiZnak(c); Niz := true;
    end else if c in ['A'..'Z', 'a'..'z', '0'..'9'] then begin
      if BesedaL = MaxDolz then begin
        { Trenutna beseda bo dolga vsaj MaxDolz + 1, torej to ni rezervirana
          beseda. Izpišimo, kar smo že shranili, ostalo pa bomo izpisovali sproti.
          Spremenljivka BesedaL bo ostala pri vrednosti MaxDolz + 1. }
        for i := 1 to BesedaL do IzpisiZnak(Beseda[i]);
        BesedaL := BesedaL + 1;
      end; {if}
      if BesedaL > MaxDolz then IzpisiZnak(c)
      else begin BesedaL := BesedaL + 1; Beseda[BesedaL] := c end;
    end else IzpisiZnak(c);
  end; {while}
  if BesedaL > 0 then KonecBesede;
end. {Izpis}
```

N: 3

R1991.1.4

Minimalno število poskusnih glasovanj je štiri. V nalogi se skriva testiranje logičnega (digitalnega) vezja, sestavljenega iz treh vrat IN (kot kaže leva slika na str. 13), kjer se je na enih od vrat en vhodni signal „zataknil“ na 1. Sprejemnike na Marsu, Jupitru in glavni postaji označimo z $x_1, \ldots, x_6$. Pri vsakem poskusnem glasovanju določimo, kakšne vrednosti pridejo (s posameznih naseljenih lun) do vhodov $x_1, \ldots, x_4$, nato pa pogledamo, kakšen je izhod vezja.

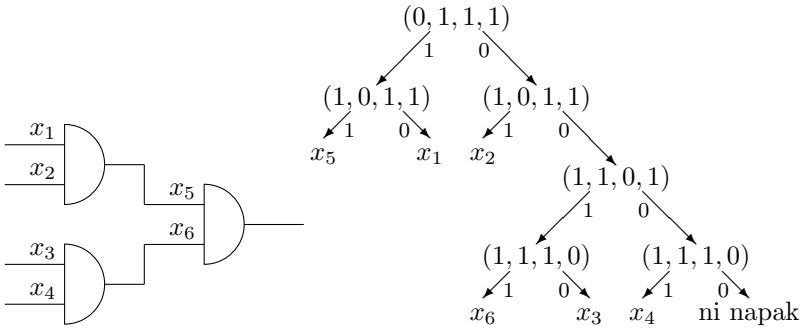
Do primernega zaporedja poskusnih glasovanj lahko pridemo iz pravilnostnih tabel za vseh šest logičnih funkcij, ki opisujejo možna vezja, ki jih dobimo iz prikazanega, če se eden od vhodov „zatakne“ na 1. Lahko

pa si pomagamo tudi s preprostim razmislekom. Če na primer postavimo $x_1 = 0, x_2 = x_3 = x_4 = 1$, izhod iz vezja pa je vendarle 1, pomeni, da je $x_5 = 1$, torej je bodisi x_5 zataknjen na 1 ali pa x_5 deluje normalno, vendar je že do njega prišla vrednost 1, kar pa pomeni, da je x_1 zataknjen na 1. Ti dve možnosti lahko ločimo tako, da postavimo $x_1 = 1, x_2 = 0$: ker x_2 gotovo ni zataknjen, pride zdaj do x_5 zagotovo vrednost 0; če je izhod iz vezja še vedno 1, pomeni, da je zataknjen x_5, sicer pa mora biti zataknjen x_1.

Če pa je prvi test dal rezultat 0, vemo, da je z x_1 in x_5 vse v redu. Zdaj lahko postavimo $x_1 = 1, x_2 = 0$ in če je rezultat vezja zdaj 1, pomeni, da je zataknjen x_2 (druga možnost bi bila, da bi bil zataknjen x_5, kar pa že vemo, da ni res).

Zdaj lahko enako razmišljanje ponovimo za drugo polovico vezja, torej počnemo enake stvari kot v prejšnjih dveh odstavkih, le namesto x_5 si mislimo x_6, namesto x_1 in x_2 pa x_3 in x_4. Če tudi zdaj ne odkrijemo zataknjenega vhoda, pomeni, da so vsi vhodi dobri.

Primerno zaporedje poskusnih glasovanj, s katerim lahko testiramo celoten sistem, je torej: 0111, 1011, 1101, 1110. Pri tem smo vsako glasovanje predstavili z zaporedjem (vektorjem) štirih bitov, ki povedo, kaj pride do vhodov $x_1, \ldots, x_4$. Celoten postopek testiranja lahko zapišemo tudi v obliki odločitvenega drevesa. Vsako notranje vozlišče predstavlja eno poskusno glasovanje; v odvisnosti od rezultata, ki ga ugotovi glavna postaja, se moramo po tem glasovanju premakniti v enega od obeh otrok trenutnega vozlišča. Ko pridemo do lista, pa že lahko točno ugotovimo, kateri vhod je pokvarjen.



Prepričajmo se še o tem, da ni mogoče sestaviti odločitvenega drevesa, ki bi vedno odkrilo zataknjeni vhod (ali ugotovilo, da vsi delujejo pravilno) z največ tremi glasovanji. Očitno je, da vektorjev 1111, 0101, 0110, 1001, 1010 in tistih s tremi ali štirimi ničlami nima smisla dajati na vhode, ker je rezultat neodvisen od tega, ali je kak vhod zataknjen (in kateri). Za prvo glasovanje nam ostaneta v bistvu le dve različni možnosti: vhod z eno ničlo (eden od 0111, 1011, 1101 in 1110) in tak z dvema (0011 ali pa 1100). Če se odločimo za drugo možnost, npr. za $x_1 = x_2 = 0$, $x_3 = x_4 = 1$, nam rezultat vezja pove, ali je x_5 zataknjen

(če je izhod vezja 1) ali ne (če je izhod vezja 0). Če izvemo, da ni zataknjen, nam ostane še šest možnih ugotovitev (pet možnih pokvarjenih vhodov in še možnost, da je vse v redu), med katerimi bi morali zdaj ločiti v naslednjih dveh glasovanjih. Toda z dvema glasovanjema lahko ločimo le štiri možnosti. Pri vhodu $x_1 = x_2 = 1$, $x_3 = x_4 = 0$ velja simetričen razmislek. Torej, če se hočemo izogniti potrebi po štirih glasovanjih, moramo pri prvem glasovanju postaviti na 0 le enega od x_1, x_2, x_3, x_4. Toda zdaj poteka glasovanje na način, kot smo ga opisali zgoraj: če dobimo na izhodu 1, bomo z enim dodatnim glasovanjem ugotovili, kateri vhod je zataknjen, če pa dobimo 0, bomo zdaj za dva vedeli, da sta dobra (na primer x_1 in x_5 pri odločitvenem drevesu na sliki), še vedno pa nam ostane pet možnosti (štirje možni pokvarjeni vhodi in možnost, da je vse v redu), torej spet ne bomo mogli ločiti vseh s samo dvema glasovanjema. Tako torej vidimo, da glasovanj ne moremo razporediti tako, da bi vedno preizkusili vezje s tremi ali manj glasovanji.

Prikazano odločitveno drevo bi odkrilo tudi primere, ko je zataknjenih več vhodov, le da se tedaj ne da vedno samo z opazovanjem izhoda celega vezja ugotoviti, kateri so zataknjeni. Na primer, če sta zataknjena x_1 in x_2 obenem, je izhod vezja vedno enak, kot če bi bil zataknjen vhod x_5.

Če bi obstajala tudi možnost, da se eden od vhodov zatakne na 0, medtem ko ostali delujejo pravilno, bi jo lahko odkrili tako, da bi postavili $x_1 = x_2 = x_3 = x_4 = 1$; če je kateri od vhodov zataknjen na 0, bo izhod iz vezja 0 namesto 1.

REŠITVE NALOG ZA DRUGO SKUPINO

N: 4

R1991.2.1

Program izpiše, v koliko istoležnih bitih se razlikujeta dvojiška zapisa prebranih števil.

Ob klicih podprograma Podprogram med izvajanjem prve zanke po k je i vedno enak 2^{k-1}, torej je potenca števila 2. Zato izraz p1 **div** i zamakne število p1 za $k-1$ mest v desno, (p1 **div** i) **mod** 2 pa izlušči bit $k-1$ števila p1 (b1 dobi vrednost 1, če je bil bit $k-1$ v p1 prižgan, sicer pa 0). Podobno v b2 pripravimo istoležni bit števila p2. Vrednost Abs(b2 − b1) je zdaj enaka 0, če sta bila oba bita prižgana ali oba ugasnjena, sicer pa ima vrednost 1: to je torej ravno b1 **xor** b2. Zdaj bi radi ta rezultat shranili v bit $k-1$ spremenljivke b2 (prejšnja vrednost tega bita se bo ob tem izgubila). Vrednost p2 **mod** i vsebuje spodnjih $k-1$ bitov števila p2, torej od 0 do $k-2$. Vrednost p2 **div** (2 * i) vsebuje bite od vključno k naprej, vendar so zamaknjeni na mesta od 0 naprej; če jo pomnožimo z 2 * i, pridejo ti biti spet na prvotna mesta, na nižjih mestih pa so ničle. Če zdaj to dvoje seštejemo, se rezultat od prvotne vrednosti p2 razlikuje le po tem, da ima bit $k-1$ zanesljivo ugasnjen. Ko prištejemo še Abs(b2 − b1) * i, se bo bit $k-1$ prižgal, če je bilo b1 različno od b2, sicer pa bo ostal ugasnjen. Tako smo dosegli prav to, kar smo želeli: bit $k-1$ števila p2 smo **xor** ali z istoležnim bitom števila p1.

Ko se v zanki pokliče Podprogram po enkrat za vsak bit, je skupni rezultat vsega tega prav tak, kot če bi izvedli prireditev b := a **xor** b. Druga zanka **for** prešteje, koliko je v tej vrednosti prižganih bitov. i ima na začetku vrednost 2^8, ker pa ga takoj na začetku vsake iteracije delimo z 2, ima potem vrednost 2^{k-1}. Število b **div** i zdaj vsebuje b-jeve bite od vključno bita $k-1$ naprej, le da so zamaknjeni na najbolj spodnje položaje; bit $k-1$ sam je tako zamaknjen v bit 0 in lahko preverimo, če je prižgan, preprosto tako, da preverimo, če je število b **div** i liho. Tako lahko preštejemo enice v številu b (oz. v a **xor** b za prvotni vrednosti a in b) ter s tem izvemo, pri koliko bitih se a in b razlikujeta. Ta vrednost (ki se nam med izvajanjem te zanke počasi nabira v spremenljivki Ham) se imenuje tudi Hammingova razdalja med bitnima nizoma a in b.

R1991.2.2

N: 5

Program reši problem tako, da se premika po obeh vrstah hkrati in za vsak rob pravokotnika izračuna razdaljo do pravokotnika v drugi vrsti. Iskana najmanjša razdalja med osnovnicama je enaka največji od teh medsebojnih razdalj.

V mislih postavimo obe vrsti pravokotnikov eno nad drugo tako, da se obe vrsti začenjata na x-koordinati 0. Naj bo $g(x)$ globina zgornje vrste pravokotnikov pri koordinati x, $h(x)$ pa višina spodnje vrste pri koordinati x. Če nočemo, da se bosta vrsti pri tej x-koordinati prekrivali, morata biti njuni osnovnici razmaknjeni vsaj za $g(x) + h(x)$. To mora zdaj veljati pri vseh x, zato je najmanjši primerni razmik kar $\max_x(g(x) + h(x))$. Pri tem je dovolj, če gre x od 0 do širine krajše od obeh vrst — od tam naprej ostane le še ena vrsta in se torej ne more prekrivati z drugo.

Vrednost funkcije $g(x)$ se spremeni le, ko pridemo v zgornji vrsti do naslednjega pravokotnika, vrednost $h(x)$ pa, ko pridemo do naslednje v spodnji vrsti. Zato je dovolj, če izračunamo $g(x) + h(x)$ le pri vsakem takem x, ko se je katera od njiju spremenila, in poiščemo maksimum tako dobljenih vsot. Spodnji program se v zanki premika desno po obeh vrstah; na začetku vsake ponovitve velja, da smo v zgornji vrsti pregledali že prvih ia pravokotnikov (njihova skupna širina pa je SirinaA), v spodnji pa prvih ib pravokotnikov (s skupno širino SirinaB). Naslednja sprememba funkcije $g(x)$ torej nastopi pri $x =$ SirinaA, naslednja sprememba $h(x)$ pa pri $x =$ SirinaB. Zato pogledamo manjšo od teh dveh vrednosti; če je to prva, se premaknemo naprej po zgornji vrsti pravokotnikov (povečamo ia), sicer pa po spodnji (povečamo ib). V vsakem primeru je vrednost $g(x)$ enaka a[ia].Globina, vrednost $h(x)$ pa b[ib].Visina in njuna vsota (Razdalja) je nova kandidatka za najmanjši sprejemljivi razmik med osnovnicama (ki ga bomo pripravili v DosedanjaRazdalja).

```
const VrstaM = 999;
type
  SkatlaT = record Visina, Globina, Sirina: integer end;
  VrstaT = array [1..VrstaM] of SkatlaT;
```

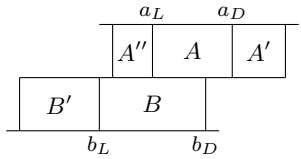
```
function NajmanjsaRazdalja(var a: VrstaT; aL: integer;
                           var b: VrstaT; bL: integer): integer;
{ Funkcija vrne najmanjšo razdaljo med sosednjima vrstama pravokotnikov,
  pri kateri se pravokotniki obeh vrstic še ne prekrivajo med seboj. }
var
  Konec: boolean;
  ia, ib: 1..VrstaM; { indeksa trenutno pregledovanega pravokotnika v obeh vrsticah }
  SirinaA, SirinaB: integer; { doslej pregledani dolžini obeh vrstic }
  StaraSirinaA: integer;
  Razdalja: integer; { globina gornjega pravokotnika + višina spodnjega
                       pravokotnika v trenutni točki }
  DosedanjaRazdalja: integer; { največja razdalja doslej }
begin
  if (aL = 0) or (bL = 0) then begin NajmanjsaRazdalja := 0; exit end;
  ia := 1; ib := 1; Konec := false;
  SirinaA := a[ia].Sirina; SirinaB := b[ib].Sirina;
  DosedanjaRazdalja := a[ia].Globina + b[ib].Visina;
  repeat
    StaraSirinaA := SirinaA;
    if SirinaA <= SirinaB then begin
      if ia >= aL then Konec := true
      else begin ia := ia + 1; SirinaA := SirinaA + a[ia].Sirina end;
    end; {if}
    if SirinaB <= StaraSirinaA then begin
      if iB >= bL then Konec := true
      else begin ib := ib + 1; SirinaB := SirinaB + b[ib].Sirina end;
    end; {if}
    if not Konec then begin
      Razdalja := a[ia].Globina + b[ib].Visina;
      if Razdalja > DosedanjaRazdalja then
        DosedanjaRazdalja := Razdalja;
    end; {if}
  until Konec;
  NajmanjsaRazdalja := DosedanjaRazdalja;
end; {NajmanjsaRazdalja}
```

Če je spodnja vrstica krajša od zgornje, bi se lahko zgodilo, da bi kak zelo globok pravokotnik na koncu zgornje vrstice štrlel v globino še pod osnovnico druge vrstice. Podobno se seveda lahko zgodi, če je zgornja krajša od spodnje in je pri koncu spodnje nek zelo visok pravokotnik. Če nas to moti (npr. ker nam zapleta nadaljnje delo, če bi hoteli zdaj pod drugo vrstico položiti še tretjo in tako naprej), bi morali krajšo vrstico podaljšati s še enim „navideznim“ pravokotnikom, ki bi imel višino in globino 0, širino pa tolikšno, da bi se širini obeh vrstic potem ujemali.

Zanimiv problem nastane, če dopustimo še premikanje vrst levo in desno. V nekaterih primerih je potem mogoče vrstici še bolj približati.

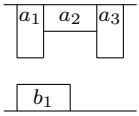
Recimo, da bo zgornja vrsta pravokotnikov pri miru in se bo začela vedno pri $x = 0$; spodnja vrsta pa naj se recimo začne pri $x = t$ za različne vrednosti t. Recimo, da se dva pravokotnika (eden iz gornje in eden iz spodnje vrstice) „stikata“, če pokrivata kakšen skupen interval x-koordinat. Množico vseh stikov, do katerih pride pri nekem konkretnem položaju spodnje vrstice (torej pri nekem konkretnem t), označimo s $S(t)$. Naj bo „razmik“ posameznega stika s, recimo mu $r(s)$, vsota globine zgornjega in višine spodnjega pravokotnika, ki se tu stikata. Če fiksiramo t (torej levi rob spodnje vrstice), je najmanjši potrebni razmik med osnovnicama obeh vrstic kar $R(t) := \max_{s \in S(t)} r(s)$. Nas zanima najmanjša vrednost $R(t)$ po vseh t z nekega intervala (verjetno je omejitev vsaj ta, da se mora interval x-koordinat, ki jih pokriva spodnja vrstica, vsaj deloma prekrivati s tistim, ki ga pokriva zgornja vrstica, saj drugače vrstici druga drugi sploh ne bi bili v napoto in bi bil problem trivialen). Kot vidimo iz formule za $R(t)$, je dovolj, če pogledamo le tiste t, pri katerih pride v množici $S(t)$ do kakšne spremembe — torej takrat, ko nastane kakšen nov stik ali pa se kakšen stari izgubi. Koristno je pregledovati t-je sistematično od manjših proti večjim, saj so tako spremembe v množici $S(t)$ vedno majhne in postopne in se ni potrebno vsakič znova ukvarjati z vsemi stiki.

Naj bo $n(s)$ najmanjša naslednja vrednost zamika t, pri kateri stik s doživi kakšno spremembo. Za to je nekaj možnosti, vzamemo pa najmanjšo med njimi. Označimo trenutno vrednost t-ja s t_0. Recimo, da zgornji pravokotnik (recimo mu A) stika s pokriva na x-osi interval $[a_L, a_D]$, spodnji (recimo mu B) pa $[b_L, b_D]$ (glej sliko). (1) Vsekakor bo prišlo do spremembe pri $t = t_0 + (a_D - b_L)$, ko bo stik s sploh izginil. (2) No, če je $b_D < a_D$, bo nastopila sprememba tudi pri $t = t_0 + (a_D - b_D)$: pojavil se bo nov stik med B in med A-jevim naslednikom v zgornji vrstici (razen če je A zadnji v zgornji vrstici). (3) Še ena možnost je, če je $b_L < a_L$ in B ni prvi pravokotnik spodnje vrstice; tedaj bo pri $t = t_0 + (a_L - b_L)$ nastal nov stik, in sicer med A-jem ter B-jevim predhodnikom v spodnji vrstici. (Pri tej možnosti je dovolj, če jo upoštevamo le takrat, ko je A prvi v svoji vrstici; pri ostalih bomo ustrezne stike dobili že zaradi točke (2).) — Vrednost $n(s)$ bo torej najmanjša izmed teh treh možnosti (oz. kolikor jih je pri tem konkretnem s v resnici mogočih).



(1) Ko se bo spodnja vrstica premaknila desno za $a_D - b_L$ enot, bo stik (A, B) izginil.
(2) Ko se bo premaknila za $a_D - b_D$ enot, bo nastal nov stik (A', B).
(3) Ko se bo premaknila za $a_L - b_L$ enot, bo nastal nov stik (A, B'). Toda če obstaja pravokotnik A'', bomo novi stik (A, B') opazili tudi pri delu s stikom (A'', B').

Ko smo torej končali z delom pri trenutnem t, je naslednji, s katerim se bo treba ubadati, $t' := \min_{s \in S(t)} n(s)$. Da bomo lahko to vrednost učinkoviteje določili, je koristno vrednosti $n(s)$ za vse trenutne stike (torej vse $s \in S(t)$) hraniti v kopici, tako da je najmanjša med njimi vedno pri roki.[5] Tako torej ni težko ugotoviti, za koliko naj povečamo t; nato pa si moramo ogledati stik, ki je ta premik povzročil, in ga po potrebi zbrisati in/ali ustvariti kakšen nov stik, pač v skladu z razmislekom iz prejšnjega odstavka. Paziti moramo še na možnost, da ima več dotedanjih stikov isto vrednost $n(s)$ in moramo torej ob premiku t-ja poskrbeti za spremembe pri vseh teh stikih. S tem bomo po premiku t-ja primerno popravili oz. ažurirali tudi množico $S(t)$. Nato nas bo seveda zanimal minimalni potrebni razmik med osnovnicama vrstic pri tem t-ju, torej vrednost $\max_{s \in S(t)} r(s)$; da nam ne bo treba iti po vseh stikih, je koristno imeti tudi te vrednosti v neki kopici (in to tako, da bo največja vedno pri vrhu). Pomembno je, da od vseh sprememb, ki nastopijo pri prehodu na novi t, najprej izvedemo brisanja stikov, nato izračunamo $\max_{s \in S(t)} r(s)$ in šele nato dodamo nove stike. Brez tega namreč ne bi mogli izkoristiti nekaterih bolj tesnih sestavitev pravokotnikov (primer je na spodnji sliki: brisanje stika (a_1, b_1) nastopi pri istem t kot dodajanje stika (a_3, b_1); če torej ne pogledamo $\max_{s \in S(t)} r(s)$ med tem brisanjem in tem dodajanjem, ne bomo opazili, da lahko pravokotnik b_1 postavimo v vdolbino med a_1 in a_3 (in pod a_2)). Zato si v spodnji psevdokodi pomagamo s pomožnim seznamom D.



Kakšna je časovna zahtevnost tega postopka? Če pregledamo vse možne vrednosti t, pride vsak pravokotnik spodnje vrstice z vsakim pravokotnikom zgornje prej ali slej vsaj za nekaj časa v stik. Če je v vsaki vrstici N pravokotnikov, imamo torej vsega skupaj $O(N^2)$ stikov. Vsak stik lahko, kot smo videli ob razmišljanju o vrednosti $n(s)$, največ trikrat povzroči spremembe, torej bo od nas zahteval le konstantno število operacij v kopici; vsaka od teh operacij pa ima časovno zahtevnost $O(\log N)$, saj v vsakem trenutku obstaja le $O(N)$ stikov (če gremo po x-osi od leve proti desni, pride do novega stika natanko tam, kjer se začne ali konča kak pravokotnik v zgornji in/ali v spodnji

[5] Bralec, ki mu kopice niso domače, si lahko namesto kopice misli tudi navaden seznam (ali tabelo), v katerem je ob vsakem stiku napisana še njegova vrednost $n(s)$. Pri takem seznamu ni težko dodajati in brisati elementov, pa tudi poiskati takega z najmanjšo vrednostjo $n(s)$; težava je le v tem, da pri seznamu vsaj kakšna od teh operacij zahteva v najslabšem primeru linearen sprehod po celem seznamu. Kopica pa je podatkovna struktura, ki podpira prav takšne operacije, vendar bolj učinkovito — v času $O(\log N)$, če je N število elementov v kopici.

vrstici; in ker imamo $O(N)$ pravokotnikov, je tudi le toliko stikov). Za celoten algoritem tako porabimo $O(N^2 \log N)$ časa.

Vhod: seznama pravokotnikov $a = \langle A_1, \ldots, A_{|a|} \rangle$ in $b = \langle B_1, \ldots, B_{|b|} \rangle$.
Pomožni funkciji: $r(i,j)$ = globina pravokotnika A_i + višina B_j
$n(i,j,t)$ (opisana spodaj).

$t :=$ −skupna širina vseh pravokotnikov seznama b;
$K_n :=$ prazna kopica (manjše vrednosti bodo pri vrhu);
$K_r :=$ prazna kopica (večje vrednosti bodo pri vrhu);
dodaj stik $(1, |b|)$ v obe kopici;
$M := r(1, |b|)$;
$D :=$ prazen seznam;
while K_n ni prazna **do begin**
    $t' :=$ najmanjša vrednost iz K_n;
    if $t' > t$ **then begin**
        $M := \min\{M,$ največja vrednost iz $K_r\}$;
        dodaj vse stike iz D v obe kopici;
        $D :=$ prazen seznam; $t := t'$;
    end;
    $(i,j) :=$ stik z najmanjšo vrednostjo v K_n;
    $a_L := t +$ (skupna širina $A_1, \ldots, A_{i-1}$); $a_D := a_L +$ širina A_i;
    $b_L := t +$ (skupna širina $B_1, \ldots, B_{j-1}$); $b_D := b_L +$ širina B_j;
    if $b_D = a_D$ **and** $i < |a|$ **then** dodaj $(i+1, j)$ v D;
    if $b_L = a_L$ **and** $i = 1$ **and** $j > 1$ **then** dodaj $(i, j-1)$ v D;
    if $b_L = a_D$ **then** zbriši (i,j) iz obeh kopic
    else spremeni vrednost stika (i,j) v K_n na $n(i,j,t)$;
end;
vrni M;

Ob dodajanju novega stika v kopici uporabimo kot vrednost tega stika v kopici K_n vrednost $n(i,j,t)$, v kopici K_r pa vrednost $r(i,j)$. Vsote širin pravokotnikov od A_1 do A_{i-1} si naračunamo enkrat samkrat, na začetku, za vse i; podobno tudi za $B_1, \ldots, B_{j-1}$ za vse j.

Funkcija $n(i,j,t)$:
    izračunaj a_L, a_D, b_L, b_D enako kot zgoraj;
    $n := t + a_D - b_L$; (pri takem t bo treba ta stik zbrisati)
    if $(b_D < a_D)$ **and** $(i < |a|)$
    then $n := \min\{n, t + a_D - b_D\}$; (takrat bo treba dodati stik $(i+1, j)$)
    if $(b_L < a_L)$ **and** $(i = 1)$ **and** $(j > 1)$
    then $n := \min\{n, t + a_L - b_L\}$; (takrat t bo treba dodati stik $(i, j-1)$)
    vrni n;

Doslej opisani postopek dovoli za t vse take vrednosti, pri katerih imata obe vrstici sploh kaj skupnega (torej take, pri katerih obstaja vsaj en stik). Če pa je dovoljen le nek manjši interval t-jev, recimo od t_1 do t_2, moramo postopek malo dopolniti. Množico stikov, ki so v veljavi pri $t = t_1$, lahko določimo s postopkom, ki je čisto podoben podprogramu NajmanjsaRazdalja z začetka te rešitve. Po vsakem povečanju t-ja (v stavku $t := t'$) pa moramo pogledati, če ni zdaj večji od t_2; če je, moramo prekiniti izvajanje glavne zanke.

Oglejmo si še malo preprostejšo različico doslej opisanega postopka. Stik med pravokotnikom $A = [a_L, a_D]$ iz zgornje vrstice in $B = [b_L, b_D]$ iz spodnje vrstice je prisoten natanko tedaj, ko se spodnja vrstica začne pri nekem $x = t$, za katerega velja $t + b_L < a_D$ (drugače bi bil B v celoti desno od A) in $t + b_D > a_L$ (drugače bi bil B v celoti levo od A). Stik $s = (A, B)$ je torej prisoten od $t = a_L - b_D$ do $t = a_D - b_L$; recimo tema dvema vrednostma „leva“ in „desna“ meja stika s. Za vse možne pare (A, B) lahko izračunamo obe meji, vse te meje zložimo v en sam dolg seznam in jih uredimo naraščajoče (če je več enakih mej, naj pridejo desne pred levimi); pri vsaki meji si tudi zapomnimo, kateremu stiku pripada in ali je leva ali desna. Potem se moramo le sprehoditi po tako urejenem seznamu od začetka do konca; ko se premaknemo mimo neke leve meje, moramo njen stik dodati v kopico K_r, ko se premaknemo mimo neke desne meje, pa moramo njen stik pobrisati iz K_r. Po vsaki taki spremembi pogledamo, kakšen razmik zahteva trenutna množica stikov, med vsemi tako dobljenimi razmiki pa si zapomnimo najmanjšega. Lepo pri tej različici rešitve je, da se nam ni več treba ubadati z vzdrževanjem kopice K_n in z razmišljanjem o tem, kdaj je treba vanjo kaj dodati; časovna zahtevnost je še vedno $O(N^2 \log N)$, enako kot doslej; slabost te različice s seznamom pa je, da imamo v seznamu podatke za vse stike, torej zasede $O(N^2)$ prostora, kopica K_n pri prejšnji rešitvi pa je vsebovala le $O(N)$ stikov in je torej zasedla le $O(N)$ prostora. V obeh različicah pa lahko uporabljeno tehniko gledamo kot primer „pometanja“ ali preleta ravnine (*plane sweep*), ki pride prav pri mnogih geometrijskih problemih (gl. npr. nalogo 1998.2.3 in njeno rešitev).

N: 6

R1991.2.3

Za vsak možen znak (od 0 do 9) pogledamo, na koliko mestih se razlikuje od prebranega (to prešteje podprogram Razlika). V spremenljivki MinRazl si zapomnimo razliko do doslej najpodobnejšega najdenega znaka, NajblizjiZnak pa pove, kateri znak je to bil. Če je novi znak še bližji, si ga zapomnimo kot novega najbližjega; če pa se razlikuje od prebranega za prav toliko kot doslej najbližji, postavimo NajblizjiZnak na presledek, kajti če ne bomo našli kakšnega še bližjega znaka, se ne bomo mogli enolično odločiti za najbližji znak in bomo morali zato vrniti presledek.

```
type ZnakT = array [1..7] of boolean;
var OblikeZnakov: array ['0'..'9'] of ZnakT;

function DolociZnak(zn: ZnakT): char;    { Vrne najbolj podoben znak podani obliki }
```

```
var                                  { ali pa presledek, če najde več enakovrednih kandidatov. }
  c, NajblizjiZnak: char;
  Razl, MinRazl: integer;

  function Razlika(var zn1, zn2: ZnakT): integer;
  { Funkcija vrne število segmentov, v katerih se razlikujeta dva znaka. }
  var i, Razl: integer;
  begin {Razlika}
    Razl := 0;
    for i := 1 to 7 do
      if zn1[i] <> zn2[i] then Razl := Razl + 1;
    Razlika := Razl;
  end; {Razlika}

begin {DolociZnak}
  NajblizjiZnak := ' '; MinRazl := 8;
  for c := '0' to '9' do begin
    Razl := Razlika(zn, OblikeZnakov[c]);
    if Razl < MinRazl then
      begin MinRazl := Razl; NajblizjiZnak := c end
    else if Razl = MinRazl then
      NajblizjiZnak := ' '; { Ne moremo enolično določiti znaka. }
  end; {for}
  DolociZnak := NajblizjiZnak;
end; {DolociZnak}
```

R1991.2.4

N: 6

Naloga ima seveda precej bistveno različnih rešitev. Ena od možnih rešitev, ki je podobna v praksi znanemu protokolu SDLC ali HDLC, je na kratko naslednja:

Za začetni znak si izberemo npr. zaporedje bitov 01111110. Pred vsakim paketom oddamo najprej ta znak, bit 1, ki označuje, da gre za veljaven paket, nato oddajamo zaporedne bite paketa in na koncu spet ta znak. Med paketi oddajamo same ničle. Če se v oddanem zaporedju bitov pojavi zaporedje bitov 0111111, pred naslednjim bitom oddamo vrinjeni bit 1, nato pa nadaljujemo z oddajanjem podatkovnih bitov.

Ko na sprejemni strani sprejmemo zaporedje 0111111, počakamo še na naslednji bit. Če je ta bit 0, nam bit za njim pove, ali gre za začetek novega paketa (1) ali pa je to konec prejšnjega paketa (0). Če pa je bil naslednji (osmi) bit 1, ga zavržemo in sprejemamo nadaljevanje paketa.

Za ta postopek potrebujemo na sprejemni strani samo nekaj bitov pomnilnika (če začnemo sprejemati začetni znak, ga moramo hraniti, dokler ne ugotovimo, ali gre za pravi začetni znak ali pa bo imel vrinjeno enico). Šele ko sprejmemo vrinjeno enico, smemo zadržane bite zares razglasiti za sprejete podatke.

Učinkovitost tega postopka (razmerje med količino informacijskih bitov

in skupno količino prenesenih bitov) pri paketih dolžine n bitov in povsem slučajno porazdeljenih podatkovnih bitih je približno $n/(17 + n + n/256) = 256n/(257n + 17)$. Pri $n = 1000$ je to 97,95 % in pri $n = 10000$ že 99,44 %.

Če bi si izbrali začetni znak drugačne dolžine, bi lahko učinkovitost pri daljših blokih še poljubno povečali, seveda pa bi zato potrebovali nekaj več bitov pomnilnika na sprejemni strani.

Zahteva o majhni porabi pomnilnika v tej nalogi ni umetna, saj tak prenos delajo običajni sinhroni vmesniki, kjer pomnilnika navadno res nimamo veliko. Zaporedje bitov v našem primeru je navadno kar zaporedje bytov (čeprav ne nujno), zato je izbira osembitnega začetnega znaka zelo naravna. Hkrati nam pakiranje v osembitne byte na sprejemni strani predstavlja natanko tisto zakasnitev pri sprejemu, ki je potrebna, da se lahko odločimo, ali smo sprejeli začetni znak ali pa smo sprejeli znak z vrinjeno enico. Po vrivanju enic ima tak postopek ime "bit stuffing".

REŠITVE NALOG ZA TRETJO SKUPINO

N: 6

R1991.3.1

Podprogram deli prvi parameter z drugim. V resnici oponaša pisno deljenje. Poglejmo ga še enkrat, tokrat z ustreznimi komentarji.

```
function KajStorim(a, b: byte): byte;
{ Funkcija vrne kvocient a/b dveh nepredznačenih 8-bitnih števil. }
var
  Stevec: byte;
  Rezultat: byte; { Tu bomo sestavljali rezultat. }
  Premik: byte;   { Tu zbiramo številke levo od označenega mesta. }
begin
  Rezultat := 0; Premik := 0;
  for Stevec := 1 to 8 do begin
    Premik := 2 * Premik + (a div 128); { Zgornji bit a dodamo v Premik. }
    a := (a mod 128) * 2;          { Premaknemo bite v levo. }
    Rezultat := 2 * Rezultat;      { V rezultat dodamo v spodnji bit ničlo. }
    if Premik >= b then begin      { Če je dobljeno število že večje od delitelja, }
      Premik := Premik - b;        { ga zmanjšamo za delitelj. }
      Rezultat := Rezultat + 1;    { Ničlo v rezultatu spremenimo v ena. }
    end; {if}
  end; {for}
  KajStorim := Rezultat;
end; {KajStorim}
```

Razlika v primerjavi s pisnim deljenjem, kot smo ga sicer navajeni, je le v tem, da delamo tu v dvojiškem zapisu, ne pa v desetiškem. Zato se nam ni treba vprašati, *kolikokrat* gre delitelj b v Premik, pač pa le, ali sploh gre ali ne.

Če gre, dodamo na konec količnika števko 1, sicer pa 0. Dodajanje števke na konec pravzaprav pomeni, da dotedanjo vrednost količnika pomnožimo z 2 in prištejemo novo števko. Na vsakem koraku moramo na koncu Premika pripisati še naslednjo števko (torej: naslednji bit) deljenca. Pri tem sproti skrbimo, da se ta vedno nahaja v najvišjem bitu spremenljivke a; tako lahko do nje pridemo z a **div** 128, nato pa jo v a postavimo na 0 (z a **mod** 128) in nato a pomnožimo z 2, tako da se vsi preostali biti zamaknejo za eno mesto navzgor in s tem na najvišje mesto pride naslednji bit deljenca.

Pri deljenju z 0 se gornji podprogram obnaša malo nenavadno: delitelj b gre zdaj vedno v Premik, zato količniku vedno pripišemo enico in tako ne glede na a dobimo rezultat 255.

Na tak način lahko realiziramo deljenje na procesorjih, ki ne znajo deliti. Za izvajanje operacij a **div** 128 in a **mod** 128, ki se pojavljata v gornjem podprogramu, namreč ne potrebujemo pravega deljenja, ampak le preproste operacije na bitih. Za a **div** 128 je dovolj že premik vrednosti a za sedem bitov v desno (ali pa preverjanje, če je bit 7 prižgan), za a **mod** 128 pa je dovolj, če v a ugasnemo bit 7.

R1991.3.2

N: 7

Pravilna rešitev naloge predstavlja centralni del sintaktičnega analizatorja za prolog. Razlika je le v tem, da prolog dopušča poleg infiksnih operatorjev (med argumentoma) tudi prefiksne (pred argumentom) in postfiksne (za argumentom). En operator je lahko hkrati vseh teh tipov in različne prioritete za vsak tip.

Izraz in njegove podizraze lahko v pomnilniku predstavimo z drevesi. Notranja vozlišča drevesa ustrezajo operatorjem, listi drevesa pa predstavljajo člene izraza.

Podprogram Term prebira izraz, dokler ne naleti na operator, ki veže dovolj šibko (prioriteta > MxOpPri). Prebrani del izraza predstavi z drevesom in v Root vrne kazalec nanj. (Za začetek zgradi drevo z enim samim vozliščem, ki predstavlja trenutno prebrani člen.) Koren tega drevesa mora biti najšibkejši operator v prebranem delu izraza; njegovo prioriteto si zapomnimo v spremenljivki MnOpPri. Če naletimo na nek še šibkejši operator, pomeni, da bo v resnici koren moral postati ta, doslej zgrajeno drevo pa bo le levo poddrevo tega novega korena. Njegovo desno poddrevo pa bo moralo pokrivati nadaljnji del izraza do naslednjega šibkejšega operatorja, tako da lahko to poddrevo zgradimo z rekurzivnim klicem.

WrtTerm le pravilno izpiše izraz in si pri tem spet pomaga z rekurzijo. Podprogramu Parse tako ni treba storiti drugega, kot da kliče Term z dovolj visoko vrednostjo parametra, da bo Term zanesljivo prebral ves izraz; potem je treba izraz le še izpisati.

Spodnji program tudi predpostavi, da ima na voljo nek podprogram Error, ki ga lahko kliče v primeru napak. Error naj bi izpisal kakšno sporočilo o napaki

in prekinil delovanje programa. V praksi bi bilo verjetno dobro poleg številke napake izpisati tudi, kje v vhodnem izrazu je prišlo do nje.

```
const
  MxPri = 1200;
type
  SymDef = (sClen, sOp, sKonec);
  PtExpDef = ↑ExpDef;
  ExpDef = record
             case Sym: SymDef of
               sClen: (Val: char);
               sOp: (Op: char; LExp, RExp: PtExpDef);
           end;
var
  Sym: SymDef;
  Ch: char;
  OpPri: integer;

  procedure Error(n: integer); external;
  procedure GetSym; external;

procedure Term(MxOpPri: integer; var Root: PtExpDef);
var
  MnOpPri: integer;
  OldRoot: PtExpDef;
begin {Term}
  if Sym <> sClen then Error(2); { Predpostavimo, da je trenutni simbol člen. }
  New(Root); Root↑.Sym := Sym; Root↑.Val := Ch;
  MnOpPri := 0; GetSym;
  while (Sym = sOp) and (OpPri <= MxOpPri) do begin
    OldRoot := Root; New(Root); Root↑.Sym := Sym;
    Root↑.Op := Ch; Root↑.LExp := OldRoot; MnOpPri := OpPri;
    GetSym; Term(MnOpPri, Root↑.RExp);
  end; {while}
end; {Term}

procedure WrtTerm(Root: PtExpDef);
begin
  if Root↑.Sym = sClen then Write(Root↑.Val)
  else begin
    Write('('); WrtTerm(Root↑.LExp); Write(Root↑.Op);
    WrtTerm(Root↑.RExp); Write(')');
  end; {if}
end; {WrtTerm}

procedure Parse;
var Root: PtExpDef;
begin
```

```
  GetSym; Term(MxPri, Root);
  if Sym <> sKonec then Error(3);
  WrtTerm(Root);
end; {Parse}
```

R1991.3.3

N: 8

Problem lahko rešimo rekurzivno. Če se vzorec začenja na nekaj znakov, različnih od `?` in `*`, se mora tudi niz začenjati na enake znake, sicer se ne ujemata. Ko v vzorcu naletimo na `*`, lahko nekaj (nič ali več) znakov niza preskočimo, nato pa se mora preostanek niza ujemati s preostankom vzorca, kar lahko preverimo z rekurzivnim klicem. Seveda vnaprej ne moremo vedeti, koliko znakov naj bi ustrezalo tisti zvezdici, tako da moramo preizkusiti vse možnosti (notranja zanka **repeat** v spodnjem programu). Če pa v vzorcu naletimo na `?`, preskočimo en znak niza in nadaljujemo normalno. Podprogram Ujemanje, ki se bo klical rekurzivno, mora reševati probleme oblike „ali se del niza od položaja Ni naprej ujema z delom vzorca od položaja Vi naprej?“.

```
type Niz = packed array [1..10] of char;

function Ujemanje(N, V: Niz; Nd, Vd: integer): boolean;

  function Poskus(Ni, Vi: integer): boolean;
  var Uspesno: boolean;
  begin
    if Vi > Vd then { Prišli smo do konca vzorca — ali tudi do konca niza? }
      Poskus := Ni > Nd
    else if V[Vi] = '*' then begin
      Ni := Ni - 1;
      repeat { Poskusimo z zvezdico pokriti 0, 1, 2, ... znakov niza. }
        Ni := Ni + 1; Uspesno := Poskus(Ni, Vi + 1);
      until Uspesno or (Ni > Nd);
      Poskus := Uspesno;
    end else if (Ni <= Nd) and ((V[Vi] = '?') or (V[Vi] = N[Ni])) then
      { Trenutni znak niza in trenutni znak vzorca se ujemata. }
      Poskus := Poskus(Ni + 1, Vi + 1)  { Pregledujmo naprej. }
    else
      { Trenutni znak vzorca ni niti zvezdica niti vprašaj, trenutni znak niza se
        z njim ne ujema; torej se vzorec V[Vi..Vd] ne ujema z nizom N[Ni..Nd]. }
      Poskus := false;
  end; {Poskus}

begin
  Ujemanje := Poskus(1, 1);
end; {Ujemanje}
```

Slabost takšne rešitve je, da se lahko pri hudobno sestavljenem vzorcu podprogram Poskus kliče zelo velikokrat. Vsaka zvezdica se lahko ujema z 0, 1, 2, 3

itd. znaki. Naš podprogram preizkuša te možnosti po vrsti in šele ko se mu pri eni zatakne, poskusi z naslednjo. Če je v vzorcu veliko zvezdic in je niz takšne oblike, da se ne da dovolj zgodaj ugotoviti, če smo z zvezdico pokrili premalo ali preveč znakov niza, se lahko zgodi, da bo skupno število vseh preizkušenih možnosti naraščalo eksponentno hitro v odvisnosti od števila zvezdic v vzorcu. V takih primerih bi lahko porabil naš podprogram nesprejemljivo veliko časa.[6]

To rešitev lahko izboljšamo, če opazimo, da Poskus niti ne spreminja niza ali vzorca niti ne dela s kakšnimi globalnimi spremenljivkami (z drugimi besedami: je brez „stranskih učinkov“), tako da je njegov rezultat odvisen le od tega, kakšna parametra Ni in Vi je dobil. Če ga torej večkrat kličemo z istima vrednostma teh dveh parametrov, bo dajal vsakič tudi enak rezultat. Torej je pametno, da si po prvem klicu z nekim parom (Ni, Vi) rezultat zapomnimo v neki tabeli in kasneje, če bi ga spet potrebovali, ne izvedemo novega rekurzivnega klica, pač pa vzamemo rezultat iz tabele. Zdaj se bo torej izvedel največ en klic za vsak par (Ni, Vi), teh pa je največ $\mathsf{Nd} \cdot \mathsf{Vd}$).

Namesto takega sprotnega shranjevanja rezultatov v tabeli (čemur pravijo tudi „memoizacija“) bi jih lahko računali tudi sistematično, saj vemo, v kakšnem vrstnem redu jih bomo potrebovali: lahko bi jih računali po padajočih Ni in pri vsakem od njih še po padajočem Vi. Potem tudi vidimo, da si v resnici ni treba zapomniti več kot ene vrstice tabele rezultatov (ko računamo za nek Ni, potrebujemo le rezultate za $\mathsf{Ni} - 1$ in razne možne Vi). Taki tehniki reševanja pravimo tudi „dinamično programiranje“.

Še ena majhna izboljšava bi bila, da pri zvezdici ne bi poskušali na vse možne načine preskočiti 0 ali več znakov, tako da se iz klica (Ni, Vi) zaredijo

[6] Oglejmo si konkreten primer. Recimo, da bi imeli niz `aa...a` (n `a`-jev) in vzorec `*a*a...*ab` (n zvezdic in `a`-jev ter nato `b`). Niz se seveda ne ujema z vzorcem, vendar mora naš program, da se o tem res prepriča, izvesti med drugim vse tiste rekurzivne klice podprograma Poskus, ki ne izvedejo sami nobenega rekurzivnega klica. To pa so tisti, ki jim od vzorca ostane le še `b`, in tisti, ki jim je niza že zmanjkalo (vidijo le še prazen niz), pri vzorcu pa trenutno vidijo črko `a`. No, do prve možnosti lahko pridemo le tako, da pri nobeni zvezdici ne preskočimo nobenega znaka, ker bo drugače v nizu zmanjkalo `a`-jev, še preden bomo v vzorcu prišli do `b`-ja; tak rekurziven klic je torej samo en. Do druge možnosti pa lahko pridemo na veliko načinov: če vidimo v vzorcu i-ti `a`, pomeni, da smo s prejšnjimi i zvezdicami preskočili vsega skupaj $n - i + 1$ črk `a` v nizu (ostalih $i - 1$ `a`-jev niza pa se je ujelo z dosedanjimi $i - 1$ `a`-ji iz vzorca). Na koliko načinov lahko razbijemo $n - i + 1$ `a`-jev na i kosov (lahko praznih)? To je tako, kot če bi med `a`-je vrivali $i - 1$ „ograjic“ `|`, ki bi ponazarjale meje med kosi; dobimo ravno vse nize dolžine n, ki jih sestavlja $n - i + 1$ `a`-jev in $i - 1$ ograjic `|`. Takih pa je $\binom{n}{i-1}$ (izmed n mest moramo izbrati $i - 1$ mest, na katerih bodo ograjice, drugod pa bodo `a`-ji). Skupno se torej na ta način izvede $\sum_{i=1}^{n} \binom{n}{i-1} = \sum_{i=0}^{n-1} \binom{n}{i}$ robnih rekurzivnih klicev. Če prištejemo k temu še tisti en klic, ki pride v vzorcu do `b`-ja, in upoštevamo, da je $1 = \binom{n}{n}$, dobimo vsoto $\sum_{i=0}^{n} \binom{n}{n}$; to je vsota števil v $(n+1)$-vi vrstici Pascalovega trikotnika in znaša, kot vemo, ravno 2^n. Torej bi se pri takem nizu in takem vzorcu izvedlo v našem programu 2^n takih rekurzivnih klicev podprograma Poskus, ki ne izvedejo sami naprej nobenega novega rekurzivnega klica; že samo zaradi njih (če sploh ne razmišljamo še o rekurzivnih klicih nad njimi) bi bila časovna zahtevnost našega postopka v tem primeru eksponentna v odvisnosti od dolžine niza.

klici (i, Vi + 1) za i od Ni do Nd; dovolj bi bilo že, če bi izvedli le klica (Ni, Vi + 1) (če zvezdica ne pokrije nobenega znaka niza) in (Ni + 1, Vi) (zvezdica za začetek pokrije trenutni znak niza, nato pa bo že rekurzivni klic razmislil o tem, ali mora pokriti še kaj več kot to).

R1991.3.4

N: 8

Pri hitrosti 1 MB/s in 256 znakov dolgih blokih potrebujemo približno 0,002 s za sam prenos podatkov enega bloka, poleg tega pa še dvakrat po 1/4 s čakalnega časa, da oddajni računalnik prejme potrditev sprejema bloka od sprejemnika. To pomeni, da je izkoristek komunikacijske linije manj kot pol odstotka, zato opisani preprosti protokol ni primerjen za komunikacijo na zelo velike razdalje.

Izkoristek bi lahko povečali z uporabo daljših blokov, vendar dolžine bloka zaradi verjetnosti napak ne smemo poljubno podaljševati (če se pojavi napaka kjerkoli v bloku, je treba ponoviti prenos celotnega bloka).[7]

Bolje je, da ne čakamo potrditve vsakega bloka posebej, ampak nadaljujemo s pošiljanjem zaporednih oštevilčenih blokov, obenem pa si poslane, a še ne potrjene bloke shranimo za primer, da bi jih bilo treba zaradi komunikacijske napake ponovno poslati.

Ko sprejemni računalnik uspešno prejme nek blok, pošlje oddajnemu računalniku zaporedno število tega uspešno prejetega bloka. Tako lahko oddajni računalnik sprosti pomnilnik za že potrjene bloke.

Če sprejemni računalnik naleti na napačen blok ali ugotovi, da številke blokov niso zaporedne (kakšen blok manjka), pošlje zahtevo za ponovitev vseh blokov od prvega napačnega dalje. Nato čaka na ponovitev zahtevanega bloka in njegovih naslednikov — morebitne bloke, ki še prihajajo za napačnim blokom, zavrže. Po ponovnem sprejemu zahtevanega bloka lahko s komunikacijo normalno nadaljuje.

S skrbnejšim protokolom bi lahko zahtevali ponovitev le napačnih blokov in izkoristili morebitne vmesne pravilne bloke, vendar zaporednost številk blokov olajša delo in omogoča odkrivanje manjkajočih blokov brez dodatnega časovnega nadzora.

V praksi moramo dopustiti možnost, da oddajni računalnik pošilja bloke hitreje, kot jih je sprejemni računalnik sposoben obdelovati. Zato je treba skupaj s potrditvami sprejema pošiljati tudi število paketov, ki jih je sprejemnik še sposoben sprejeti (ima zanje dovolj pomnilnika). Temu številu pravimo „kredit“. Oddajnik mora tedaj skrbeti, da nikoli ne pošlje več paketov, kot ima kredita.

[7] Poleg tega včasih ne bi radi čakali, da se nam bo nabralo dovolj podatkov za cel dolg blok, pa tudi ne bi radi pošiljali dolgega bloka z malo podatki (in veliko ničelnimi biti ali kakšno podobno nekoristno vsebino), ker potem traja dlje, da dobimo od sprejemnika kakršen koli odgovor (ker mora sprejemnik prevzeti in pregledati dolg blok namesto kratkega). Res pa je, da to ni tako pomembno, če imamo opravka s tako dolgimi zakasnitvami kot pri tej nalogi (četrt sekunde v vsako smer).

Običajno moramo dopustiti tudi možnost, da se kakšna potrditev izgubi ali da je zveza začasno prekinjena. Da prebrodimo to vrsto težav, je treba protokol dopolniti z največjimi čakalnimi časi na posamezen dogodek in z akcijami, ki so potrebne ob prekoračitvi teh časovnih omejitev.

Optimalna dolžina blokov je odvisna od zanesljivosti prenosa in dolžine dodatne informacije, ki je dodana vsakemu bloku. Če je verjetnost napake velika, moramo uporabljati kratke bloke, da vsaj nekateri ostanejo nepoškodovani. Pri majhni verjetnosti napake so primernejši daljši bloki, da pošiljanje dodatne informacije ni tako pogosto. (Če naš protokol v primeru napake zahteva ponovno pošiljanje vseh paketov od vključno tistega, pri katerem je prišlo do napake, pa je škoda tudi pri kratkih blokih velika, saj je treba ponoviti za vsaj pol sekunde prometa.)