2. tekmovanje IJS v znanju računalništva

Institut Jožef Stefan, Ljubljana, 31. marca 2007

Bilten

Bilten 2. tekmovanja IJS v znanju računalništva
Institut Jožef Stefan, 2007

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Primož Gabrijelčič, Uroš Jovanovič, Mark Martinec, Mojca Miklavec, Marjan Šterk, Mitja Trampuš, Miha Vuk, Klemen Žagar, Janez Brank.

Tisk: Tiskarna knjigoveznica Radovljica, d. o. o.
Naklada: 300 izvodov

Vprašanja, pripombe, komentarji, popravki ipd. v zvezi z biltenom so dobrodošli. Pišite nam na naslov `rtk-info@ijs.si`.

CIP — Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

371.27:004(497.4)

TEKMOVANJE IJS v znanju računalništva (2 ; 2007 ; Ljubljana)
Bilten [Elektronski vir] / 2. tekmovanje IJS v znanju računalništva, Ljubljana, 31. marca 2007 ; avtorji nalog Nino Bašić ... [et al.] ; uredil Janez Brank. — Ljubljana : Institut Jožef Stefan, 2007

Način dostopa (URL): `http://rtk.ijs.si/2007/rtk2007-bilten.pdf`

ISBN 978-961-6303-98-9
1. Bašić, Nino 2. Brank, Janez, 1979–
235861248

KAZALO

PREDGOVOR

Računalništvo ima dandanes mnogo vidikov — od zelo matematičnih preko praktičnih do čisto družbenih in sociooloških. Od samih začetkov v dvajsetem stoletju pa spremlja računalništvo morda najpomembnejši vidik, ki je algoritmično razmišljanje. Gre za to, kako dani problem opisati in razrešiti z jezikom, pri katerem bo postopek razdeljen na osnovne korake, ki jih zna računalnik učinkovito izvesti — tak postopek imenujemo algoritem, zapišemo pa ga največkrat kar kot program, ki se izvaja na računalniku. Računalniško tekmovanje, čigar zbornik je pred nami, služi predvsem razvijanju postopkovnega razmišljanja, ki omogoča reševanje problemov na računalniku. V zadnjih dveh desetletjih je seveda računalništvo postalo še vse kaj drugega kot le programiranje, za katero je potrebno algoritmično razmišljanje. Vendar, če podrobneje analiziramo, kakšne veščine si morajo uporabniki računalnikov razviti, da uspešno komunicirajo z računalnikom, spet vidimo, da gre za pretežno postopkovno ramišljanje. V tem smislu se naložba v prebiranje in reševanje nalog, ki so tudi v tem biltenu, zagotovo izplača ne glede na nadaljevanje poklicne poti.

Marko Grobelnik

STRUKTURA TEKMOVANJA

Tekmovanje poteka v treh težavnostnih skupinah. Tekmovalec se lahko prijavi v katerokoli od teh treh skupin ne glede na to, kateri letnik srednje šole obiskuje. Prva skupina je najlažja in je namenjena predvsem tekmovalcem, ki se ukvarjajo s programiranjem šele nekaj mesecev ali mogoče kakšno leto. Druga skupina je malo težja in predpostavlja, da tekmovalci osnove programiranja že poznajo; primerna je za tiste, ki se učijo programirati kakšno leto ali dve. Tretja skupina je najtežja, saj od tekmovalcev pričakuje, da jim ni prevelik problem priti do dejansko pravilno delujočega programa; koristno je tudi, če vedo kaj malega o algoritmih in njihovem snovanju.

V lažjih dveh skupinah traja tekmovanje tri ure; tekmovalci rešujejo naloge na papir, nato pa njihove odgovore oceni temovalna komisija. Naloge v teh dveh skupinah večinoma zahtevajo, da tekmovalec opiše postopek ali pa napiše program ali podprogram, ki reši določen problem. Pri pisanju izvorne kode programov ali podprogramov načeloma ni posebnih omejitev glede tega, katere programske jezike smejo tekmovalci uporabljati.

V tretji skupini pa rešujejo tekmovalci naloge na računalnikih, za kar imajo pet ur časa. Pri vsaki nalogi je treba napisati program, ki prebere podatke iz vhodne datoteke, izračuna nek rezultat in ga izpiše v izhodno datoteko. Programe se potem ocenjuje tako, da se jih na ocenjevalnem računalniku izvede na več testnih primerih, število točk pa je sorazmerno s tem, pri koliko testnih primerih je izpisal pravilni rezultat. (Podrobnosti točkovanja v 3. skupini so opisane na strani 19.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++ in java.

Nekaj težavnosti tretje skupine izvira tudi od tega, da je pri njej mogoče dobiti točke le za delujoč program, ki vsaj nekaj testnih primerov reši pravilno; če imamo le pravo idejo, v delujoč program pa nam je ni uspelo preliti (npr. ker nismo znali razdelati vseh podrobnosti, odpraviti vseh napak, ali pa ker smo ga napisali le do polovice), ne bomo dobili pri tisti nalogi nič točk.

Tekmovalci vseh treh skupin si lahko pri reševanju pomagajo z zapiski in literaturo, v tretji skupini pa tudi z dokumentacijo raznih prevajalnikov in razvojnih orodij, ki so nameščena na tekmovalnih računalnikih.

Na začetku smo tekmovalcem razdelili tudi list z nekaj nasveti in navodili (str. 7–8 za 1. in 2. skupino, str. 18–20 za 3. skupino).

Omenimo še, da so rešitve, objavljene v tem biltenu, večinoma obsežnejše od tega, kar na tekmovanju pričakujemo od tekmovalcev, saj je namen tukajšnjih rešitev pogosto tudi pokazati več poti do rešitve naloge in bralcu omogočiti, da bi se lahko iz razlag ob rešitvah še česa novega naučil.

Poleg tekmovanja v znanju računalništva smo organizirali tudi tekmovanje programov (ki je podrobneje predstavljeno na straneh 88–94) in tekmovanje v off-line nalogi (ki je podrobneje predstavljeno na straneh 95–102).

NASVETI ZA 1. IN 2. SKUPINO

Nekatere naloge so tipa **napiši program** (ali **napiši podprogram**), nekatere pa tipa **opiši postopek**. Pri slednjih ti ni treba pisati programa ali podprograma v kakšnem konkretnem programskem jeziku, ampak lahko postopek opišeš tudi kako drugače: z besedami (v naravnem jeziku), psevdokodo (glej spodaj), diagramom poteka itd. Glavno je, da je tvoj opis dovolj natančen, jasen in razumljiv, tako da je iz njega razvidno, da si dejansko našel in razumel pot do rešitve naloge.

Psevdokodi pravijo včasih tudi strukturirani naravni jezik. Postopek opišemo v naravnem jeziku, vendar opis strukturiramo na podoben način kot pri programskih jezikih, tako da se jasno vidi strukturo vejitev, zank in drugih programskih elementov.

Primer opisa postopka v psevdokodi: recimo, da imamo zaporedje besed in bi ga radi razbili na več vrstic tako, da ne bo nobena vrstica preširoka.

```
naj bo trenutna vrstica prazen niz;
pregleduj besede po vrsti od prve do zadnje:
    če bi trenutna vrstica z dodano trenutno besedo (in presledkom
    pred njo) postala predolga,
        izpiši trenutno vrstico in jo potem postavi na prazen niz;
    dodaj trenutno besedo na konec trenutne vrstice;
če trenutna vrstica ni prazen niz, jo izpiši;
```

Če pa v okviru neke rešitve pišeš izvorno kodo programa ali podprograma, obvezno poleg te izvorne kode v nekaj stavkih opiši, kako deluje (oz. naj bi delovala) tvoja rešitev in na kakšni ideji temelji.

Pri ocenjevanju so vse naloge vredne enako število točk. Svoje odgovore dobro utemelji. Prizadevaj si predvsem, da bi bile tvoje rešitve pravilne, ob tem pa je zaželeno, da so tudi čim bolj učinkovite (take dobijo več točk kot manj učinkovite). Za manjše sintaktične napake se načeloma ne odbije veliko točk. Priporočljivo in zaželeno je, da so tvoje rešitve napisane pregledno in čitljivo. Če je na listih, ki jih oddajaš, več različic rešitve za kakšno nalogo, jasno označi, katera je tista, ki naj jo ocenjevalci upoštevajo.

Če naloga zahteva branje ali obdelavo kakšnih vhodnih podatkov, lahko tvoja rešitev (če v nalogi ni drugače napisano) predpostavi, da v vhodnih podatkih ni napak (torej da je njihova vsebina in oblika skladna s tem, kar piše v nalogi).

Nekatere naloge zahtevajo branje podatkov s standardnega vhoda in pisanje na standardni izhod. Za pomoč je tu nekaj primerov programov, ki delajo s standardnim vhodom in izhodom:

- Program, ki prebere s standardnega vhoda dve števili in izpiše na standardni izhod njuno vsoto:

```
program BranjeStevil;
var i, j: integer;
begin
  ReadLn(i, j);
  WriteLn(i, ' + ', j, ' = ', i + j);
end. {BranjeStevil}
```

```
#include <stdio.h>
int main() {
  int i, j; scanf("%d %d", &i, &j);
  printf("%d + %d = %d\n", i, j, i + j);
  return 0;
}
```

- Program, ki bere s standardnega vhoda po vrsticah, jih šteje in prepisuje na standardni izhod, na koncu pa izpiše še skupno dolžino:

```
program BranjeVrstic;
var s: string; i, d: integer;
begin
  i := 0; d := 0;
  while not Eof do begin
    ReadLn(s);
    i := i + 1; d := d + Length(s);
    WriteLn(i, '. vrstica: "', s, '"');
  end; {while}
  WriteLn(i, ' vrstic, ', d, ' znakov.');
end. {BranjeVrstic}
```

```
#include <stdio.h>
#include <string.h>
int main() {
  char s[201]; int i = 0, d = 0;
  while (gets(s)) {
    i++; d += strlen(s);
    printf("%d. vrstica: \"%s\"\n", i, s);
  }
  printf("%d vrstic, %d znakov.\n", i, d);
  return 0;
}
```

Opomba: C-jevska različica gornjega programa predpostavlja, da ni nobena vrstica vhodnega besedila daljša od dvesto znakov. Funkciji gets se je v praksi bolje izogibati, ker pri njej nimamo zaščite pred primeri, ko je vrstica daljša od naše tabele s. Namesto gets bi bilo bolje uporabiti fgets; vendar pa za rešitev naših tekmovalnih nalog v prvi in drugi skupini zadošča tudi gets.

- Program, ki bere s standardnega vhoda po znakih, jih prepisuje na standardni izhod, na koncu pa izpiše še število prebranih znakov (ne vštevši znakov za konec vrstice):

```
program BranjeZnakov;
var i: integer; c: char;
begin
  i := 0;
  while not Eof do begin
    while not Eoln do
      begin Read(c); Write(c); i := i + 1 end;
    if not Eof then begin ReadLn; WriteLn end;
  end; {while}
  WriteLn('Skupaj ', i, ' znakov.');
end. {BranjeZnakov}
```

```
#include <stdio.h>

int main() {
  int i = 0, c;
  while ((c = getchar()) != EOF) {
    putchar(c); if (i != '\n') i++;
  }
  printf("Skupaj %d znakov.\n", i);
  return 0;
}
```

NALOGE ZA PRVO SKUPINO

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši, na kakšni ideji temelji tvoja rešitev.

1. Poraba goriva

Del opreme sodobnega avtomobila je tudi potovalni računalnik. Zanj skrbi samostojni program, ki zajema podatke iz avtomobilskega informacijskega sistema in jih v obdelani obliki prikazuje na zaslončku. V našem primeru računalnik zajame, obdela in prikaže podatke enkrat na sekundo.

Eden od teh podatkov je povprečna poraba goriva. Predstavlja količino goriva, ki bi ga ob trenutni porabi potrebovali za prevoz stokilometrske razdalje. Ker pa izmerjena poraba med vožnjo zelo niha, potrebujemo podprogram, ki bo na podlagi meritev izračunal povprečno porabo v zadnjih desetih sekundah.

Napiši podprogram PovprecnaPoraba za izračun povprečne porabe. Tvoj podprogram naj bo takšne oblike:

```
function PovprecnaPoraba(Poraba1sec, Pot1sec: real): real;        { v pascalu }
double PovprecnaPoraba(double Poraba1sec, double Pot1Sec);        /* v C-ju */
```

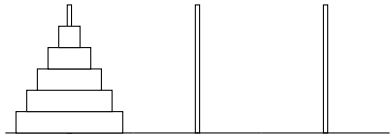
Glavni program potovalnega računalnika ga bo poklical vsako sekundo in mu podal dva podatka — porabo goriva v zadnji sekundi (v litrih) in prevoženo pot v zadnji sekundi (v metrih). (Ta dva podatka nista nikoli manjša od 0; tudi če se avtomobil na primer vozi vzvratno, je prevožena razdalja predstavljena s pozitivnim številom.) Tvoj podprogram mora vrniti povprečno porabo v zadnjih desetih sekundah, izraženo v porabljenih litrih goriva na 100 kilometrov. Če se avtomobil v zadnjih desetih sekundah sploh ni premaknil, naj tvoj podprogram vrne 0. Glavni program bo ta podatek prikazal na zaslonu potovalnega računalnika.

Če hočeš, lahko uporabiš tudi globalne spremenljivke. Deklariraj jih izven funkcijskega podprograma PovprecnaPoraba. Upoštevaš lahko, da so ob začetku delovanja vse globalne spremenljivke nastavljene na vrednost 0. Predpostavi, da je pred prvim klicem podprograma PovprecnaPoraba avtomobil miroval in ni trošil goriva.

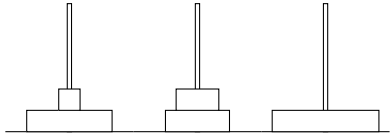
2. Hanojski stolpi

Hanojski stolpi so igra za enega igralca. Za igranje potrebujemo n preluknjanih okroglih ploščic različnih premerov ter tri palice, na katere natikamo ploščice. Na začetku igre nataknemo vse ploščice na prvo palico, in sicer tako, da so urejene po velikosti in je največja na dnu — dobimo nekakšno piramido. Cilj igre je, da celotno piramido prestavimo na tretjo palico, pri čemer moramo upoštevati pravila igre:

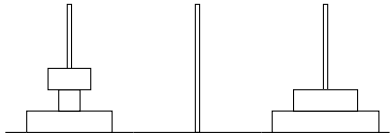
- V vsakem koraku smemo prestaviti le eno ploščico.
- Nikoli se ne sme zgoditi, da bi večja ploščica ležala na manjši.



Začetno stanje igre s 5 ploščicami.



Možno stanje med igro.



Prepovedano stanje.

Napiši program, ki prebere neko stanje igre (torej: neko razporeditev ploščic na palicah) in izpiše, ali je takšno stanje ploščic dovoljeno in ali je končno. Stanje je dovoljeno, če pri njem nobena ploščica ne leži na neki manjši ploščici. Stanje je končno, če je dovoljeno in so vse ploščice na zadnji (tretji) palici.

Stanje je opisano v štirih vrsticah: v prvi vrstici je število ploščic, recimo n (ki je najmanj 1 in največ 10), v naslednjih treh vrsticah pa so opisi palic. Za vsako palico piše najprej število ploščic na njej, nato pa še velikosti teh ploščic (najmanjša ploščica ima velikost 1, največja pa n); velikosti ploščic so navedene od spodaj navzgor — prva ploščica je tista, ki je na dnu palice, zadnja pa tista, ki je na vrhu. Predpostaviš lahko, da se vsaka ploščica (od 1 do n) pojavlja na natanko eni palici. Glede oblike izpisa ti ni treba posebej kompliciati; glavno je, da se bo iz tega, kar tvoj program izpiše, videlo, ali je stanje dovoljeno in ali je končno.

Nekaj primerov:

```
5
5 5 4 3 2 1
0
0
```

To je stanje z začetka igre (prva slika zgoraj).
To stanje je dovoljeno, ni pa končno.

```
5
2 4 1
2 3 2
1 5
```

To je stanje z druge slike zgoraj; je dovoljeno, ni pa končno.

```
5
3 4 1 2
0
2 5 3
```

To je stanje s tretje slike zgoraj; ni niti dovoljeno niti končno.

```
7
0
0
7 7 6 5 4 3 2 1
```

To stanje je dovoljeno in končno.

```
3
0
0
3 1 2 3
```

To stanje ni niti dovoljeno niti končno.

3. Ceste

Cestarji so pravkar dogradili več cest, ki vodijo iz nekega mesta v sosednje vasi. Na vsako cesto je potrebno na vsak kilometer postaviti tablico, ki označuje razdaljo od mesta. Tvoja naloga je **napisati program**, ki bo glede na število cest in njihove dolžine sestavil seznam potrebnih tablic. Vse ceste so dolge celo število kilometrov, tablic pa ne postavljamo na začetek in konec ceste.

Predpostaviš lahko, da že obstajata naslednji funkciji, ki ju lahko pokličeš, da dobiš podatke o novih cestah:

```
function StCest: integer;    { v pascalu }
int StCest();                /* v C-ju */
```

Vrne število novih cest.

```
function DolzinaCeste(i: integer): integer;    { v pascalu }
int DolzinaCeste(int i);                       /* v C-ju */
```

Vrne dolžino i-te nove ceste (v kilometrih), pri čemer mora biti i vsaj 1 in največ enak številu cest.

Primer: recimo, da imamo dve novi cesti, dolgi 4 km in 2 km. Potem potrebujemo naslednje tablice:

napis na tablici	število takih tablic
1 km	2
2 km	1
3 km	1

Z drugimi besedami, potrebujemo dve tablici z napisom „1 km“, eno tablico z napisom „2 km“ in eno tablico z napisom „3 km“. Glede oblike izpisa ti ni treba preveč kompliciratí, tvoj program naj se zgleduje kar po gornji tabeli.

4. Zlogovna pisava

Zlogovna pisava je pisava, v kateri posamezni znaki praviloma ne predstavljajo posameznih glasov, pač pa cele zloge. Vendar pa takšna pisava ponavadi ne vsebuje po enega znaka za čisto vsak zlog, ki se v določenem jeziku pojavlja, ker bi to zahtevalo neugodno veliko znakov. Pogosto se omejimo samo na vse tiste zloge, ki so sestavljeni iz enega soglasnika in enega samoglasnika (v tem vrstnem redu), na primer:

$$ta = \text{}, \; te = \text{}, \; pi = \text{}, \; ru = \text{}, \; sa = \text{}$$

in podobno. Pri pretvarjanju besed iz latinice v takšno zlogovno pisavo pa naletimo na problem: nekaterih besed se ne da razdeliti na zloge te oblike (torej take, ki so sestavljeni iz dveh črk, od katerih je prva soglasnik in druga samoglasnik); tistim, ki se jih da, bomo rekli *veljavne*, ostalim pa *neveljavne*. Besedi *kolo* in *teka* sta na primer veljavni, besede *kolut*, *tekma*, *in*, *obok* in *boa* pa ne.

Če hočemo neko neveljavno besedo vendarle zapisati s takšno zlogovno pisavo, se lahko zatečemo k vrivanju dodatnih črk. **Napiši podprogram** KolikoCrk:

```
function KolikoCrk(s: string): integer;      { v pascalu }
int KolikoCrk(char* s);                      /* v C-ju */
```

Ta podprogram naj za dano besedo `s` pove, kolikšno je najmanjše število dodatnih črk, ki bi jih bilo treba vriniti v to besedo, da bi postala veljavna. Primer: v besedo *oblak* je treba vriniti najmanj tri črke (soglasnik na začetek, samoglasnik na konec in še en samoglasnik med *b* in *l*).

Predpostaviš lahko, da bo niz `s`, ki ga bo kot parameter dobil tvoj podprogram, sestavljen samo iz malih črk angleške abecede (*a*, ..., *z*). Za samoglasnike veljajo črke *a*, *e*, *i*, *o*, *u*, ostale pa so soglasniki.

Opomba: če ne znaš drugače, lahko predpostaviš tudi, da *že obstaja* nek podprogram z imenom `JeSamoglasnik`, ki ti pove, ali je dani znak samoglasnik ali ne:

```
function JeSamoglasnik(c: char): boolean;    { v pascalu }
bool JeSamoglasnik(char c);                  /* v C-ju */
```

Če uporabiš tak podprogram, ne da bi si ga napisal sam, lahko dobiš pri tej nalogi največ polovico vseh možnih točk.

5. Enačbe

Opiši postopek, ki za nek dani niz preveri, če ta niz predstavlja veljavno enačbo. Enačba je veljavna natanko tedaj, ko ustreza naslednjim zahtevam:

- V enačbi smejo nastopati le števke ter znaka „+“ in „=“.
- Enačaj se mora pojavljati natanko enkrat. Levo in desno od njega mora biti v nizu vsaj po ena števka.
- Levo in desno od vsakega znaka „+“ mora biti vsaj po ena števka.

Primeri:

Nekaj nizov, ki predstavljajo veljavne enačbe:

```
1+2=456
12+34=40+3+2+01
123=143
12+3=1+23
123=103+20
0+00+000+00100=00000+000000
```

Nekaj nizov, ki ne predstavljajo veljavnih enačb:

```
1+2=3=2+1
12=3*4
a+1=1+a
12++3=15
tralala
1 + 2 = 3
c#
1+2+=3
=
+1+2=3
```

NALOGE ZA DRUGO SKUPINO

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši, na kakšni ideji temelji tvoja rešitev.

1. DKIM

Danes se pri branju e-pošte pogosto srečujemo s prevarami, pri katerih se pošiljatelj izdaja za nekoga drugega, na primer za neko finančno ustanovo, in skuša od prejemnika izvabiti osebne podatke ali denar. Prejemniku bi prišlo prav, če bi znal preveriti pristnost sporočila, kar je možno z uporabo digitalnega podpisa, ki je nekakšno število, izračunano iz vsebine besedila (in dodatno kriptografsko zaščiteno). Vsaka sprememba v besedilu povzroči v izračunanem številu spremembo (ki jo prepozna nek kriptografski postopek) in ponaredek je tako odkrit.

Pri programih za delo z e-pošto pa se včasih srečujemo z manjšimi spremembami v vsebini sporočil, ki jih ti programi hote ali nehote naredijo, denimo dodajanje ali odvzemanje praznih vrstic na koncu besedila ali dodajanje ali odvzemanje presledkov med besedami. Tudi tovrstne spremembe bi povzročile razveljavitev digitalnega podpisa, čemur pa bi se radi izognili. Prav te dni se zaključuje postopek standardizacije za predpis DKIM (DomainKeys Identified Mail), ki med drugim tudi določa postopek, s katerim besedilo pred podpisovanjem in preverjanjem najprej poenotimo tako, da odstranimo odvečne presledke in prazne vrstice na koncu besedila.

Napiši program, ki bo prepisal neko besedilo s standardnega vhoda (bere naj do konca standardnega vhoda, torej do EOF) na standardni izhod in ga pri tem predelal po naslednjih pravilih:

- več zaporednih presledkov zamenja z enim;
- presledke na koncu vrstic odstrani;
- morebitne prazne vrstice na koncu besedila zavrže; pri tem se za prazno vrstico šteje vsaka taka, ki vsebuje samo presledke, lahko tudi nobenega.

Predpostaviš lahko, da vrstice besedila niso nikoli daljše od 200 znakov.

2. Vsote

Minister Zverina se je odločil, da bo nagradil dva uspešna dolenjska športna kluba. Športniki se med sabo sicer ne marajo najbolj, vendar pa je minister odločen, da bo mednje prinesel spravo. Ministrovi sluge so sestavili dva seznama klubov tako, da noben klub s prvega seznama ne mara nobenega kluba z drugega in obratno. Za vsak klub poznamo tudi njegovo število članov. Za nagrado bo minister podaril potovanje, zaradi nesoglasij pa mora na potovanje poslati po en klub s prvega in en klub z drugega seznama. Za prevoz ima na razpolago vojaške osemkolesnike, ki imajo skupaj n sedežev.

Pomagaj ministru prinesti mir na Dolenjsko in **opiši** čim učinkovitejši **postopek**, ki pove, če je možno na potovanje poslati po en klub z vsakega seznama, tako da je

skupno število članov obeh klubov natanko n. Odkar so ljudje izvedeli za potovanje, so začeli ustanavljati nove klube kot gobe po dežju, zato upoštevaj, da je število klubov v obeh seznamih lahko zelo veliko.

3. Komparatorji

Programi za urejanje (sortiranje) datotek so običajno že priloženi operacijskemu sistemu, so učinkoviti in hitri, a včasih ne znajo preurediti vrstic besedila drugače kot leksikografsko — kar po kodah znakov privzete abecede.[1]

Včasih pa bi si želeli urediti vrstice v datoteki tudi kako drugače, denimo v obratnem vrstnem redu, ali po slovenski abecedi, ali pa upoštevati velike in male črke kot enakovredne. Da bi napisali lasten program za urejanje, je naporen posel, prav verjetno pa tudi ne bi bil tako učinkovit kot že napisani.

Lahko si pomagamo takole: za vsako vrstico (t.j. element urejanja) izračunamo nek nadomestni niz (ključ za urejanje) in ga pripnemo k vrstici (ne bomo se spustili v podrobnosti, kako to naredimo, lahko ga na primer vrinemo na začetek vsake vrstice). Poženemo program za urejanje in mu naročimo, naj pri ugotavljanju vrstnega reda vrstic primerja med seboj le pripete nadomestne nize (prestavlja pa z njimi vred tudi celotne vrstice). Na koncu urejevalne ključe zavržemo in delo je opravljeno.

Za primer poglejmo, kako bi lahko uredili besede po rimah — po črkah od zadaj naprej (tako imenovani odzadnji slovar, ki pride prav pesnikom):

Izvorna datoteka:

```
nagaja
hodi
raja
kazale
podgan
ban
iskale
blodi
```

Za vsako vrstico sestavimo in ji na začetku pripnemo niz, ki ga sestavljajo črke tiste besede v obratnem vrstnem redu, ter vse skupaj uredimo po pripetem nizu:

```
ajagan     nagaja
ajar       raja
elaksi     iskale
elazak     kazale
idoh       hodi
```

[1] *Opomba:* Leksikografski vrstni red pomeni, da so nizi urejeni najprej po številski vrednosti prvega znaka; če se jih več ujema v prvem znaku, se jih uredi po številski vrednosti drugega znaka in tako naprej. Prazen niz je leksikografsko pred vsakim drugim nizom. Primer nekaj nizov v leksikografskem vrstnem redu:

a, aa, aaa, aaaa, aaab, aab, az, aza, azz, b, ba, baa, baaa, bac, bba, bbb, wzz, z, zaaa, zzz, zzzzz.

To je, mimogrede, prav tak vrstni red, kakršnega uporabljata tudi operator $<$ za primerjanje nizov v pascalu ter funkcija strcmp za primerjanje nizov v C-ju. V pascalu nam številsko vrednost znaka `c` pove funkcija Ord(c), znak s številsko vrednostjo `k` pa dobimo s funkcijo Chr(k). V jezikih C/C++ lahko znake (spremenljivke tipa **char**) že same po sebi obravnavamo kot števila. Pri tej nalogi predpostavi, da imajo znaki številske vrednosti od vključno 1 do vključno 255. Pri tem imajo velike črke angleške abecede številske vrednosti od 65 (`'A'`) do 90 (`'Z'`), male črke od 97 (`'a'`) do 122 (`'z'`), števke od 48 (`'0'`) do 57 (`'9'`); presledek ima kodo 32. Na drugih kodah so ločila in razni posebni znaki.

```
idolb      blodi
nab        ban
nagdop     podgan
```

Na koncu še zavržemo pripeti niz in preostane nam slovar za pesnike (le drugi stolpec). Potrebovali smo le podprogram, ki iz nekega niza znakov (npr. „`nagaja`“) izračuna in vrne nov niz, ki predstavlja urejevalni ključ („`ajagan`“).

Napiši tak podprogram Kljuc:

```
function Kljuc(Z: string): string;      { v pascalu }
char* Kljuc(char* Z);                   /* v C-ju */
```

da bodo vrstice na koncu opisanega postopka urejene po naslednjih kriterijih (podnaloge so samostojne in med seboj neodvisne — za vsako napiši po en podprogram):

(a) urejene po obrnjenem abecednem vrstnem redu (predpostaviš lahko, da se v vrsticah pojavljajo le male črke angleške abecede);

(b) urejene po abecednem redu, vendar tako, da so velike in male črke med seboj enakovredne, presledki pa se ne upoštevajo (tako so denimo urejeni slovarji), primer: `a`, `Ana`, `ar`, `jo`, `joga`, `jo jo`, `joka` (predpostaviš lahko, da se v vrsticah pojavljajo le presledki in črke angleške abecede);

(c) urejene naraščajoče po dolžini; če je več vrstic enako dolgih, pa naj bodo med sabo urejene leksikografsko;

(d) vsaka vrstica vsebuje najmanj eno in največ osem števk (`0`, ..., `9`), ki tako predstavljajo neko celo število; rezultat urejanja naj bodo vrstice, urejene po naraščajočem vrstnem redu števil, ki jih predstavljajo, na primer: `1`, `10`, `0023`, `101`, `500`, `123456`;

Primer rešitve za odzadnji slovar:

```
function Kljuc(Z: string): string;
var K: string; i: integer;
begin
  K := '';
  for i := Length(Z) downto 1 do K := K + Z[i];
  Kljuc := K;
end; {Kljuc}

char* Kljuc(char* Z)
{
  int i, dolzina = strlen(Z);
  char *K = (char*) malloc(dolzina + 1);
  for (i = 0; i < dolzina; i++) K[i] = Z[dolzina - 1 - i];
  K[dolzina] = 0;
  return K;
}
```

4. Bančni računi

Neka banka uporablja naslednji podprogram, ki odobri in izvede dvig z računa ali pa ga zavrne:

```
    { Z računa z oznako Racun dvigni Znesek denarnih enot. Pri tem ne dovoli negativnega
      stanja na računu (torej da se dvigne več, kot je razpoložljivo na računu). }
 1  procedure Dvig(Racun: integer; Znesek: real);
 2  var TrenutnoStanje, NovoStanje: real;
 3  begin
      { Preveri, če je na računu dovolj denarja. }
 4    if StanjeNaRacunu(Racun) >= Znesek then begin
        { Drugim strežnikom prepreči hkratno izvajanje tega podprograma za ta račun. }
 5      ZakleniRacun(Racun);
        { Izračunaj novo končno stanje na računu. }
 6      TrenutnoStanje := StanjeNaRacunu(Racun);
 7      NovoStanje := TrenutnoStanje - Znesek;
 8      NastaviStanjeNaRacunu(Racun, NovoStanje);
        { Ponovno dovoli drugim strežnikom delo s tem računom. }
 9      OdkleniRacun(Racun);
10    end; {if}
11  end; {Dvig}
```

Ali, v C-ju:

```
    /* Z računa z oznako Racun dvigni Znesek denarnih enot. Pri tem ne dovoli negativnega
       stanja na računu (torej da se dvigne več, kot je razpoložljivo na računu). */
 1  void Dvig(int Racun, double Znesek)
 2  {
 3    double TrenutnoStanje, NovoStanje;
      /* Preveri, če je na računu dovolj denarja. */
 4    if (StanjeNaRacunu(Racun) >= Znesek) {
        /* Drugim strežnikom prepreči hkratno izvajanje tega podprograma za ta račun. */
 5      ZakleniRacun(Racun);
        /* Izračunaj novo končno stanje na računu. */
 6      TrenutnoStanje = StanjeNaRacunu(Racun);
 7      NovoStanje = TrenutnoStanje - Znesek;
 8      NastaviStanjeNaRacunu(Racun, NovoStanje);
        /* Ponovno dovoli drugim strežnikom delo s tem računom. */
 9      OdkleniRacun(Racun);
10    }
11  }
```

Podprogram StanjeNaRacunu(Racun) vrne trenutno stanje na računu z oznako Racun. Podobno podprogram NastaviStanjeNaRacunu(Racun, Stanje) nastavi novo stanje na danem računu.

Ker je banka velika, potrebuje več strežnikov, ki skrbijo za stanje na računu in na katerih bi se zgornji podprogram lahko izvedel. Izkušnje so pokazale, da ni priporočljivo, če dva strežnika hkrati spreminjata stanje na istem računu, zato so razvijalci vpeljali podprogram ZakleniRacun(Racun), ki deluje takole:

1. Če je kak drug strežnik klical podprogram ZakleniRacun za isti račun in ga še ni odklenil (s klicem OdkleniRacun), čaka, dokler drugi strežnik ne kliče OdkleniRacun.

2. V nasprotnem primeru se takoj vrne.

Račun je ob vrnitvi iz ZakleniRacun zaklenjen, dokler ga naš strežnik ne odklene s klicem OdkleniRacun.

Kljub tej skrbnosti pa podprogram Dvig še vedno lahko povzroči, da bo stanje na računu negativno, čeprav je v njem pogoj, ki naj bi to preprečeval. Naloga:

1. **Opiši scenarij**, ki pripelje do negativnega stanja na računu.
2. **Predlagaj**, kako dopolniti podprogram, da do tega ne bo moglo več priti.

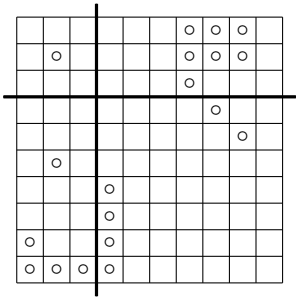
5. Šahovnica

Štirje otroci se igrajo neko strateško igrico, ki se odvija na kvadratni plošči z $n \times n$ polji. (Podobna je šahovnici, le da je veliko večja.) Pravil igre sicer ne poznamo in niso pomembna, vemo le to, da imajo na začetku vsi štirje igralci skupaj neko število figuric, ki do konca igre ostanejo na plošči; lahko se le dobro prerazporedijo. Na vsakem polju je največ ena figurica.

Problem je, da se otroci po koncu igre ne morejo dogovoriti, kako bodo pospravili figurice. Tukaj pa nastopi starejša sestra enega od otrok, ki velja za avtoriteto in odloči, da bo ploščo razdelila z dvema črtama (vodoravno in navpično) na 4 dele (ki niso nujno ploščinsko enaki). (Delitvena črta mora vedno potekati po meji med dvema vrsticama oz. stolpcema, ne pa npr. po sredi neke vrstice ali stolpca.) Želi si, da bi bila delitev kar se da pravična, tako da bi imel vsakdo na svojem delu plošče približno enako število figuric (čisto enakega števila figuric ni vedno moč doseči). Pri tem potrebuje tvojo pomoč.

Najbolj pravična delitev je tista, kjer imata območje z največ figurami in območje z najmanj figurami najmanjšo razliko v številu figur (po absolutni vrednosti). Označimo to vrednost s s.

Opiši postopek, ki bo za dano razporeditev figuric na plošči poiskal najpravičnejšo razdelitev in izpisal s in položaj delilnih črt pri tej razdelitvi. (Če je možnih več enako dobrih najpravičnejših razdelitev, torej takih z enakim s, je vseeno, katero od njih tvoj postopek poišče.)



Primer plošče z 19 figurami. Najpravičnejše možne delitve dosežejo $s = 6$ in ena od njih je označena z debelima črtama. Še dve enako dobri delitvi dobimo, če premaknemo vodoravno črto za eno vrstico navzgor ali pa navpično črto za en stolpec levo.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*.`in` in izpisujejo svoje rezultate v *imenaloge*.`out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil mapo s svojim uporabniškim imenom ter navodili, ki jih pravkar prebiraš. Ko boš sedel pred računalnik, boš dobil nadaljnja navodila za prijavo v sistem.

Na vsakem računalniku imaš na voljo enoto (disk) `U:`, na kateri lahko kreiraš svoje datoteke (datoteke, ki so tam že od prej, pusti pri miru). Programi naj bodo napisani v programskem jeziku Pascal, C, C++ ali Java, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal, GNU C/C++ in Sunovim prevajalnikom za Javo 1.5. Za delo lahko uporabiš `FP` oz. `ppc386` (FreePascal), `GCC`/`G++` (GNU C/C++ — command line compiler), `GCJ` (za javo 1.4) in Java 2 SDK (za javo 1.5).

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `RTK.EXE`, ki ga lahko uporabiš za preverjanje svojih rešitev. Tukaj si lahko tudi ogledaš anonimizirane rezultate ostalih tekmovalcev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka, brez šumnikov).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

`rtk` *imenaloge*.`pas`
`rtk` *imenaloge*.`c`
`rtk` *imenaloge*.`cpp`
`rtk` *ImeNaloge*.`java`

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Na spletni strani boš dobil za vsak testni primer obvestilo o tem, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku, razen s predpisano vhodno in izhodno datoteko. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku), prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi od 0 do 10 točk (praviloma 10, če je izpisal popolnoma pravilen odgovor, sicer pa 0; izjema je 5. naloga, kjer dobijo boljše rešitve več točk kot slabše), nato pa se te točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N - 1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

Izjema pri opisanem načinu točkovanja je 5. naloga, pri kateri se oddaja (glej opis naloge!) po eno izhodno datoteko za vsak testni primer. Za vsako dobiš od 0 do 10 točk. Skupno število točk pri tej nalogi dobimo tako, da za vsak testni primer upoštevamo najboljšo od izhodnih datotek, ki jih je tekmovalec oddal za ta primer. Število oddaj pri tej nalogi ne zmanjšuje števila točk.

Skupno število točk tekmovalca je vsota po vseh nalogah. Tekmovalce razvrstimo po skupnem številu točk.

Vsak tekmovalec se mora sam zase odločiti o tem, katerim nalogam bo posvetil svoj čas, v kakšnem vrstnem redu jih bo reševal in podobno. Verjetno je priporočljivo najprej reševati lažje naloge.

Poskusna naloga (ne šteje k tekmovanju) (`poskus.in`, `poskus.out`)

Napiši program, ki iz vhodne datoteke prebere dve celi števili (obe sta v prvi vrstici, ločeni z enim presledkom) in izpiše desetkratnik njune vsote v izhodno datoteko.

Primer vhodne datoteke:

```
123 456
```

Ustrezna izhodna datoteka:

```
5790
```

Primeri rešitev (dobiš jih tudi kot datoteke na `http://rtk/`):

```
program PoskusnaNaloga;
var T: text; i, j: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i, j); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * (i + j)); Close(T);
end.
```

```c
#include <stdio.h>
int main() {
  FILE *f = fopen("poskus.in", "rt");
  int i, j; fscanf(f, "%d", &i, &j); fclose(f);
  f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * (i + j));
  fclose(f); return 0;
}
```

```cpp
#include <fstream>
int main() {
  std::ifstream ifs("poskus.in"); int i, j; ifs >> i >> j;
  std::ofstream ofs("poskus.out"); ofs << 10 * (i + j);
  return 0;
}
```

```java
import java.io.*;
import java.util.Scanner;
public class Poskus {
  public static void main (String[] args) throws IOException {
    Scanner fi = new Scanner(new File("poskus.in"));
    int i = fi.nextInt(); int j = fi.nextInt();
    PrintWriter fo = new PrintWriter("poskus.out");
    fo.println(10 * (i + j)); fo.close();
  }
}
```

NALOGE ZA TRETJO SKUPINO

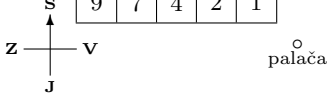
1. Nebotičniki (`neboticniki.in`, `neboticniki.out`)

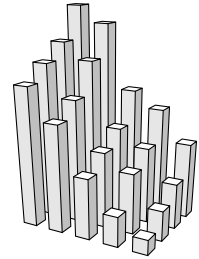
Dubajski emir bi rad zgradil novo mestno četrt samih nebotičnikov. V ta namen je že pripravil veliko pravokotno zemljišče in ga razdelil na parcele. Parcele tvorijo karirasto mrežo s h vrsticami in w stolpci. Na vsaki parceli bo stal po en nebotičnik. Ostane le še vprašanje, kako visok naj bo posamezni nebotičnik. Glede tega je emir postavil naslednje omejitve:

- Vsak nebotičnik mora biti visok vsaj c_1 nadstropij in kvečjemu c_2 nadstropij.
- Emir hoče imeti iz svoje palače lep razgled na novo četrt, zato ne mara, da bi mu nebotičniki preveč zakrivali pogled drug na drugega. Zato je uvedel predpis, da mora biti vsak nebotičnik vsaj za eno nadstropje nižji od svojega zahodnega soseda (če ga ima) in svojega severnega soseda (če ga ima).

Primer: recimo, da imamo $w = 5$, $h = 4$, $c_1 = 1$ in $c_2 = 20$. Zgornjim zahtevam lahko ugodimo na primer s takšnimi višinami nebotičnikov:

13	12	8	7	5
11	10	6	5	3
10	8	5	4	2
9	7	4	2	1





Kljub zgoraj naštetim omejitvam se lahko zgodi, da obstaja veliko različnih razporedov nebotičnikov po višini. Če imamo na primer $w = 3$, $h = 2$, $c_1 = 5$ in $c_2 = 9$, je možnih deset razporedov:

9	8	7	9	8	7	9	8	7	9	8	7	9	8	6
8	7	6	8	7	5	8	6	5	7	6	5	8	7	5
9	8	6	9	8	6	9	7	6	9	7	6	8	7	6
8	6	5	7	6	5	8	6	5	7	6	5	7	6	5

Napiši program, ki prebere števila w, h, c_1 in c_2 ter izpiše, koliko je možnih razporedov višin nebotičnikov, ki ustrezajo vsem opisanim zahtevam.

Vhodna datoteka: vsebuje le eno vrstico, v njej so cela števila w, h, c_1 in c_2 (v tem vrstnem redu, ločena s po enim presledkom). Veljalo bo: $1 \leq w \leq 10$, $1 \leq h \leq 10$, $1 \leq w \cdot h \leq 50$, $1 \leq c_1 \leq c_2 \leq 20$; pravilni odgovor, torej število možnih razporedov, bo pri vsakem izmed desetih testnih primerov, na katerih bomo preizkušali tvojo rešitev, manjši od 10^7.

Izhodna datoteka: vanjo izpiši število možnih razporedov višin nebotičnikov, ki ustrezajo vsem opisanim zahtevam.

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:
`3 2 5 9`	`10`

2. Pleskarji (`pleskarji.in`, `pleskarji.out`)

Na Discworldu, magičnem svetu, stoji ob glavnem poslopju čarovniške Univerze tudi najvišji stolp daleč naokoli. Dekan Mustrum se je odločil, da si bo prav na vrhu uredil prijetno razgledno sobico. Ker pa se za dekana spodobi, da dobi le najboljše, bo treba sobico najprej prepleskati.

Stolp je zelo visok: do vrha vodi n stopnic, $1 \leq n \leq 10^9$. Da bi bila sobica na vrhu čim hitreje prepleskana, pošljejo proti vrhu enega za drugim p pleskarjev, $1 \leq p \leq 10^6$. Pleskarji se ne prehitevajo: naslednji se odpravi na pot šele, ko prejšnji že pride na vrh. Težava je v tem, da so pleskarji pred tem že opravljali vsak svoj posel in imajo zato umazane čevlje, ki puščajo barvaste odtise na stopnicah, na katere pleskarji stopijo. Tako i-ti pleskar pušča odtise barve c_i (barve so predstavljene s celimi števili od 1 do 26).

Pleskarji so različno veliki, zato delajo različno dolge korake: i-ti pleskar stopi samo na vsako k_i-to stopnico, $1 \leq k_i \leq 10$. Vsi začnejo prav pri dnu, na „stopnici“ številka 0 (stopnice so oštevilčene s celimi števili od vključno 0 do vključno $n-1$); pleskar, za katerega je npr. $k_i = 3$, bo tako stopil na stopnice 0, 3, 6, 9 in tako naprej.

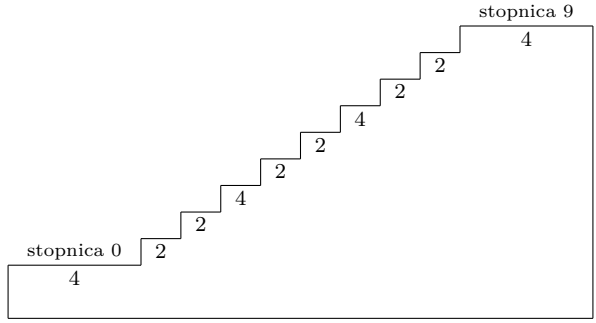
Napiši program, ki za vsako barvo od 1 do 26 ugotovi, koliko stopnic je popackanih z njo, ko se na vrhu končno zberejo vsi pleskarji (upoštevati je treba vse stopnice od vključno 0 do vključno $n-1$). Pri tem velja, da so barve povsem prekrivne: če na neko stopnico npr. stopi nekdo s čevlji barve a, pozneje pa še nekdo s čevlji barve b, se barve a ne vidi več in štejemo, da je opazovana stopnica popackana le z barvo b. Upoštevaj tudi, da so bile pred prihodom prvega pleskarja vse stopnice brezbarvne; če neke stopnice ne pohodi noben pleskar, bo brezbarvna tudi ostala.

Vhodna datoteka: V prvi vrstici sta navedeni celi števili n (število stopnic) in p (število pleskarjev), ločeni s presledkom. Sledi še p vrstic, ki opisujejo pleskarje v takem vrstnem redu, v kakršnem hodijo po stopnicah. Pri tem i-ta vrstica vsebuje celi števili k_i (dolžino koraka) in c_i (barvo čevljev), ločeni s presledkom.

Izhodna datoteka: Vanjo izpiši 26 vrstic, po eno za vsako barvo od 1 do 26. Prva vrstica torej opisuje barvo 1, druga barvo 2 itd. Vsaka vrstica naj vsebuje eno samo celo število — število stopnic, ki so zamazane s tisto barvo.

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:

`10 3`
`5 1`
`1 2`
`3 4`
	`0`
	`6`
	`0`
	`4`
	`0`
	⋮ (še 21 ničel)



Stanje stopnišča, ko so se po njem sprehodili vsi pleskarji iz primera.

3. Pogrešane osebe (`osebe.in`, `osebe.out`)

Policija se je odločila za razvoj **programa**, ki bo pregledoval sporočila ter ugotavljal, ali se v njih nahajajo imena pogrešanih oseb.

Program dobi kot vhod seznam imen pogrešanih oseb ter sporočilo. Na izhodu pa mora izpisati, kolikokrat se v tem sporočilu pojavi ime katerekoli od pogrešanih oseb s seznama.

Vhodna datoteka: v prvi vrstici je celo število n ($1 \leq n \leq 10\,000$), ki pomeni število pogrešanih oseb. Sledi n vrstic; vsaka od njih vsebuje ime ene od pogrešanih oseb. Dolžina vsakega od teh imen je najmanj 1 in največ 20 znakov. Sledi še ena vrstica, ki vsebuje prestreženo sporočilo; to je dolgo vsaj 1 in največ 10 000 000 znakov.

Izhodna datoteka: vanjo izpiši eno samo celo število, ki pove skupno število vseh pojavitev vseh imen pogrešanih oseb. Pri tem štejejo le pojavitve, pri katerih se ime pojavi v nespremenjeni obliki (ujemati se mora tudi to, katere črke so male in katere velike — npr. „`Peter`“ in „`pEtEr`“ se ne ujemata) kot podniz v besedilu; ni pa nujno, da se pojavi kot samostojna beseda.

Opomba: pri vseh testnih primerih te naloge bodo kot sporočila uporabljena besedila v naravnem jeziku. Tudi kot imena oseb bodo uporabljene besede (ne sicer nujno imena) v naravnem jeziku. Od desetih testnih primerov bo pri petih sporočilo krajše od 200 000 znakov, pri dveh drugih pa bo sporočilo krajše od 2 000 000 znakov. Tako sporočila kot vzorci bodo sestavljeni le iz znakov z ASCII kodami od 32 do 127.

Primer vhodne datoteke:

```
3
Peter
Peter Novak
Janez
Janeza videli v Petersburgu.  Petra ne.  Peter Novak se je sprehajal po gozdu.
```

Pripadajoča izhodna datoteka:

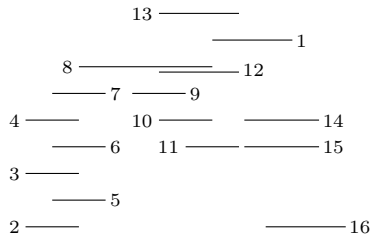
```
4
```

Pravilni odgovor je 4 zato, ker se `Janez` pojavlja enkrat, `Peter` dvakrat (enkrat v „`Petersburgu`“ in enkrat v „`Peter Novak`“) in `Peter Novak` enkrat; skupaj imamo torej $1 + 2 + 1 = 4$ pojavitve.

4. Lemingi (`lemingi.in`, `lemingi.out`)

Ekipa programerjev, ki razvija preprosto različico igre Lemingi, te prosi za pomoč pri izdelavi igralnih zemljevidov. Igra je narejena zelo preprosto. Igralna površina je dvodimenzionalna in sestavljajo jo vodoravne črte, po katerih lahko hodijo lemingi. Črte so različno dolge in nameščene na različnih višinah. Ko se leming znajde na eni od teh črt, lahko začne hoditi v levo ali pa v desno in ko pride do roba, pade navpično navzdol na naslednjo črto. Spet se odloči za eno od smeri in hodi ves čas v tisti smeri, dokler spet ne pade čez rob. Pod vsemi temi črtami ležijo tla, ki se na levo in na desno raztezajo v neskončnost. Na tej poti se lahko leming ubije, če pade s prevelike višine. **Napiši program**, ki prebere opis igralnega zemljevida (torej položaje vseh črt) in ugotovi, katera je najnižja črta, ki jo lahko leming doseže, ne da bi se pri kakšnem padcu ubil.

Opomba: predpostavimo, da so lemingi zelo majhni in lahko hodijo po neki črti, tudi če je tik nad njo še neka druga črta (npr. če so na črti 12 na spodnji sliki, lahko pridejo do njenega levega roba in od tam padejo na črto 9); lahko tudi padejo skozi poljubno ozko špranjo med dvema črtama (npr. če pade leming z desnega roba črte 12, bo mirno poletel skozi špranjo med 11 in 15 ter pristal šele na tleh). Leming ob padcu ne more pristati na robu črte (v tem primeru bi se njegov padec nadaljeval, kot da te črte ni; na primer, če pade z desnega roba črte 12, ne bi pristal na črti 11, pač pa na tleh). Črte se ne dotikajo (nimajo nobene skupne točke).



Vhodna datoteka: V prvi vrstici sta celi števili n in d, ločeni s presledkom. Pri tem je n število vseh črt na igralnem zemljevidu ($1 \leq n \leq 1000$), d pa je višina najvišjega padca, ki ga leming še preživi (torej, če je razlika med višino črte, s katere je padel, in tiste, na kateri je pristal, manjša ali enaka d, bo preživel, drugače pa ne). Sledi n vrstic, od katerih vsaka vsebuje trojico števil, a, b in h, s katero je opisana ena črta (števila so ločena s po enim presledkom). V običajnem koordinatnem sistemu sta a in b leva in desna x-koordinata, h pa je y-koordinata, torej višina črte nad tlemi (za vsako črto velja $a < b$ in $h > 0$). Koordinate so cela števila med vključno 0 in vključno 1 000 000. Vsak teren ima tla, ki v vhodnih podatkih niso posebej podana in si jih lahko mislimo kot $a = -\infty$, $b = +\infty$, $h = 0$. Leming začne svojo pot na črti, ki je v vhodni datoteki navedena na prvem mestu.

Izhodna datoteka: izpiši eno samo celo število, in sicer višino najnižje črte, ki jo lahko leming doseže, ne da bi se na poti do nje ob kakšnem padcu ubil (živ mora

ostati tudi po tistem zadnjem padcu, v katerem je pristal na tej črti). Če lahko živ doseže tudi tla (in ostane živ po tem zadnjem skoku na tla), izpiši 0.

Primer vhodne datoteke:

```
16 30
70 100 80
0 20 10
0 20 30
0 20 50
10 30 20
10 30 40
10 30 60
20 70 70
40 60 60
50 70 50
60 80 40
50 80 69
50 80 90
81 110 50
81 110 40
90 120 10
```

Pripadajoča izhodna datoteka:

```
40
```

To je primer z zgornje slike. S črte 1 lahko lemingi pridejo na 12 in 14; z 12 lahko pridejo na 9, s te na 10 in s te na 11. Najnižja črta, ki jo lahko lemingi dosežejo živi, je torej črta 11, ki je na višini 40. Drugi možni padci (pri katerih pa se leming ubije) so še: s 14 na 16; in z 9, 10, 11, 12 ali 14 na tla.

5. Štoparji (`stoparji01.out`, `stoparji02.out`, ..., `stoparji10.out`)

Z vozilom, ki ima poleg voznika še prostor za k potnikov, se nameravaš odpeljati v oddaljeno mesto in na poti do tja pobirati štoparje. Cesto, po kateri boš potoval, si že izbral, vzdolž nje pa ležijo mesta, kjer stojijo štoparji. Označimo jih s števili od 1 do n, pri čemer je 1 naše začetno mesto (tudi tukaj lahko poberemo štoparje), n pa naš cilj.

Mesta so označena v takem vrstnem redu, v kakšnem si sledijo po poti. Peljali se bomo natanko enkrat od 1 do n, brez vračanja. Za vsakega štoparja vemo, v katerem mestu se nahaja in v katero mesto hoče priti (stvar storitve SMS-Štopar-Info `:->`). V vsakem mestu lahko sami izbiramo, katerega štoparja bomo pobrali (če sploh katerega; lahko pa jih poberemo tudi več), toda če ga že poberemo, ga moramo peljati v mesto, kamor želi priti (ne smemo ga npr. že prej odložiti iz avtomobila). Zanima nas, katere štoparje je treba pobrati, da bo skupno število prepeljanih štoparjev čim večje. Ker je prostih sedežev samo k, lahko naenkrat peljemo le k štoparjev, drugače pa ni posebnih omejitev glede tega, koliko jih lahko skupno prepeljemo med celo vožnjo.

Pri tej nalogi ne boš oddajal izvorne kode programa, pač pa boš dobil deset vhodnih datotek (`stoparji01.in`, ..., `stoparji10.in`), za vsako od njih pa moraš oddati eno izhodno datoteko (s čim boljšo rešitvijo za primer iz tiste vhodne datoteke). Imena izhodnih datotek naj bodo `stoparji01.out`, ..., `stoparji10.out`.

Vhodna datoteka: v prvi vrstici so tri cela števila, najprej n (število mest), nato k (največje število potnikov, ki se lahko peljejo hkrati) in nato še m (število štoparjev); ločena so s po enim presledkom. Veljalo bo: $2 \leq n \leq 100$, $1 \leq k \leq 100$ in $1 \leq m \leq 1000$. Sledi še m vrstic, za vsakega štoparja po ena; v i-ti od teh vrstic (za $i = 1, \ldots, n$) sta dve celi števili s_i in e_i, ločeni s presledkom. Pri tem je s_i številka mesta, v katerem čaka štopar i, e_i pa je številka mesta, v katero se želi štopar i peljati. Za vsak i bo veljalo: $1 \leq s_i < e_i \leq n$. Štoparji so navedeni naraščajoče po številki mesta, kjer čakajo; tisti, ki čakajo v istem mestu, pa so navedeni naraščajoče po številki mesta, do katerega se želijo peljati.

Izhodna datoteka: v prvi vrstici izpiši celo število g, to je skupno število štoparjev, ki bi se jih dalo prepeljati ob upoštevanju vseh omejitev iz besedila naloge. Sledi naj še g vrstic; v vsaki od njih naj bo številka nekega štoparja (celo število od 1 do m) in to tako, da se noben štopar ne pojavlja več kot enkrat in da je mogoče tako opisano množico štoparjev res prepeljati, ne da bi jih bilo v avtomobilu kdaj več kot k hkrati. (Če v nekem mestu nekaj štoparjev izstopi, nekaj pa jih vstopi, lahko predpostaviš, da se izstopi zgodijo pred vstopi.)

Točkovanje: če tvoja izhodna datoteka ne ustreza zgoraj opisanim zahtevam, dobiš pri tistem testnem primeru 0 točk. Drugače pa dobiš od 0 do 10 točk, in sicer tem več, čim več štoparjev si uspel prepeljati. Naj bo g število prepeljanih štoparjev iz tvoje rešitve, $g^\star$ pa število štoparjev iz najboljše možne rešitve za dani testni primer; potem dobi tvoja rešitev pri tem testnem primeru $\lfloor 10g/g^\star \rfloor$ točk.

Primer vhodne datoteke:

```
7 4 6
1 3
1 5
1 6
2 7
4 5
4 5
```

Ena od možnih
pripadajočih izhodnih datotek:

```
5
1
2
3
5
6
```

Tako prepeljemo skupno 5 štoparjev; izkaže se, da je to pri tem testnem primeru tudi najboljša možna rešitev (če smemo voziti največ štiri štoparje hkrati, ne bomo mogli pomagati vsem šestim štoparjem).

NALOGI ZA OGREVANJE

Naslednji nalogi smo tekmovalcem poslali po elektronski pošti nekaj dni pred tekmovanjem, nista pa šteli za del tekmovanja (torej, če kdo reši ti dve nalogi, mu to ne prinaša na tekmovanju nobenih dodatnih točk ali česa podobnega).

1. Nizi

Ministrstvo za dobro voljo je izdalo odlok, da morajo vsi ponudniki spletnih storitev cenzurirati strani z mračno vsebino (da bi ljudi obvarovali pred slabo voljo). Za začetek si želijo blokirati dostop do vseh strani, ki omenjajo vojno (v kateremkoli sklonu in številu: *vojna*, *vojne*, *vojn*, ...).

Napiši funkcijo, ki bo za argument dobila niz in vrnila true, če niz vsebuje samostalnik *vojna* v katerikoli obliki, in false, če besedilo vojne ne omenja.[2]

Primeri:

- `Vojne so v zadnjih letih terjale mnogo zrtev.` — vrne true
- `Vojaski cevlji so ga tiscali.` — vrne false
- `S pravim ovojnim papirjem boste darilu vdahnili duso.` — vrne false

Tvoja funkcija naj bo takšne oblike:

```
function JeCenzuraPotrebna(S: string): boolean;    { v pascalu }
bool JeCenzuraPotrebna(char *S);                   /* v C/C++ */
```

2. Algoritmi

Matematik Koreno Glavi je v Butalah zaslovel, ko je dal natisniti debelo Knjigo modrosti, v kateri je dal natisniti v potu svojega obraza naračunano zaporedje

$$01234567891011121314151617181 9\ldots,$$

ki se je raztezalo čez nekaj sto strani. To število je bilo — tako pravijo — sila pomembno za vse tedanje raziskave.

Nekaj let po tistem pa se je v mesto prikradel Cefizelj, ki je baje imel prerokovalske sposobnosti in je znal za vsako stran in vrstico napovedati, s katero številko se začne in konča. Koreno Glavi je od žalosti pri priči umrl, toda butalskemu policaju še vedno ne da miru, kako je bilo mogoče napovedovati številko na poljubni strani knjige.

(*a*) **Napiši program**, ki bo s standardnega vhoda prebral naravno število n in izpisal številko na n-tem mestu v seznamu.

[2] V praksi se lahko stvari zapletejo: na primer, v stavku „bila sta vojna tovariša“ se samostalnik *vojna* ne pojavlja (pač pa pridevnik *vojen*). Pri tej nalogi seveda ni mišljeno, da naj zna naša rešitev pravilno ugotavljati, ali je neka beseda v stavku res oblika samostalnika *vojna* ali pa oblika neke druge besede (npr. pridevnika *vojen*), ki je slučajno enaka eni od oblik samostalnika *vojna*. Dovolj bo že, če naša funkcija vrne true, čim se v besedilu pojavi neka beseda, ki je enaka kakšni od oblik samostalnika *vojna*.

Primeri (levo je vhodni podatek, desno pa števka, ki jo mora tvoj program izpisati):

$$\begin{array}{rcl} 0 & \to & 0 \\ 8 & \to & 8 \\ 10 & \to & 1 \\ 11 & \to & 0 \\ 21 & \to & 5 \end{array}$$

Pomembno je predvsem, da je program dovolj hiter in ne preveč „požrešen“.

(*b*) Tepanjčani so za tistih časov uporabljali še sedmiški sistem. Če bi imeli tako bistroumne učenjake kot Butalci, bi podobno zaporedje nemara zapisali kot

$$0123456101112131415162021222324252630 31\ldots$$

Ali znaš nalogo rešiti tudi Tepanjčanom?

REŠITVE NALOG ZA PRVO SKUPINO

1. Poraba goriva

Podprogram uporablja dve polji z desetimi elementi, v katerih si zapomni zadnjih deset meritev porabe in poti. Za pravilno delovanje v prvih desetih sekundah je poskrbljeno že s tem, da so ob zagonu v obeh poljih ničle.

```
var Poraba10, Pot10: array [1..10] of real;
    ZadnjaMeritev: integer;

function PovprecnaPoraba(Poraba1sec, Pot1sec: real): real;
var i: integer; Poraba, Pot: real;
begin
   { Z najnovejšo meritvijo prepišemo najstarejša elementa v globalnih tabelah. }
   ZadnjaMeritev := (ZadnjaMeritev mod 10) + 1;
   Poraba10[ZadnjaMeritev] := Poraba1sec;
   Pot10[ZadnjaMeritev] := Pot1sec;
   { Seštejemo dosedanjo porabo in pot. }
   Poraba := Poraba10[1];
   Pot := Pot10[1];
   for i := 2 to 10 do begin
      Poraba := Poraba + Poraba10[i];
      Pot := Pot + Pot10[i];
   end; {for}
   { Osnovna enota za povprečno porabo so litri na 100 kilometrov, zato
     moramo ustrezno prilagoditi izmerjeno pot, ki je podana v metrih. }
   Pot := Pot / 1000 / 100;
   { Če je skupna pot 0, avto stoji in ne moremo vrniti pametne vrednosti,
     zato vrnemo kar 0.}
   if Pot = 0 then PovprecnaPoraba := 0
   else PovprecnaPoraba := Poraba / Pot;
end; {PovprecnaPoraba}
```

Ali, v C-ju:

```
double Poraba[10], Pot[10];
int ZadnjaMeritev;

double PovprecnaPoraba(double Poraba1sec, double Pot1sec)
{
   int i; double SkupajPoraba, SkupajPot;
   /* Zapomnimo si novi meritvi. */
   Poraba[ZadnjaMeritev] = Poraba1sec;
   Pot[ZadnjaMeritev] = Pot1sec;
   ZadnjaMeritev = (ZadnjaMeritev + 1) % 10;
   /* Izračunajmo skupno pot in porabo za zadnjih 10 sekund. */
   for (i = 0, SkupajPoraba = 0, SkupajPot = 0; i < 10; i++)
      SkupajPoraba += Poraba[i], SkupajPot += Pot[i];
   /* Izračunajmo rezultat. */
   SkupajPot /= 100000.0; /* Enota je zdaj 100 km. */
   return SkupajPot == 0.0 ? 0.0 : SkupajPoraba / SkupajPot;
}
```

Še nekaj časa lahko prihranimo, če v globalnih spremenljivkah hranimo tudi skupno porabo in prevoženo razdaljo v zadnjih desetih sekundah. Ti dve vsoti lahko potem vsako sekundo popravimo tako, da odštejemo meritev izpred desetih sekund

in prištejemo najnovejšo meritev. Morebitna slabost te rešitve je, da bi se na dolgi rok utegnile nabirati v teh vsotah kakšne numerične nenatančnosti (zaradi napak pri zaokrožanju).

```
var Poraba, Pot: array [1..10] of real;
    SkupajPoraba, SkupajPot: real;
    ZadnjaMeritev: integer;

function PovprecnaPoraba(Poraba1sec, Pot1sec: real): real;
begin
  ZadnjaMeritev := (ZadnjaMeritev mod 10) + 1;
  { Popravimo vsoto porabe in poti po zadnjih desetih sekundah. }
  SkupajPoraba := SkupajPoraba - Poraba[ZadnjaMeritev] + Poraba1sec;
  SkupajPot := SkupajPot - Pot[ZadnjaMeritev] + Pot1sec;
  { Zapomnimo si novo pot in porabo (z njima prekrijemo deset
    sekund staro meritev). }
  Poraba[ZadnjaMeritev] := Poraba1sec; Pot[ZadnjaMeritev] := Pot1sec;
  { Če je skupna pot 0, avto stoji in ne moremo vrniti
    pametne vrednosti, zato vrnemo kar 0. }
  if SkupajPot <= 1e-6 then PovprecnaPoraba := 0
  { Ker nas zanima poraba na 100 kilometrov, ne na en meter, bomo množili s 100000. }
  else PovprecnaPoraba := SkupajPoraba * 100000.0 / SkupajPot;
end; {PovprecnaPoraba}
```

Ali, v C-ju:

```
double Poraba[10], Pot[10], SkupajPoraba, SkupajPot;
int ZadnjaMeritev;

double PovprecnaPoraba(double Poraba1sec, double Pot1sec)
{
  /* Pozabimo najstarejši meritvi. */
  SkupajPoraba -= Poraba[ZadnjaMeritev];
  SkupajPot -= Pot[ZadnjaMeritev];
  /* Zapomnimo si novi meritvi. */
  SkupajPoraba += Poraba[ZadnjaMeritev] = Poraba1sec;
  SkupajPot += Pot[ZadnjaMeritev] = Pot1sec;
  ZadnjaMeritev = (ZadnjaMeritev + 1) % 10;
  /* Izračunajmo rezultat. Množenje s 100000 je zato,
     ker nas zanima poraba na 100 km, ne na 1 m. */
  return SkupajPot < 1e-6 ? 0.0 : SkupajPoraba * 100000.0 / SkupajPot;
}
```

2. Hanojski stolpi

V spremenljivkah Dovoljeno in Koncno hranimo naše ugotovitve o stanju, ki ga beremo: na začetku sta obe spremenljivki true, čim pa ugotovimo, da stanje ne ustreza kakšnemu od pogojev, postavimo ustrezno spremenljivko na false. Za preverjanje, če je stanje dovoljeno, si je koristno med branjem zaporedja ploščic na palici zapomniti tudi velikost predhodne ploščice na isti palici; v ta namen uporabljamo spremenljivko PrejPlosca. Vprašanje je še, kaj narediti pri prvi (najnižji) ploščici posamezne palice, kjer predhodne ploščice sploh še ni: koristno je, če pred začetkom branja posamezne palice postavimo PrejPalica na neko dovolj veliko vrednost (npr. $n + 1$, kar je večje tudi od največje ploščice), da pri prvi ploščici pogoj „**if** Plosca > PrejPlosca“ gotovo ne bo izpolnjen.

Preverjanja, če je stanje končno, se lahko lotimo na različne načine. Vsekakor je pametno preveriti, če je na tretji palici vseh n ploščic (tega, da sta prvi dve palici prazni, nam v tem primeru ni treba posebej preverjati, saj naloga obljublja, da se bo v vhodnih podatkih vsaka ploščica pojavila natanko enkrat). Poleg tega lahko preverimo, da je npr. i-ta najnižja ploščica na tretji palici velika natanko $n + 1 - i$ — to pomeni, da je najnižja ploščica velika n, druga najnižja je velika $n - 1$ in tako naprej, najvišja ploščica (ki je torej n-ta najnižja na tej palici) pa je velika 1. Tako dela spodnji program; lahko pa bi namesto tega le preverili, če je na tretji palici vseh n ploščic in če je stanje dovoljeno; iz tega dvojega že sledi, da so ploščice na tretji palici tudi v pravilnem vrstnem redu, torej je stanje končno.

```
program HanojskiStolpi;
var n, m, Palica, i, Plosca, PrejPlosca: integer;
    Dovoljeno, Koncno: boolean;
begin
  Dovoljeno := true; Koncno := true;
  ReadLn(n);
  for Palica := 1 to 3 do begin
    Read(m);
    if (Palica = 3) and (m <> n) then Koncno := false;
    PrejPlosca := n + 1;
    for i := 1 to m do begin
      Read(Plosca);
      if Plosca > PrejPlosca then Dovoljeno := false;
      if (Palica = 3) and (Plosca <> n + 1 - i) then Koncno := false;
      PrejPlosca := Plosca;
    end; {for i}
  end; {for Palica}
  Write('Stanje '); if Dovoljeno then Write('je') else Write('ni');
  WriteLn(' dovoljeno.');
  Write('Stanje '); if Koncno then Write('je') else Write('ni');
  WriteLn(' končno.');
end. {HanojskiStolpi}
```

Ali, v C-ju:

```
#include <stdio.h>
#include <stdbool.h>

int main()
{
  int n, m, i, plosca, prejPlosca, palica;
  bool dovoljeno = true, koncno = true;
  scanf("%d", &n); /* Preberimo število ploščic. */
  for (palica = 1; palica <= 3; palica++)
  {
    scanf("%d", &m); /* Preberimo število ploščic na tej palici. */
    /* Stanje je končno le, če so vse ploščice na tretji palici. */
    if (palica == 3 && m != n) koncno = false;
    /* Preverimo, če je vsaka ploščica manjša od tiste pod njo. */
    for (i = 0, prejPlosca = n + 1; i < m; i++) {
      scanf("%d", &plosca);
      if (plosca >= prejPlosca) dovoljeno = false;
      prejPlosca = plosca; }
  }
  /* Če stanje ni dovoljeno, tudi končno ne more biti. */
```

```
  if (! dovoljeno) koncno = false;
  /* Izpišimo rezultat. */
  printf("Stanje %s dovoljeno in %s končno.\n",
    dovoljeno ? "je" : "ni", koncno ? "je" : "ni");
  return 0;
}
```

3. Ceste

V zanki pregledujmo vse večje številke tablic (od 1 naprej; trenutno številko hranimo v spremenljivki km) in pri vsaki številki tablice preglejmo vse ceste, da vidimo, na koliko cestah bomo potrebovali tablico s to številko (namreč na tistih, pri katerih je dolžina večja od številke na tablici). Za ta pregled cest uporabimo notranjo zanko (**for** StCeste v spodnjem programu); v spremenljivki StTablic štejmo, koliko tablic s to številko potrebujemo. Če ugotovimo, da te tablice sploh ne potrebujemo (StTablic je 0), pomeni, da smo že dosegli dolžino najdaljše ceste in tudi tablic z večjimi številkami ne bomo potrebovali; takrat se lahko naš program konča.

```
program Tablice;
var km, StTablic, StCeste: integer;
begin
  km := 0;    { Številke tablic bomo pregledovali naraščajoče od 1 naprej. }
  repeat
    StTablic := 0; km := km + 1;
    for StCeste := 1 to StCest do              { Preglejmo vse ceste. }
      if DolzinaCeste(StCeste) > km then       { Ali na tej cesti potrebujemo to tablico? }
        StTablic := StTablic + 1;              { Da, potrebujemo jo — povečajmo števec. }
    if StTablic > 0 then    { Izpišimo, koliko primerkov te tablice potrebujemo. }
      WriteLn('Potrebujemo ', StTablic, ' tablic z oznako ', km, ' km.');
  until StTablic = 0;         { Očitno smo prišli do konca vseh cest, tudi najdaljše. }
end. {Tablice}
```

Pa še v C-ju:

```
#include <stdio.h>
extern int StCest();
extern int DolzinaCeste(int i);
int main()
{
  int km, StTablic, StCeste;
  for (km = 1; ; km++) /* Glejmo tablice z vse večjimi številkami. */
  {
    /* Koliko cest potrebuje tablico s to stacionažo? */
    for (StCeste = 1, StTablic = 0; StCeste <= StCest(); StCeste++)
      if (km < DolzinaCeste(StCeste)) StTablic++;
    /* Če takih tablic ne potrebujemo, tudi tistih z večjimi
       številkami ne bomo potrebovali in lahko končamo. */
    if (StTablic == 0) break;
    else printf("Potrebujemo %d tablic z oznako %d km.\n", StTablic, km);
  }
  return 0;
}
```

4. Zlogovna pisava

Vrivanje dodatnih črk je potrebno v naslednjih primerih: (1) če sta dve zaporedni črki samoglasnika, je treba vmes vriniti nek soglasnik; (2) če sta dve zaporedni črki soglasnika, je treba vmes vriniti nek samoglasnik; (3) če je prva črka samoglasnik, je treba pred njo vriniti nek soglasnik; (4) če je zadnja črka soglasnik, je treba za njo vriniti nek samoglasnik.

Primere tipa (1) in (2) lahko odkrijemo tako, da se z zanko zapeljemo po nizu in za dva zaporedna znaka preverjamo, če sta samoglasnika ali ne. Če ta test pri obeh vrne enak rezultat, pomeni, da sta oba samoglasnika ali pa oba samoglasnika in bo treba mednju vriniti novo črko, sicer pa ne. Primera (3) in (4) lahko preverimo posebej.

```
function KolikoCrk(s: string): integer;
const Samoglasniki = ['a', 'e', 'i', 'o', 'u'];
var i, L, n: integer;
begin
  L := Length(s); n := 0;
  for i := 1 to L − 1 do
    if (s[i] in Samoglasniki) = (s[i + 1] in Samoglasniki) then n := n + 1;  { pogoja 1 in 2 }
  if L > 0 then begin
    if s[1] in Samoglasniki then n := n + 1;          { pogoj 3 }
    if not (s[L] in Samoglasniki) then n := n + 1;    { pogoj 4 }
  end; {if}
  KolikoCrk := n;
end; {KolikoCrk}
```

Še lažje je, če za stražarja postavimo en samoglasnik na začetek niza in en soglasnik na konec niza. Potem postane pogoj (3) le še en primer pogoja (1), pogoj (4) pa le še en primer pogoja (2) in ju ni treba preverjati posebej:

```
function KolikoCrk(s: string): integer;
const Samoglasniki = ['a', 'e', 'i', 'o', 'u'];
var i, n: integer;
begin
  n := 0; s := 'a' + s + 'b';
  for i := 1 to Length(s) − 1 do
    if (s[i] in Samoglasniki) = (s[i + 1] in Samoglasniki) then n := n + 1;
  KolikoCrk := n;
end; {KolikoCrk}
```

Preverjanje, ali je neka črka samoglasnik, lahko izvedemo na različne načine. Zgornja podprograma imata samoglasnike navedene kar v konstanti Samoglasniki, ki je množica (**set of** char), tako da lahko pripadnost preverjamo z operatorjem **in**. Še ena možnost je seveda ta, da znak, ki nas zanima, kar po vrsti primerjamo z vsemi petimi samoglasniki:

```
  if (c = 'a') or (c = 'e') or (c = 'i') or (c = 'o') or (c = 'u') then ...
```

Ker pa potrebujemo takšno preverjanje na več koncih, je bolje, če ga zapakiramo v samostojen podprogram:

```
function JeSamoglasnik(c: char): boolean;
begin
  JeSamoglasnik := (c = 'a') or (c = 'e') or (c = 'i') or (c = 'o') or (c = 'u');
end; {JeSamoglasnik}
```

Zapišimo našo rešitev še v C-ju:

```
bool JeSamoglasnik(char c) {
  return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u'; }
int KolikoCrk(const char *s)
{
  /* Pretvarjali se bomo, da je pred prvo črko že nek samoglasnik.
     To bo poskrbelo za vrivanje soglasnika, če je prva črka samoglasnik. */
  bool prejSam, sam = true;
  int n = 0; /* Števec vrinjenih črk. */
  while (*s)
  {
    prejSam = sam;
    sam = JeSamoglasnik(*s++);
    /* Kjer sta dva soglasnika skupaj ali dva samoglasnika skupaj, */
    if (sam == prejSam) n++; /* bo treba vmes vriniti še eno črko. */
  }
  if (! sam) n++; /* Zadnja črka je soglasnik, vriniti bo treba še eno črko. */
  return n;
}
```

Mimogrede, pisava, ki smo jo uporabili za primere v besedilu naloge, je linearna B, ki so jo uporabljali v Grčiji v 2. tisočletju pr. n. š. Ta se sicer z našo nalogo ne ujema popolnoma, saj ni imela le znakov za pare „soglasnik + samoglasnik“, pač pa tudi za posamezne samoglasnike (ne pa tudi za posamezne soglasnike). Zato jim pri pisanju ni bilo treba vrivati v besede namišljenih soglasnikov, pač pa le samoglasnike. Še ena razlika v primerjavi z našo nalogo je, da težav pri pisanju niso reševali le z vrivanjem dodatnih črk, pač pa včasih tudi z brisanjem (če se je beseda končala na soglasnik). Napis na začetku naloge je poskus, da bi v tej pisavi zapisali besedi „zlogovna pisava“ (kot *zo-lo-go-u-na pi-sa-u̯a*).

5. Enačbe

Vhodni niz pregledujmo črko za črko. Pri vsaki črki preverimo, če je številka, enačaj ali znak plus; če ni nič od tega trojega, vemo, da je niz neveljaven. Če vidimo enačaj, si to zapomnimo v neki globalni spremenljivki (v spodnjem podprogramu je to Enacaj); če kasneje vidimo še enega, vemo, da je niz neveljaven. Ostale zahteve iz besedila naloge še najlaže upoštevamo tako, da gledamo po dva zaporedna znaka vhodnega niza. Če je trenutni znak enačaj ali plus, mora biti prejšnji znak številka, ne pa enačaj ali plus. Tudi prvi znak niza mora biti številka, ne pa enačaj ali plus; to lahko preverjamo z istim pogojem kot za kasnejše znake, če se pretvarjamo, da je pred prvim znakom v nizu še en znak plus (zato spodnji podprogram inicializira spremenljivko c, ki bo drugače hranila prejšnji znak, na `'+'`). Tudi zadnji znak niza mora biti številka; to lahko preverimo posebej.

```
function JeVeljavna(S: string): boolean;
var i: integer; Enacaj: boolean; c: char;
begin
  JeVeljavna := false;
  Enacaj := false; c := '+';
  for i := 1 to Length(S) do begin
    if S[i] = '=' then begin
```

```
        if Enacaj then exit { več kot en enačaj }
        else if c = '+' then exit { enačaj na začetku ali za plusom }
        else Enacaj := true;
      end else if S[i] = '+' then begin
        if (c = '+') or (c = '=') then exit; { plus mora biti za številko }
      end else if (S[i] < '0') or (S[i] > '9') then
        exit; { znak, ki ni niti številka niti plus niti enačaj }
      c := S[i];
    end; {for i}
    if not Enacaj then exit; { S mora vsebovati enačaj }
    if (c < '0') or (c > '9') then exit; { zadnji znak mora biti številka }
    JeVeljavna := true;
end; {JeVeljavna}
```

Lahko bi naredili tudi več prehodov po vhodnem nizu in vsakič preverjali po enega od pogojev iz naloge. Spodnja rešitev na primer najprej preveri, da niz ne vsebuje neveljavnih znakov; nato prešteje enačaje; nato preveri, če sta prvi in zadnji znak številki; in končno preveri še, da se nikjer ne pojavljata skupaj plus in enačaj ali dva plusa ali dva enačaja skupaj.

```
function JeVeljavna2(S: string): boolean;
var i, n, e: integer;
begin
    JeVeljavna2 := false; n := Length(S);
    { Preverimo, če so vsi znaki številke, plusi in enačaji. }
    for i := 1 to n do if not (S[i] in ['+', '=', '0'..'9']) then exit;
    { Preverimo, če je enačaj natanko eden. }
    e := 0; for i := 1 to n do if S[i] = '=' then e := e + 1;
    if e <> 1 then exit;
    { Preverimo, da prvi in zadnji znak nista plusa ali enačaja. }
    if (S[1] in ['+', '=']) or (S[1] in ['+', '=']) then exit;
    { Preverimo, da nista nikjer dva plusa, dva enačaja ali plus in enačaj skupaj. }
    for i := 2 to n do if (S[i - 1] in ['+', '=']) and (S[i] in ['+', '=']) then exit;
    JeVeljavna2 := true;
end; {JeVeljavna2}
```

Oglejmo si še rešitev v C-ju:

```
bool JeVeljavna(const char *s)
{
    /* Pretvarjajmo se, da je pred prvim znakom še en plus;
       tako bomo, če prvi znak ni številka, prepoznali niz kot neveljaven. */
    char c = '+'; bool enacaj = false;

    while (*s)
    {
        if (*s == '=') { /* Enačaj naj bo en sam, pred njim naj bo številka. */
            if (c == '+' || enacaj) return false;
            else enacaj = true; }
        else if (*s == '+') { /* Pred plusom mora biti številka. */
            if (c == '+' || c == '=') return false; }
        else if (*s < '0' || *s > '9')
            return false; /* Vsak znak mora biti plus, enačaj ali številka. */
        c = *s++;
    }
    /* Enačaj mora biti prisoten, zadnji znak mora biti številka. */
    return enacaj && ! (c == '+' || c == '=');
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. DKIM

Vhodno datoteko lahko beremo znak po znak; pri tem si v neki spremenljivki zapomnimo, ali smo v trenutni vrstici doslej videli same presledke ali že tudi kakšen drug znak (v spodnjem programu je to ZacVrstice), v neki drugi spremenljivki pa, ali je bil zadnji prebrani znak presledek (v spodnjem programu je to Presledki). Poleg tega tudi štejmo, koliko praznih vrstic smo prebrali pred trenutno vrstico (StPraznih v spodnjem programu).

Ko preberemo presledek, ne izpisujemo ničesar, le spremenljivko Presledki postavimo na true. Podobno je, če preberemo prazno vrstico (ali pa tako, v kateri so bili le presledki): tudi takrat ne izpišemo znaka za konec vrstice, le StPraznih povečamo za 1. Ko pa preberemo nek znak, ki ni presledek, izpišemo prebrani znak in pred tem še vse tisto, kar smo dolžni za nazaj: morebitne prazne vrstice (to nam pove StPraznih, ki ga nato postavimo na 0) in presledke (če je Presledki = true). To nam zagotavlja, da presledkov na koncu vrstice in praznih vrstic na koncu datoteke ne bomo izpisali.

```
program DKIM;
var StPraznih: integer;
    Presledki, ZacVrstice: boolean;
    c: char;
begin
  StPraznih := 0;
  while not Eof do begin
    ZacVrstice := true; Presledki := false;
    while not (Eoln or Eof) do begin
      Read(c);
      if c = ' ' then Presledki := true
      else begin
        while StPraznih > 0 do begin WriteLn; StPraznih := StPraznih - 1 end;
        if Presledki then begin Write(' '); Presledki := false end;
        Write(c); ZacVrstice := false;
      end; {if}
    end; {while}
    if Eoln then ReadLn;
    if ZacVrstice then StPraznih := StPraznih + 1
    else WriteLn;
  end; {while}
end. {DKIM}
```

Še hitreje gre, če beremo besedilo po vrsticah. Trenutno vrstico preberimo v nek niz S in porežimo presledke s konca (če jih je kaj). Če po tem dobimo prazen niz, povečamo števec StPraznih, sicer pa vrstico izpišemo (in pred njo še morebitne prazne vrstice od prej), le da moramo pred tem še zamenjati več zaporednih presledkov z enim samim.

```
program PoVrsticah;
var S: string; StPraznih, i, j, n: integer;
begin
  StPraznih := 0;
  while not Eof do begin
```

```
      ReadLn(S); n := Length(S);
      while n > 0 do
        if S[n] <> ' ' then break else n := n − 1;
      if n = 0 then StPraznih := StPraznih + 1
      else begin
        for i := 1 to StPraznih do WriteLn;
        StPraznih := 0;
        j := 1;
        for i := 2 to n do
          if (S[i] <> ' ') or (S[i − 1] <> ' ') then
            begin j := j + 1; S[j] := S[i] end;
        WriteLn(Copy(S, 1, j));
      end; {while}
  end; {if}
end. {PoVrsticah}
```

Še rešitev v C-ju:

```
#include <stdio.h>

int main()
{
  char s[202], *p, *r; int stPraznih = 0;
  while (fgets(s, sizeof(s), stdin))
  {
    p = r = s; while (*r) r++;
    /* Porežimo presledke in znak za konec vrstice s konca niza. */
    while (p < r && isspace(r[−1])) r−−;
    /* Ali je ostala prazna vrstica? */
    if (p == r) { stPraznih++; continue; }
    /* Izpišimo prazne vrstice od prej. */
    while (stPraznih > 0) { printf("\n"); stPraznih−−; }
    /* Več zaporednih presledkov spremenimo v enega. */
    for (*r = 0, r = ++p; *r; r++)
      if (! (*r == ' ' && r[−1] == ' ')) *p++ = *r;
    *p = 0; printf("%s\n", s); /* Izpišimo trenutno vrstico. */
  }
  return 0;
}
```

Enovrstičnica v perlu:

```
perl −ne 's/ +/ /g;  s/ $//;  /^$/ ? $z++ : (print("\n" x $z, $_), $z=0)'
```

Pri tem stikalo `−ne` pomeni, da je program podan kot naslednji parameter na ukazni vrstici (`−e`) in da naj ga interpreter izvede po enkrat za vsako vrstico vhodne datoteke (`−n`). Stavek `s/ +/ /g` zamenja vse pojavitve enega ali več zaporednih presledkov (poiščemo jih z regularnim izrazom „ +“) z enim samim presledkom. Stavek `s/ $//` poskrbi za brisanje presledkov s konca vrstice (znak `$` v regularnem izrazu zahteva ujemanje le na koncu vrstice). Izraz `/^$/` preveri, če je vrstica potem prazna (`^` se namreč ujame le z začetkom vrstice, `$` pa, kot smo rekli že zgoraj, le s koncem vrstice); če je, povečamo števec praznih vrstic (spremenljivka `$z`); sicer pa izpišemo najprej `$z` praznih vrstic in nato še trenutno vrstico (`$_`), števec `$z` pa postavimo na 0. Tako bomo poskrbeli za brisanje praznih vrstic na koncu datoteke.

2. Vsote

Recimo, da je na prvem seznamu m klubov, na drugem pa k klubov. Naj bo a_i velikost (število članov) i-tega kluba s prvega seznama, b_i pa velikost i-tega kluba z drugega seznama.

Verjetno najpreprostejša, vendar tudi najbolj neučinkovita, je rešitev z dvema gnezdenima zankama. Zunanja zanka se sprehaja po klubih z enega seznama, pri vsakem od njih pa potem notranja zanka pregleda vse klube z drugega seznama in preveri, če imata kakšna dva kluba skupaj točno n članov.

```
for i := 1 to m do
  for j := 1 to k do
    if a_i + b_j = n then
      izpiši rešitev: i-ti klub s prvega in j-ti klub z drugega seznama;
```

Slabo pri tej rešitvi je, da mora v najslabšem primeru pregledati vse pare klubov: časovna zahtevnost je torej $O(mk)$.

Nalogo, ki jo v tem postopku opravlja notranja zanka, lahko pravzaprav na kratko povzamemo takole: v drugem seznamu poskuša poiskati kakšen klub z $n - a_i$ člani. Ta bi dal potem skupaj z i-tim klubom prvega seznama ravno n ljudi, kolikor jih potrebujemo. Zdaj je torej vprašanje, ali lahko tak klub poiščemo še kako hitreje kot s pregledovanjem celega drugega seznama.

Ena možnost je na primer bisekcija. Drugi seznam za začetek uredimo, tako da bo veljalo $b_1 \le b_2 \le \ldots \le b_{k-1} \le b_k$. Zdaj se po njem premikajmo z dvema indeksoma, ki nam označujeta območje, na katerem bi utegnila ležati iskana vrednost; na začetku tadva indeksa pokrivata cel drugi seznam. V vsakem koraku poglejmo element na sredini našega območja; če je večji od iskane vrednosti, se lahko v bodoče omejimo na levo polovico našega območja, če je manjši, pa na desno polovico. Tako se opazovano območje na vsakem koraku razpolovi in po največ $O(\log k)$ korakih ostane v njem en sam element: če je iskana vrednost tam, smo jo našli, če pa je ni tam, potem vemo, da je sploh ni v drugem seznamu.

```
uredi drugi seznam;
for i := 1 to m do begin
  q := n - a_i;    (* vrednost, ki jo iščemo v drugem seznamu *)
  if q < b_1 then continue; (* tako majhne vrednosti ni *)
  l := 1; r := m + 1;
  while r - l > 1 do begin
    (* Zdaj velja: b_l ≤ q < b_r (pri r = m + 1 si mislimo b_r = ∞).
       Torej, če je iskana vrednost q sploh kje v drugem seznamu, je na enem
       od indeksov l, l + 1, ..., r - 1. *)
    c := (l + r) div 2;    (* Srednji element opazovanega območja. *)
    if q < b_c then r := c else l := c;
  end; (* while *)
  if b_l = q then
    izpiši rešitev: i-ti klub s prvega in l-ti klub z drugega seznama;
end; (* for *)
```

Za urejanje drugega seznama porabimo $O(k \log k)$ časa, če uporabimo katerega od učinkovitih postopkov za urejanje, npr. quicksort. Ostaneta nam še obe gnezdeni

zanki, pri čemer zdaj zunanja še vedno naredi m iteracij, tako kot prej, notranja pa jih naredi le $O(\log k)$. Časovna zahtevnost postopka je torej $O((m+k)\log k)$. Če za urejanje drugega seznama uporabimo kakšnega od manj učinkovitih postopkov, nam bo urejanje vzelo $O(k^2)$ časa in vse skupaj zato $O(k^2 + m\log k)$ časa. V vsakem primeru je koristno, če je drugi seznam čim krajši — v ta namen lahko, če je treba, pred začetkom postopka prvi in drugi seznam tudi zamenjamo, saj je z vidika naloge vseeno, kateri je kateri.

Naloga pravi, da je klubov veliko, ne pove pa, koliko članov ima posamezen klub. Recimo, da število članov posameznega kluba ni večje od s in da število s ni veliko v primerjavi s številom klubov (torej $m+k$). Potem je še najpreprosteje, če si pripravimo kar tabelo s elementov, kjer bo za vsako možno velikost kluba pisalo, kateri klub iz drugega seznama (če sploh kateri) ima točno toliko članov. Če je takih klubov več, je načeloma vseeno, katerega si tam zapomnimo (oz. je to odvisno od tega, kateri klub je politikom bolj simpatičen `:)`).

naj bo t neka tabela s s elementi;
for $r := 1$ **to** s **do** $t[r] := -1$;
for $j := 1$ **to** k **do** $t[b_j] := j$;
for $i := 1$ **to** m **do**
  if $1 \le n - a_i \le s$ **then if** $t[n-a_i] > 0$ **then**
    izpiši rešitev: i-ti klub s prvega in $t[n-a_i]$-ti klub z drugega seznama;

Zdaj nimamo več dveh gnezdenih zank — preverjanje, ali je na drugem seznamu kakšen klub z $n - a_i$ člani, je zelo poceni, le v ustrezno celico tabele t moramo pogledati. Časovna zahtevnost postopka je le še $O(m+k+s)$; porabimo pa tudi $O(s)$ dodatnega pomnilnika (za tabelo t). To je zelo učinkovita rešitev, če le vrednost s (= število članov v največjem klubu) ni prevelika.

Če imajo klubi vendarle preveč članov, lahko t iz navadne tabele predelamo v razpršeno tabelo. To pomeni, da zdaj vsaki velikosti kluba sicer pripada neka celica tabele t, vendar indeks te celice ni kar velikost kluba sama, pač pa bomo indeks izračunali iz velikosti kluba z neko funkcijo (ki ji pravimo razprševalna funkcija). Primerna funkcija je na primer ostanek po deljenju: klub z r elementi naj pripada celici $r \bmod p$, če je p velikost tabele t. Kot vidimo, se zdaj lahko zgodi, da posamezna celica tabele t predstavlja več velikosti klubov (klubi z $r, r+p, r+2p, r+3p, \ldots$ elementi bi vsi prišli v isto celico); namesto indeksa kluba bo zato v tej celici kazalec na seznam vseh klubov, katerih velikost pripada tej celici. Pri tem se bomo zanašali na dejstvo, da bodo ti seznami v praksi kratki: imamo k klubov in če se njihove velikosti kolikor toliko enakomerno razpršijo med vseh p celic tabele, imajo naši seznami povprečno po k/p elementov. Za p je torej pametno vzeti neko dovolj veliko število — recimo vsaj tolikšno kot k.

naj bo t neka tabela s p elementi;
for $r := 1$ **to** p **do** $t[r] :=$ prazen seznam;
for $j := 1$ **to** k **do**
  dodaj število j v seznam $t[b_j \bmod p]$;
for $i := 1$ **to** m **do begin**
  $q := n - a_i$;
  za vsako število j v seznamu $t[q \bmod p]$:

if $b_j = q$ **then**
izpiši rešitev: i-ti klub s prvega in j-ti klub z drugega seznama;
end; (* *for* *)

Če vzamemo p približno tako velik kot k in se klubi dobro razpršijo po tabeli, tako da imajo seznami povprečno le po en element, nam notranja zanka („za vsako število j v seznamu $t[q \bmod p]$“) vzame vsakič le $O(1)$ časa. Časovna zahtevnost celotnega postopka je tako le $O(m + k + p) = O(m + k)$.

Elegantna rešitev pa je tudi tale: oba seznama uredimo in se nato hkrati premikamo po prvem od začetka proti koncu in po drugem od konca proti začetku. Če vidimo, da imata trenutno opazovana kluba skupaj več kot n članov, se premaknemo po drugem seznamu proti začetku (da se bo število članov zmanjšalo); če jih imata premalo, pa se premaknemo po prvem seznamu proti koncu (da se bo število članov povečalo).[3]

uredimo oba seznama;
$i := 1$; $j := k$;
while $i \leq m$ **and** $j \geq 1$ **do**
(* *Na tem mestu velja: vsote n se ne da dobiti z nobenim parom $a_l + b_r$ za $l < i$ ali $r > j$.* *)
if $a_i + b_j > n$ **then** $j := j - 1$
else if $a_i + b_j < n$ **then** $i := i + 1$
else izpiši rešitev: i-ti klub s prvega in j-ti klub z drugega seznama;

Večino časa pri tej rešitvi porabimo za urejanje: $O(m \log m)$ za prvi seznam in $O(k \log k)$ za drugega. Ko je to opravljeno, naredi zanka **while** le $O(m + k)$ iteracij, saj na vsakem koraku premakne enega od števcev; i se lahko poveča največ m-krat, j pa se lahko zmanjša največ k-krat. Časovna zahtevnost celotnega postopka je torej $O(m \log m + k \log k)$.

3. Komparatorji

(a) Naloga pravi, da imajo znaki številske kode od 1 do 255. Ključ naj bo enako dolg kot vhodna vrstica (niz Z); kjer je v nizu Z znak s kodo n, naj ima ključ znak s kodo $256 - n$. Tako bo zagotovo veljalo, da je nek ključ leksikografsko manjši od drugega natanko v primeru, če je bila prva vhodna vrstica leksikografsko večja od druge. Prvo neujemanje v istoležnih znakih dveh ključev nastopi na istem mestu kot pri njunih pripadajočih vrsticah, le da bo, če je imela ena vrstica tam manjši znak kot druga, imel njen ključ zdaj tam večji znak kot ključ druge vrstice.

```
function KljucPadajoce(Z: string): string;
var i: integer; K: string;
begin
  K := '';
  for i := 1 to Length(Z) do
  K := K + Chr(256 - Ord(Z[i]));
  KljucPadajoce := K;
end; {KljucPadajoce}
```

[3] Za še en primer uporabe tega prijema glej rešitev naloge 2001.1.4, str. 447 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004*.

In v C-ju:

```
char *KljucPadajoce(const char *Z)
{
  int d = strlen(Z), i;
  char *K = (char *) malloc(d + 1);
  for (i = 0; i < d; i++) K[i] = 256 - Z[i];
  K[d] = 0; return K;
}
```

(b) Ključ sestavimo tako, da v vhodni vrstici spremenimo male črke v velike, nečrkovne znake (besedilo podproblema (b) pravi, da so nečrkovni znaki v naših vrsticah v bistvu le presledki) pa pri sestavljanju ključa preskočimo.

```
function KljucSlovarsko(Z: string): string;
var i: integer; K: string;
begin
  K := '';
  for i := 1 to Length(Z) do
    if ('A' <= Z[i]) and (Z[i] <= 'Z') then
      K := K + Z[i]
    else if ('a' <= Z[i]) and (Z[i] <= 'z') then
      K := K + Chr(Ord(Z[i]) - Ord('a') + Ord('A'));
  KljucSlovarsko := K;
end; {KljucSlovarsko}
```

In v C-ju:

```
char *KljucSlovarsko(const char *Z)
{
  int d = strlen(Z), i, j;
  char *K = (char *) malloc(d + 1);
  for (i = 0, j = 0; i < d; i++)
    if ('A' <= Z[i] && Z[i] <= 'Z') K[j++] = Z[i];
    else if ('a' <= Z[i] && Z[i] <= 'z') K[j++] = Z[i] - 'a' + 'A';
  K[j] = 0; return K;
}
```

(c) Ključ definirajmo takole: če je vhodna vrstica dolga n znakov, naj se začne ključ na n primerkov črke `b`, za njimi pride črka `a`, nato pa še kopija vhodne vrstice. Prepričajmo se, da bodo tako sestavljeni ključi res poskrbeli za urejanje vrstic po dolžini. Naj bo s neka vrstica, dolga n znakov, in naj bo t neka vrstica, dolga m znakov. Naj bo k_s ključ, ki ga dobimo iz vrstice s, in naj bo t_s ključ, ki ga dobimo iz vrstice t.

Recimo, da je $n > m$; potem se bosta niza k_s in k_t ujemala v prvih m znakih (pri obeh bodo to `bji`), na $(m+1)$-vem mestu pa bo imel k_s znak `b`, niz k_t pa bo imel tam znak `a`. Torej bo k_t leksikografsko manjši od k_s in vrstica t bo pristala v urejeni datoteki pred vrstico s — kot je tudi prav, saj je t krajša od s.

Če je $n < m$, je razmislek podoben, le s in t zamenjata vlogi.

Če pa je $n = m$, se niza k_s in k_t ujemata v prvih $n + 1$ znakih (n `a`jev in en `b`), zato lahko pri primerjanju teh dveh ključev nastopijo razlike šele v nadaljevanju; od $(n+2)$-tega znaka naprej pa je leksikografsko primerjanje nizov k_s in k_t enako, kot če bi primerjali niza s in t (od prvega znaka naprej). Zato bosta vrstici s in t v izhodni datoteki urejeni leksikografsko, kar je tudi prav, saj sta enako dolgi.

```
function KljucPoDolzini(Z: string): string;
var i: integer; K: string;
begin
  K := '';
  for i := 1 to Length(Z) do K := K + 'b';
  KljucPoDolzini := K + 'a' + Z;
end; {KljucPoDolzini}
```

In v C-ju:

```
char *KljucPoDolzini(const char *Z)
{
  int d = strlen(Z), i;
  char *K = (char *) malloc(2 * d + 2);
  for (i = 0; i < d; i++) K[i] = 'b';
  K[d] = 'a';
  for (i = 0; i < d; i++) K[d + 1 + i] = Z[i];
  K[2 * d + 1] = 0; return K;
}
```

(d) Pri primerjanju nizov, ki predstavljajo števila, je nerodna stvar ta, da je npr. niz `23` leksikografsko večji od niza `145`, čeprav je število 23 manjše od 145. Leksikografsko primerjanje nizov `23` in `145` bi začelo pri števkah `2` in `1`, čeprav ima prva vrednost 20, druga pa kar 100. Poskrbeti moramo torej za take ključe, da bomo pri leksikografskem primerjanju števil vedno primerjali dve enakovredni števki: stotice s stoticami, desetice z deseticami in tako naprej. To lahko dosežemo z vrivanjem ničel na začetek niza. Naloga pravi, da imamo opravka z največ osemmestnimi števili, tako da je treba vriniti le toliko ničel, da dobimo osem znakov dolg ključ.

```
function KljucStevilo(Z: string): string;
var i: integer; K: string;
begin
  K := '';
  for i := 1 to 8 - Length(Z) do K := K + '0';
  KljucStevilo := K + Z;
end; {KljucStevilo}
```

In v C-ju:

```
char *KljucStevilo(const char *Z)
{
  int d = strlen(Z), i, MaxD = 8;
  char *K = (char *) malloc(MaxD + 1);
  for (i = 0; i < MaxD - d; i++) K[i] = '0';
  for (i = 0; i < d; i++) K[MaxD - d + i] = Z[i];
  K[MaxD] = 0; return K;
}
```

Bolj za šalo kot zares pa se lahko domislimo tudi naslednje rešitve: poglejmo, katero število predstavlja naša vhodna vrstica; recimo, da je to neko število n; za ključ zdaj uporabimo niz n `a`jev. Vsi ključi so torej sestavljeni iz samih `a`jev, pravila leksikografskega urejanja pa nam zagotavljajo, da bodo zato urejeni po dolžini, pripadajoče vrstice pa bodo s tem urejene po vrednosti števila, ki ga vsebujejo. Ta rešitev je seveda zelo nepraktična, saj so lahko ključi dolgi po več milijonov znakov.

```
function KljucStevilo2(Z: string): string;
var i, n: integer; K: string;
begin
  n := 0; for i := 1 to Length(Z) do n := 10 * n + Ord(Z[i]) - Ord('0');
  K := ''; for i := 1 to n do K := K + 'a';
  KljucStevilo2 := K;
end; {KljucStevilo2}
```

In v C-ju:

```
char *KljucStevilo2(const char *Z)
{
  int d = 0, i; char *K;
  while (*Z) d = d * 10 + *Z++ - '0';
  K = (char *) malloc(d + 2);
  for (i = 0; i < d; i++) K[i] = 'b';
  K[d] = 'a'; K[d + 1] = 0; return K;
}
```

4. Bančni računi

Scenarij: stanje na računu je 100 EUR. Na prvem strežniku se kliče podprogram za dvig 60 EUR. Pogoj v stavku **if** dopusti izvajanje dviga. Ker noben drug strežnik še ni zaklenil računa, se izvajanje nadaljuje: TrenutnoStanje dobi vrednost 100, nato novoStanje postane $100 - 60 = 40$.

Zdaj pa drugi strežnik prične z izvajanjem. V trenutku izvajanja stavka **if** stavka je vrednost na računu še vedno 100, zato stavek **if** nadaljuje z izvajanjem. Vendar pa ZakleniRacun zaustavi izvajanje za toliko časa, dokler prvi strežnik ne konča.

Prvi strežnik medtem zapiše novo vrednost računa (40 EUR). Takoj zatem odklene račun, kar drugemu strežniku omogoči nadaljevanje izvajanja.

Drugi strežnik prebere vrednost računa (40 EUR) v spremenljivko TrenutnoStanje. NovoStanje tako postane -20 EUR, ki se tudi nastavi.

Dopolnitev: klic podprograma ZakleniRacun je potrebno prestaviti pred stavek **if**, klic podprograma OdkleniRacun pa za ta stavek.

5. Šahovnica

Recimo, da smo si že izbrali nek položaj delilnih črt; šahovnica nam s tem razpade na štiri dele, nas pa bo za vsakega od teh štirih delov zanimalo, koliko figur je na njem. To lahko ugotovimo tako, da z dvema gnezdenima zankama (po vrsticah in po stolpcih) pregledamo vsa polja šahovnice; pri vsakem polju, ki vsebuje kakšno figuro, poglejmo, na katerem od naših štirih delov leži, in povečajmo spremenljivko, ki šteje figure v tistem delu šahovnice. Spodnja rešitev hrani te števce figur v tabeli Q. Na koncu poiščemo del z največjim in tistega z najmanjšim številom figur; spomnimo se, da naloga zahteva delitev, pri kateri bo razlika med tema številoma najmanjša. V nekih globalnih spremenljivkah si zapomnimo najboljšo doslej znano delitev in če jo trenutna delitev prekaša, jo zapišimo vanje (v spodnjem programu so to dNaj, xNaj in yNaj). Tako vidimo, da imamo z vsako delitvijo $O(n^2)$ dela, ker moramo pregledati celo šahovnico; in ker je možnih delitev tudi $O(n^2)$ (natančneje $(n-1)^2$ — na $n-1$ načinov si lahko izberemo položaj vodoravne delilne črte, na $n-1$ načinov pa položaj navpične delilne črte), je časovna zahtevnost celotnega postopka kar $O(n^4)$. To je

že zelo počasi; na dandanašnjih računalnikih bi šahovnico 100×100 obdelali v slabi sekundi, za šahovnico 1000×1000 pa bi lahko porabili kar kakšno uro ali še več.

Ker naloga zahteva le opis postopka, ni posebej navedeno, kako naj naš postopek pride do vhodnih podatkov (velikosti in stanja šahovnice). Spodnji program predpostavlja, da lahko prebere opis šahovnice s standardnega vhoda, pri čemer zvezdice pomenijo polja s figurami, drugi znaki pa prazna polja.

```
program Sahovnica1;
const MaxN = 100;
var N, x, y, i, j, qMin, qMax, xNaj, yNaj, dNaj: integer;
    Q: array [0..1, 0..1] of integer;
    A: array [1..MaxN] of string;
begin
  { Preberimo šahovnico. }
  ReadLn(N); for i := 1 to N do ReadLn(A[i]);
  { Preizkusimo vse možne delitve. }
  dNaj := N * N + 1;
  for y := 1 to N - 1 do for x := 1 to N - 1 do begin
    { Ta delitev razdeli šahovnico na 4 dele. V tabeli Q pripravimo število figur na njih. }
    for i := 0 to 1 do for j := 0 to 1 do Q[i, j] := 0;
    for i := 1 to N do for j := 1 to N do if A[i, j] = '*' then
      Q[Ord(i <= y), Ord(j <= x)] := Q[Ord(i <= y), Ord(j <= x)] + 1;
    { Poiščimo največje in najmanjše število figur na posameznem delu. }
    qMin := N * N + 1; qMax := -1;
    for i := 0 to 1 do for j := 0 to 1 do begin
      if Q[i, j] < qMin then qMin := Q[i, j];
      if Q[i, j] > qMax then qMax := Q[i, j];
    end; {for i, j}
    { Če je to najboljša delitev doslej, si jo zapomnimo. }
    if qMax - qMin < dNaj then begin dNaj := qMax - qMin; xNaj := x; yNaj := y end;
  end; {for y, x}
  { Izpišimo rezultat. }
  WriteLn(dNaj, ' ', xNaj, ' ', yNaj);
end. {Sahovnica1}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>
#define MaxN 100
int main()
{
  int n, xNaj, yNaj, dNaj, i, j, x, y, q[2][2], qMin, qMax, d;
  char a[MaxN][MaxN + 2];
  /* Preberimo vhodne podatke. */
  scanf("%d\n", &n);
  for (i = 0; i < n; i++) fgets(a[i], n + 2, stdin);
  /* Preizkusimo vse možne položaje delilnih črt. */
  xNaj = -1; yNaj = -1; dNaj = n * n + 1;
  for (y = 0; y < n - 1; y++) for (x = 0; x < n - 1; x++)
  {
    /* Preštejmo figure na vseh štirih delih. */
    for (i = 0; i < 2; i++) for (j = 0; j < 2; j++) q[i][j] = 0;
```

```
    for (i = 0; i < n; i++) for (j = 0; j < n; j++)
      if (a[i][j] == '*') q[i <= y ? 0 : 1][j <= x ? 0 : 1]++;
    /* Poiščimo največje in najmanjše število figur na posameznem delu. */
    qMin = n * n + 1; qMax = -1;
    for (i = 0; i < 2; i++) for (j = 0; j < 2; j++) {
      if (q[i][j] < qMin) qMin = q[i][j];
      if (q[i][j] > qMax) qMax = q[i][j]; }
    /* Če je to najboljša rešitev doslej, si jo zapomnimo. */
    if (qMax - qMin < dNaj) xNaj = x, yNaj = y, dNaj = qMax - qMin;
  }
  /* Izpišimo najboljšo rešitev. */
  printf("%d %d %d\n", dNaj, xNaj + 1, yNaj + 1);
  return 0;
}
```

Do učinkovitejše rešitve pridemo, če opazimo, da so si posamezni deli šahovnice, ki nastanejo pri različnih delitvah, lahko precej podobni. Mislimo si na primer delitev, pri kateri je vodoravna delilna črta med y-to in $(y+1)$-vo vrstico, navpična delilna črta pa med x-tim in $(x+1)$-vim stolpcem. Če zdaj premaknemo navpično delilno črto za en stolpec desno, se vsak od štirih delov šahovnice le malo spremeni: celice $(x+1,1),(x+1,2),\ldots,(x+1,y)$ so bile prej v zgornjem desnem delu, zdaj pa so v zgornjem levem; in celice $(x+1,y+1),\ldots,(x+1,n)$ so bile prej v spodnjem desnem delu, zdaj pa so v spodnjem levem. Če bi torej imeli že od prej izračunano število figur v posameznem delu pri stari delitvi, bi bilo zdaj ta števila zelo lahko popraviti, da bi veljala za novo delitev: na novo moramo pregledati le celice v stolpcu $x+1$ in ko na njih odkrivamo figure, moramo povečevati števca figur na levih dveh delih in zmanjševati števca figur na desnih dveh delih. Tako moramo pri vsaki delitvi na novo pregledati le $O(n)$ celic, ne pa več $O(n^2)$; časovna zahtevnost celotnega postopka bi se s tem zmanjšala z $O(n^4)$ na $O(n^3)$.

Še ena izboljšava pa je ta, da opazimo, da se bomo s celicami v stolpcu $x+1$ srečali velikokrat — vsakič, ko bomo prehajali z delitve (x,y) na delitev $(x+1,y)$; to se bo torej zgodilo $(n-1)$-krat, ker si lahko y (položaj vodoravne delilne črte) izberemo na toliko načinov. Skupine celic, o katerih smo govorili zgoraj — torej $(x+1,1),\ldots,(x+1,y)$ in $(x+1,y+1),\ldots,(x+1,n)$ — so si pri različnih y zelo podobne; na primer, če povečamo y za 1 (na $y+1$), se spremeni le to, da se prva celica iz spodnje skupine, torej celica $(x+1,y+1)$, premakne v zgornjo skupino. Če bi si torej zapomnili število figur za zgornjo in za spodnjo skupino (pri nekem y, za vse možne x), bi lahko ta števila ob povečevanju y-a zelo preprosto popravljali (ko se poveča y na $y+1$, gremo po celicah iz vrstice $y+1$; kjer vidimo figuro, povečamo število figur v zgornji skupini za tisti stolpec in zmanjšamo število figur v spodnji skupini za tisti stolpec). Spodnji program hrani te podatke o številu figur v zgornji in spodnji skupini celic vsakega stolpca v tabeli SC. Časovna zahtevnost naše rešitve je zdaj le še $O(n^2)$. Bolje pravzaprav že ne more več biti, saj potrebujemo toliko časa že tudi za samo branje šahovnice.

```
program Sahovnica2;
var N, x, y, i, j, qMin, qMax, xNaj, yNaj, dNaj, nNad, nPod: integer;
    A: array [1..MaxN] of string;
    SC: array [0..1, 1..MaxN] of integer;
    Q: array [0..1, 0..1] of integer;
```

```
begin
  { Preberimo šahovnico. }
  ReadLn(N); for i := 1 to N do ReadLn(A[i]);
  { Preštejmo figure po stolpcih in na celi šahovnici. }
  for x := 1 to N do begin SC[0, x] := 0; SC[1, x] := 0 end;
  for y := 1 to N do for x := 1 to N do if A[y, x] = '*' then SC[1, x] := SC[1, x] + 1;
  nNad := 0; nPod := 0; for x := 1 to N do nPod := nPod + SC[1, x];
  { Preglejmo vse možne položaje delitvenih črt. }
  dNaj := N * N + 1;
  for y := 1 to N - 1 do begin
    { SC[0, x] je število figur na poljih A[1, x], ..., A[y - 1, x].
      SC[1, x] je število figur na poljih A[y, x], ..., A[N, x].
      nNad je vsota SC[0, x] po vseh x od 1 do N;
      nPod je vsota SC[1, x] po vseh x od 1 do N. }
    { Popravimo zdaj te števce, da bo vrstica y prišla nad vodoravno delilno črto. }
    for x := 1 to N do if A[y, x] = '*' then begin
      SC[0, x] := SC[0, x] + 1; nNad := nNad + 1;
      SC[1, x] := SC[1, x] - 1; nPod := nPod - 1;
    end; {for x}
    { Preizkusimo vse možne položaje navpične delilne črte. }
    Q[0, 0] := 0; Q[1, 0] := 0;
    for x := 1 to N - 1 do begin
      { V Q[i, 0] je trenutno število figur na zgornjem levem in spodnjem
        levem delu, če je navpična delilna črta med stolpcema x - 1 in x.
        Popravimo ju, ker bo ker bo navpična črta zdaj med stolpcema x in x + 1. }
      Q[0, 0] := Q[0, 0] + SC[0, x]; Q[1, 0] := Q[1, 0] + SC[1, x];
      { Izračunajmo še število figur na zgornjem desnem in spodnjem desnem delu. }
      Q[0, 1] := nNad - Q[0, 0]; Q[1, 1] := nPod - Q[1, 0];
      { Ocenimo to delitev. }
      qMin := N * N + 1; qMax := -1;
      for i := 0 to 1 do for j := 0 to 1 do begin
        if Q[i, j] < qMin then qMin := Q[i, j];
        if Q[i, j] > qMax then qMax := Q[i, j];
      end; {for i, j}
      { Najboljšo delitev si zapomnimo. }
      if qMax - qMin < dNaj then
        begin dNaj := qMax - qMin; xNaj := x; yNaj := y end;
    end; {for x}
  end; {for y}
  WriteLn(dNaj, ' ', xNaj, ' ', yNaj); { Izpišimo rezultat. }
end. {Sahovnica2}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>
#define MaxN 100

int main()
{
  int n, xNaj, yNaj, dNaj, i, j, x, y, q[2][2], sc[2][MaxN], qMin, qMax, d, nNad, nPod;
  char a[MaxN][MaxN + 2];
  /* Preberimo vhodne podatke. */
  scanf("%d\n", &n);
  for (i = 0; i < n; i++) fgets(a[i], n + 2, stdin);
```

```
  /* Preštejmo figure po stolpcih. */
  for (x = 0; x < n; x++) sc[0][x] = 0, sc[1][x] = 0;
  for (y = 0; y < n; y++) for (x = 0 ; x < n; x++)
    if (a[y][x] == '*') sc[1][x]++;
  nNad = 0; nPod = 0; for (x = 0; x < n; x++) nPod += sc[1][x];

  /* Preizkusimo vse možne položaje delilnih črt. */
  xNaj = -1; yNaj = -1; dNaj = n * n + 1;
  for (y = 0; y < n - 1; y++) for (x = 0; x < n - 1; x++)
  {
    /* SC[0][x] je število figur na poljih a[0][x], ..., a[y - 1][x].
       SC[1][x] je število figur na poljih a[y][x], ..., a[n - 1][x].
       nNad je vsota SC[0][x] po vseh x, nPod pa je vsota SC[1][x] po vseh x.
       Popravimo te vrednosti tako, da bo vrstica a[y] prišla pod delilno črto. */
    for (x = 0; x < n; x++) if (a[y][x] == '*')
      sc[0][x]++, sc[1][x]--, nNad++, nPod--;
    /* Preizkusimo vse možne položaje delilne črte. */
    q[0][0] = 0; q[1][0] = 0; q[0][1] = nNad; q[1][1] = nPod;
    for (x = 0; x < n; x++)
    {
      /* Popravimo števce figur na vseh štirih delih. */
      q[0][0] += sc[0][x]; q[0][1] -= sc[0][x];
      q[1][0] += sc[1][x]; q[1][1] -= sc[1][x];

      /* Poiščimo največje in najmanjše število figur na posameznem delu. */
      qMin = n * n + 1; qMax = -1;
      for (i = 0; i < 2; i++) for (j = 0; j < 2; j++) {
        if (q[i][j] < qMin) qMin = q[i][j];
        if (q[i][j] > qMax) qMax = q[i][j]; }

      /* Če je to najboljša rešitev doslej, si jo zapomnimo. */
      if (qMax - qMin < dNaj) xNaj = x, yNaj = y, dNaj = qMax - qMin;
    }
  }
  /* Izpišimo najboljšo rešitev. */
  printf("%d %d %d\n", dNaj, xNaj + 1, yNaj + 1);
  return 0;
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Nebotičniki

Zemljišča pri tej nalogi so majhna, možni razpon višin (od c_1 do c_2) tudi, tako da lahko nalogo rešimo kar z rekurzijo, ki našteje vse možne razporede višin nebotičnikov. Zemljišče bomo pregledovali po vrsticah od severa proti jugu, v vsaki vrstici pa parcele od zahoda proti vzhodu. Kakšne so omejitve za višino nebotičnika na parceli (x, y)? Navzgor ga omejujeta višina severnega in zahodnega soseda, od katerih mora biti naš nebotičnik vsaj eno nadstropje nižji; če teh sosedov nima, je zgornja meja njegove višine pač c_2. Navzdol pa ga omejuje dejstvo, da bo treba zgraditi še nebotičnike $(x + 1, y), (x + 2, y), \ldots, (w, y)$ v isti vrstici in nato še $(w, y + 1)$, $(w, y + 2), \ldots, (w, h)$ v zadnjem stolpcu; vsak od teh mora biti nižji od prejšnjega, zadnji pa mora biti še vedno visok vsaj c_1 nadstropij. Nebotičnik (x, y) mora biti torej visok vsaj $c_1 + (h - y) + (w - x)$ nadstropij.

```
program Neboticniki;
const MaxW = 10; MaxH = 10;
var w, h, c1, c2, Stevec: integer;
    c: array [1..MaxH, 1..MaxW] of integer;

procedure Rekurzija(x, y: integer);
var MinC, MaxC, cc: integer;
begin
  MinC := c1 + (w - x) + (h - y);
  if y = 1 then MaxC := c2 else MaxC := c[y - 1, x] - 1;
  if x > 1 then if MaxC > c[y, x - 1] - 1 then MaxC := c[y, x - 1] - 1;
  for cc := MinC to MaxC do begin
    c[y, x] := cc;
    if (x = w) and (y = h) then Stevec := Stevec + 1
    else if x = w then Rekurzija(1, y + 1)
    else Rekurzija(x + 1, y);
  end; {for cc}
end; {Rekurzija}

var T: text;
begin {Neboticniki}
  Assign(T, 'neboticniki.in'); Reset(T); ReadLn(T, w, h, c1, c2); Close(T);
  Stevec := 0;
  Rekurzija(1, 1);
  Assign(T, 'neboticniki.out'); Rewrite(T); WriteLn(T, Stevec); Close(T);
end. {Neboticniki}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>
#define MaxW 10
#define MaxN 10

int w, h, c1, c2, a[MaxW][MaxN], rezultat;

void Rekurzija(int x, int y)
{
  int c, cMin, cMax;
  cMin = c1 + (w - 1 - x) + (h - 1 - y);
  cMax = (x == 0) ? c2 + 1 : a[y][x - 1];
  if (y > 0 && a[y - 1][x] < cMax) cMax = a[y - 1][x];
```

```
  /* Dopustne višine na tem zemljišču so cMin, ..., (cMax − 1).
     Preizkusimo vse možnosti, nadaljujmo z rekurzijo. */
  for (c = cMin; c < cMax; c++) {
    a[y][x] = c;
    if (y == h − 1 && x == w − 1) rezultat++;
    else if (x == w − 1) Rekurzija(0, y + 1);
    else Rekurzija(x + 1, y); }
}
int main()
{
  /* Preberimo vhodno datoteko. */
  FILE *f = fopen("neboticniki.in", "rt");
  fscanf(f, "%d %d %d %d", &w, &h, &c1, &c2); fclose(f);
  /* Izračunajmo rezultat in ga izpišimo. */
  rezultat = 0; Rekurzija(0, 0);
  f = fopen("neboticniki.out", "wt");
  fprintf(f, "%d\n", rezultat); fclose(f); return 0;
}
```

Za zelo majhne testne primere, kot so tisti pri naši nalogi, je ta rešitev dovolj hitra. Kot zanimivost pa si oglejmo še učinkovitejšo rešitev, s katero lahko dovolj hitro obdelamo tudi malo večja zemljišča (zelo velikih pa žal vendarle ne). Zaradi lažjega pisanja si mislimo (brez izgube za splošnost), da je $c_1 = 0$ in $c_2 = c - 1$ za neko število c.

Ker mora biti vsak nebotičnik nižji od svojega zahodnega soseda, so nebotičniki v posamezni vrstici vsi različno visoki. Na njihove višine lahko gledamo torej kot na množico A z w elementi (izbranimi iz množice $C := \{0, \ldots, c-1\}$) — glede vrstnega reda nebotičnikov nimamo nobene izbire: urediti jih bo treba padajoče po višini. Takih množic pa je $\binom{c}{w}$ — le toliko je torej različnih možnih stanj ene vrstice.

Zdaj si lahko zastavimo podprobleme oblike: „na koliko načinov je mogoče določiti višine nebotičnikov v prvih k vrsticah, če hočemo, da je k-ta vrstica v stanju A?“ Označimo to število s $f(k, A)$. Pri $k = 1$ je jasno, da je $f(1, A) = 1$ za vsako primerno A (torej tako, ki je podmnožica C in ima točno w elementov). Pri večjih k pa lahko $f(k, A)$ izračunamo tako, da pregledamo vsa možna stanja prejšnje vrstice: če je množica B taka, da lahko vrstico A postavimo tik pod B (in bo izpolnjen pogoj, da mora biti vsak nebotičnik v A nižji od svojega severnega soseda v B), to pomeni, da je med razporedi prvih k vrstic, ki imajo v k-ti vrstici stanje A, $f(k-1, B)$ takih razporedov, ki imajo v $(k-1)$-vi vrstici stanje B. Tako smo dobili $f(k, A) = \sum_B f(k-1, B)$, pri čemer mora iti vsota po vseh takih B, ki jih lahko postavimo tik nad A. Tako smo dobili nekakšno rekurzivno formulo za f, učinkovito pa jo lahko računamo tako, da si njene vrednosti shranjujemo v neko tabelo, tako da jih ne bo treba računati po večkrat. Ko imamo enkrat potabelirane vse $f(k-1, B)$, ne bo težko izračunati vseh $f(k, A)$ in tako naprej.

Na koncu nam je pravzaprav vseeno, kakšna točno je zadnja vrstica nebotičnikov, zato moramo izpisati $\sum_A f(h, A)$ — vsota naj gre po vseh možnih A (vseh takih podmnožicah C-ja, ki imajo w elementov).

Ostalo nam je le še nekaj tehnikalij. Na primer: kako učinkovito potabelirati vrednosti $f(k, A)$ za vse A? Videli smo že, da je takih A-jev natanko $\binom{c}{w}$; koristno bi torej bilo, če bi znali množice A oštevilčiti od 0 do $\binom{c}{w} - 1$ — te številke bi lahko uporabljali kot indekse v tabelo. To pa niti ni težko. Mislimo si vse primerne množice

A (torej take z w elementi); v vsaki od njej števila zapišimo padajoče, da iz množice nastane urejena w-terica; te w-terice zdaj uredimo leksikografsko. Recimo, da za neko A, ki nas zanima, nastane w-terica $(a_1, a_2, \ldots, a_w)$. Koliko w-teric je pred njo v leksikografskem vrstnem redu? Najprej vse tiste, ki imajo na prvem mestu nek $b_1 < a_1$; takih je $\sum_{b_1=w-1}^{a_1-1} \binom{b_1}{w-1}$; potem vse tiste, ki imajo na prvem mestu sicer a_1, na drugem mestu pa imajo nek $b_2 < a_2$; takih je $\sum_{b_2=w-2}^{a_2-1} \binom{b_2}{w-2}$; in tako naprej.

Množice A lahko naštevamo z rekurzijo kar v takšnem leksikografskem vrstnem redu (to počne podprogram PreglejVrstice spodaj). Pri vsaki A moramo potem pregledati vse primerne B, torej take, za katere velja (če obe množici zapišemo kot padajoče urejeni w-terici) $a_1 < b_1$, $a_2 < b_2$ in tako naprej, da bo res lahko vrstica A stala južno od vrstice B. Tudi te B-je lahko naštevamo z rekurzijo (podprogram PreglejPrejVrstice, pri tem pa si še pomagamo s formulami iz prejšnjega odstavka, da za vsak B sproti izračunamo tudi indeks, s katerim bomo lahko v tabeli poiskali vrednost $f(k-1, B)$.

```
function Resi(w, h, c1, c2: integer): integer;
type TabelaP = ↑TabelaT;
     TabelaT = packed array [0..99999999] of integer;
var Binom: array [0..MaxC, 0..MaxC] of integer;
    nStanj, cc: integer;
    r, rp: TabelaP;
    v, vp: array [1..MaxW] of integer;

  procedure PreglejPrejVrstice(x, s, sp: integer);
  var MaxC, a: integer;
  begin
    { Pregledali bi radi vrstice vp, ki se lahko pojavijo tik nad vrstico v.
      s je številka vrstice v v leksikografskem vrstnem redu.
      V tabeli vp smo trenutno že izbrali prvih x − 1 elementov;
      iz njih sledi, da je pred njo v leksikografskem vrstnem redu že sp drugih vrstic. }
    if x = 1 then MaxC := cc − 1 else MaxC := vp[x − 1] − 1;
    { Možne vrednosti x-tega elementa so vse od w − x naprej;
      nas seveda zanimajo le tiste, ki so večje od v[x] (da bo lahko vrstica vp
      stala nad vrstico v), vendar moramo vsaj prešteti, koliko je vrstic, ki imajo v
      x-tem elementu kakšno manjšo vrednost, da bomo lahko povečali sp. }
    for a := w − x to MaxC do begin
      if a > v[x] then begin
        vp[x] := a;
        if x = w then r↑[s] := r↑[s] + rp↑[sp]
        else PreglejPrejVrstice(x + 1, s, sp);
      end; {if}
      sp := sp + Binom[a, w − x];
    end; {for a}
  end; {PreglejPrejVrstice}

  procedure PreglejVrstice(x, s: integer);
  var MaxC, a: integer;
  begin
    { Pri pregledovanju vseh možnih stanj vrstice v smo trenutno
      pri tistih, ki se začnejo na v[1], ..., v[x − 1].
      Pred njimi je v leksikografskem vrstnem redu še s drugih vrstic. }
    if x = 1 then MaxC := cc − 1 else MaxC := v[x − 1] − 1;
    for a := w − x to MaxC do begin
      v[x] := a;
      if x = w then begin r↑[s] := 0; PreglejPrejVrstice(1, s, 0); end
```

```
        else PreglejVrstice(x + 1, s);
        s := s + Binom[a, w − x];
      end; {for a}
  end; {PreglejVrstice}

var rTemp: TabelaP;
    i, j, k, s: integer; Rezultat: integer;
begin {Resi}
  cc := c2 − c1 + 1;
  if cc < w + h − 1 then begin Resi := 0; exit end;

  { Pripravimo si tabelo binomskih koeficientov (Pascalov trikotnik). }
  Binom[0, 0] := 1;
  for i := 1 to cc do for j := 0 to i do begin
    if j > w then break;
    if (j = 0) or (j = i) then Binom[i, j] := 1
    else Binom[i, j] := Binom[i − 1, j] + Binom[i − 1, j − 1];
  end; {for i, j}

  { Koliko je možnih stanj ene vrstice? }
  nStanj := Binom[cc, w];
  { Pripravimo dve tabeli; v eni bo število razporedov za k − 1 vrstic,
    v drugi bomo računali število razporedov za k vrstic. }
  GetMem(r, nStanj * SizeOf(integer));
  GetMem(rp, nStanj * SizeOf(integer));

  { Rešitve za k = 1 (ena sama vrstica). }
  for s := 0 to nStanj − 1 do r↑[s] := 1;

  { Rešitve za več vrstic. }
  for k := 2 to h do begin
    rTemp := r; r := rp; rp := rTemp;
    PreglejVrstice(1, 0);
  end; {for k}

  { Izračunajmo rezultat. }
  Rezultat := 0;
  for s := 0 to nStanj − 1 do Rezultat := Rezultat + r↑[s];

  { Pospravimo za sabo. }
  FreeMem(r, nStanj * SizeOf(integer));
  FreeMem(rp, nStanj * SizeOf(integer));
  Resi := Rezultat;
end; {Resi}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>
#define MaxW 10
#define MaxN 10
#define MaxC 20

int w, h, c1, c2, c;
int binom[MaxC + 1][MaxC + 1];
int v[MaxW], vp[MaxW];
int *r, *rp, nStanj;

void PreglejPrejVrstice(int x, int s, int sp)
{
  /* Pregledali bi radi vrstice vp, ki lahko stojijo tik nad v.
     Položaj vrstice v v leksikografskem vrstnem redu je s.
     V vrstico vp zdaj vpisujemo od mesta x naprej,
```

```
      leksikografsko manjših vrstic pa je sp. */
   int a, cMax = (x == 0) ? c : vp[x − 1], i;
   for (a = w − 1 − x; a < cMax; a++) {
      /* Da bo vp res lahko stala nad v, mora biti
         vsak element večji od istoležnega elementa vrstice v. */
      if (a > v[x]) {
         vp[x] = a;
         if (x == w − 1) r[s] += rp[sp];
         else PreglejPrejVrstice(x + 1, s, sp); }
      sp += binom[a][w − x − 1]; }
}
void PreglejVrstice(int x, int s)
{
   /* Vrstico v je treba dopolniti od indeksa x naprej.
      Položaj v-ja v leksikografskem vrstnem redu je s.
      Možne višine nebotičnikov si mislimo od 0 do c − 1, ne od c1 do c2. */
   int a, cMax = (x == 0) ? c : v[x − 1];
   for (a = w − 1 − x; a < cMax; a++) {
      v[x] = a;
      if (x == w − 1) { r[s] = 0; PreglejPrejVrstice(0, s, 0); }
      else PreglejVrstice(x + 1, s);
      /* Ostala so nam še zemljišča od x+1 do w-1, torej w-1-x zemljišč,
         zanje pa moramo izbrati višine, nižje od a (torej so možnosti od 0 do a − 1). */
      s += binom[a][w − x − 1]; }
}
int main()
{
   int i, j, k, *t, s, rezultat;
   /* Preberimo vhodno datoteko. */
   FILE *f = fopen("neboticniki.in", "rt");
   fscanf(f, "%d %d %d %d", &w, &h, &c1, &c2); fclose(f);
   /* Pripravimo tabelo binomskih koeficientov. */
   binom[0][0] = 1;
   for (i = 1; i <= MaxC; i++) for (j = 0; j <= i; j++)
      binom[i][j] = (j == 0 || j == i) ? 1 : binom[i − 1][j − 1] + binom[i − 1][j];
   /* Pripravimo tabeli za shranjevanje rezultatov. */
   c = c2 − c1 + 1; nStanj = binom[c][w];
   r = (int *) malloc(nStanj * sizeof(int));
   rp = (int *) malloc(nStanj * sizeof(int));
   /* Rešimo podprobleme z eno vrstico. */
   for (s = 0; s < nStanj; s++) r[s] = 1;
   /* Rešimo večje podprobleme. */
   for (k = 1; k < h; k++) {
      t = r; r = rp; rp = t;
      PreglejVrstice(0, 0); }
   /* Izračunajmo rezultat in ga izpišimo. */
   for (s = 0, rezultat = 0; s < nStanj; s++) rezultat += r[s];
   f = fopen("neboticniki.out", "wt");
   fprintf(f, "%d\n", rezultat); fclose(f);
   free(r); free(rp); return 0; /* Pospravimo za sabo. */
}
```

Kakšna je časovna zahtevnost tega postopka? Izračunati moramo $f(k, A)$ za $O(h)$ vrednosti k in za $O(\binom{c}{w})$ vrednosti A; pri vsakem takem paru (k, A) moramo pregle-

dati vse primerne B, ki jih je tudi $O(\binom{c}{w})$; tudi če upoštevamo, da bi utegnilo biti klicev PreglejPrejVrstice več, kot je primernih B-jev (ker se kliče ta podprogram tudi za delno izpolnjene vrstice, torej za x < w), teh klicev gotovo ni več kot $\binom{c+1}{w}$. Tako vidimo, da ima naš algoritem časovno zahtevnost največ $O(h\binom{c}{w}\binom{c+1}{w}) = O(wh\binom{c}{w}^2)$. To sicer še vedno narašča eksponentno hitro v odvisnosti od velikosti vhodnega problema, vendar je vseeno precej hitreje od preproste rekurzivne rešitve, ki smo jo videli na začetku.

2. Pleskarji

Nalogo lahko naivno poskušamo rešiti tako, da simuliramo sprehod vsakega od pleskarjev (po vrsti) po stopnicah. Očitno to ne bo dovolj učinkovito: če hočemo ves čas v spominu udobno hraniti stanje vseh 10^9 stopnic, potrebujemo tabelo velikosti 10^9 oziroma približno 1 GB spomina. To bi še šlo, vendar pa moramo v najslabšem primeru to tabelo „prehoditi“ milijonkrat, kolikor je pleskarjev. To pa bi na današnjih računalnikih trajalo kakšno leto...[4]

Rešitev lahko bistveno pospešimo, če naredimo dve izboljšavi. Prva je ta, da odmislimo večino pleskarjev. Če namreč dva pleskarja delata korake enake velikosti, je pomembno le, kakšne barve je bil tisti, ki je prišel pozneje, saj bo popolnoma prekril sledi zgodnejšega kolega in bo rezultat enak, kot če kolega sploh ne bi bilo. V resnici moramo torej opazovati največ 10 pleskarjev: v vsaki skupini pleskarjev z določeno dolžino koraka samo tistega, ki se pojavi nazadnje.

Druga stvar, ki jo moramo opaziti, je to, da se vzorec na stopnicah ponavlja. Na $(10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1)$-to stopnico bodo namreč stopili prav vsi pleskarji, saj je število deljivo z vsako velikostjo koraka. Od te stopnice naprej se ponovi zgodba od začetka do sem, saj je situacija na tej stopnici enaka kot na stopnici 0 na dnu stopnišča (nanjo tudi stopijo vsi pleskarji). Torej je dovolj, če opazujemo le spodnjih $10{\cdot}9{\cdot}8{\cdot}7{\cdot}6{\cdot}5{\cdot}4{\cdot}3{\cdot}2{\cdot}1 = 3\,628\,800$ stopnic in iz barv, ki jih opazimo tam, sklepamo še na barve na višjih stopnicah. Namesto zgornjega produkta lahko pravzaprav postavimo še precej manjšo številko: prva stopnica, na katero bodo stopili vsi pleskarji, je najmanjši skupni večkratnik števil od 1 do 10, to je $2^3 \cdot 3^2 \cdot 5 \cdot 7 = 2520$. Deset pleskarjev na 2520 stopnicah — te pa lahko očitno opazujemo korak za korakom brez kakršnihkoli težav s pomnilnikom ali časom.

```
program Pleskarji;
const MaxKorak = 10;
      Lcm = 2*2*2*3*3*5*7; { najmanjši skupni večkratnik števil 1, ..., MaxKorak }
      StBarv = 26;
var T: text;
    Barva, Zadnji: array [1..MaxKorak] of integer;
    i, k, kk, b, nPleskarjev, nStopnic: integer;
    Stopnice: array [0..Lcm - 1] of integer;
```

[4] Porabo pomnilnika bi lahko zmanjšali tudi tako, da bi stopnišče razdelili na manjše dele, npr. po milijon stopnic. Potem najprej simuliramo vse pleskarje po vrsti, vendar vsakega le tako daleč, dokler ne vstopi v drugi del (torej čez milijonto stopnico); pri vsakem si tudi zapomnimo, katera je prva stopnica, ki jo bo prehodil v drugem delu. Na koncu torej za prvi milijon stopnic vemo, kakšne barve bo katera, nato pa se lahko lotimo simulacije dogajanja na naslednjem milijonu stopnic (spet gremo po vseh pleskarjih po vrsti) in tako naprej. S tem smo porabo pomnilnika močno zmanjšali, porabe časa pa seveda ne. Če je število pleskarjev zelo majhno (npr. do 10), lahko na ta način kljub vsemu dovolj hitro obdelamo tja do sto milijonov stopnic.

```
    StZBarvo: array [0..StBarv] of integer;
begin
  { Preberimo vhodne podatke. Za vsak korak si zapomnimo barvo
    in indeks zadnjega pleskarja s tem korakom. }
  Assign(T, 'pleskarji.in'); Reset(T);
  ReadLn(T, nStopnic, nPleskarjev);
  for k := 1 to MaxKorak do begin Barva[k] := 0; Zadnji[k] := nPleskarjev + 1 end;
  for i := 1 to nPleskarjev do begin
    ReadLn(T, k, b);
    Barva[k] := b; Zadnji[k] := i;
  end; {for i}
  Close(T);
  { Na začetku so vse stopnice brezbarvne (barve 0). }
  for i := 0 to Lcm - 1 do Stopnice[i] := 0;
  { Simulirajmo pleskarje na prvih Lcm stopnicah.
    Za vsak korak bomo simulirali le zadnjega pleskarja. }
  while true do begin
    { Poiščimo najzgodnejšega takega pleskarja, ki ga še nismo simulirali. }
    k := 1; for kk := 2 to MaxKorak do if Zadnji[kk] < Zadnji[k] then k := kk;
    if Zadnji[k] > nPleskarjev then break; { Simulirali smo že vse pleskarje. }
    { Pobarvajmo vse stopnice, ki jih obišče ta pleskar. }
    i := 0;
    while i < Lcm do begin Stopnice[i] := Barva[k]; i := i + k end;
    Zadnji[k] := nPleskarjev + 1; { Označimo, da smo ga že odsimulirali. }
  end; {while}
  { Preštejmo, kolikokrat se pojavi kakšna barva na prvih Lcm stopnicah. }
  for b := 0 to StBarv do StZBarvo[b] := 0;
  for i := 0 to Lcm - 1 do
    begin b := Stopnice[i]; StZBarvo[b] := StZBarvo[b] + 1 end;
  { Kolikokrat se vzorec prvih Lcm stopnic ponovi v vseh nStopnic stopnicah? }
  for b := 1 to StBarv do StZBarvo[b] := StZBarvo[b] * (nStopnic div Lcm);
  { Ker nStopnic ni nujno večkratnik Lcm, upoštevajmo zdaj še preostale stopnice. }
  for i := 0 to (nStopnic mod Lcm) - 1 do
    begin b := Stopnice[i]; StZBarvo[b] := StZBarvo[b] + 1 end;
  { Izpišimo rezultate. }
  Assign(T, 'pleskarji.out'); Rewrite(T);
  for b := 1 to StBarv do WriteLn(T, StZBarvo[b]);
  Close(T);
end. {Pleskarji}
```

Še rešitev v C-ju:

```
#include <stdio.h>
#define StBarv 26 /* barve so 1..26; 0 = brezbarvne stopnice */
#define MaxKorak 10
/* Lcm je najmanjši skupni večkratnik števil 1, 2, ..., MaxKorak. */
#define Lcm (2 * 2 * 2 * 3 * 3 * 5 * 7)

int main()
{
  int barva[MaxKorak + 1], zadnji[MaxKorak + 1], stopnice[Lcm];
  int nStopnic, nPleskarjev, i, j, b, k, stZBarvo[StBarv + 1];
  /* Preberimo vhodno datoteko. */
  FILE *f = fopen("pleskarji.in", "rt");
  fscanf(f, "%d %d", &nStopnic, &nPleskarjev);
```

```
  for (k = 1; k <= MaxKorak; k++) zadnji[k] = nPleskarjev;
  for (i = 0; i < nPleskarjev; i++) {
    fscanf(f, "%d %d", &k, &b); zadnji[k] = i; barva[k] = b; }
  fclose(f);
  /* Na začetku so vse stopnice brezbarvne. */
  for (j = 0; j < Lcm; j++) stopnice[j] = 0;
  /* Simulirajmo pleskarje na prvih Lcm stopnicah. */
  for ( ; ; )
  {
    /* Poiščimo najzgodnejšega še nesimuliranega pleskarja. */
    for (k = 1, i = 2; i <= MaxKorak; i++)
      if (zadnji[i] < zadnji[k]) k = i;
    if (zadnji[k] >= nPleskarjev) break; /* Odsimulirali smo že vse. */
    /* Odsimulirajmo tega pleskarja. */
    for (j = 0; j < Lcm; j += k) stopnice[j] = barva[k];
    zadnji[k] = nPleskarjev; /* Označimo, da je odsimuliran. */
  }
  /* Preštejmo barve na prvih Lcm stopnicah. */
  for (b = 1; b <= StBarv; b++) stZBarvo[b] = 0;
  for (j = 0; j < Lcm; j++) stZBarvo[stopnice[j]]++;
  /* Kolikokrat gre Lcm v nStopnic? */
  for (b = 1; b <= StBarv; b++) stZBarvo[b] *= nStopnic / Lcm;
  /* Preglejmo še preostanek stopnic (če nStopnic ni večkratnik Lcm). */
  for (j = 0; j < nStopnic % Lcm; j++) stZBarvo[stopnice[j]]++;
  /* Izpišimo rezultate. */
  f = fopen("pleskarji.out", "wt");
  for (b = 1; b <= StBarv; b++) fprintf(f, "%d\n", stZBarvo[b]);
  fclose(f); return 0;
}
```

Naslednja tabela kaže velikosti desetih testnih primerov, ki smo jih uporabili na našem tekmovanju:

Št. stopnic	Št. pleskarjev
1	1
623 327	5
1 000 000	7
9 910 472	198 584
99 174 373	906 238
598 144 835	689 996
989 229 976	239 330
909 880 112	317 450
996 184 633	484 454
946 379 795	233 709

Rešitev, kakršno smo opisali zgoraj, reši vse te primere zelo hitro. Slabša rešitev, ki bi simulirala največ 10 pleskarjev (po enega za vsako možno dolžino koraka), vendar na celem stopnišču, bi uspela dovolj hitro (torej v desetih sekundah na našem ocenjevalnem računalniku) rešiti prvih pet primerov, ostalih pa načeloma ne. Še slabša rešitev, ki bi simulirala vse pleskarje na celem stopnišču, bi dovolj hitro rešila le prve tri testne primere.

3. Pogrešane osebe

Naj bo s naš dolgi niz, v katerem iščemo pojavitve kratkih nizov $p^1, \ldots, p^n$ (slednjim bomo rekli tudi *vzorci*). Posamezne črke nizov bomo označevali z indeksi spodaj: s_i je i-ta črka niza s ipd. Dolžino niza x označimo z $|x|$; posebej naj bo $m = |s|$.

Naivna rešitev in nekaj izboljšav. Najbolj naivna rešitev naše naloge je očitno ta, da se postavimo na vsak možni začetni položaj v nizu s in tam za vse možne vzorce pogledamo, če se slučajno pojavljajo na tem mestu v nizu s. V ta namen po vrsti primerjamo črke niza s in črke vzorca, dokler ne opazimo neujemanja (ali pa pridemo do konca vzorca).

```
for i := 1 to m do
  for j := 1 to n do begin
    k := 0;
    while k < |p^j| do
      if s_{i+k} = p^j_{k+1} then k := k + 1 else break;
    if k = |p^j| then
      vzorec p^j se pojavlja v nizu s na mestu i (torej s_i s_{i+1} ... s_{i+|p^j|-1} = p^j);
  end; (* for *)
```

Naša najbolj notranja zanka (**while** k) se ustavi, čim opazi neujemanje; to je dobro, sploh pri nizih iz naravnih jezikov, kjer lahko pričakujemo, da so tako v nizu s kot v vzorcih kolikor toliko pogosto zastopane številne črke abecede. Katerakoli črka že je s_i, večina vzorcev se najbrž sploh ne začne na to črko, zato bo naša zanka **while** pri večini vzorcev odnehala že v prvi iteraciji. To je, kot rečeno, po eni strani dobro, ker se nam s takimi vzorci ni treba ukvarjati še naprej; po drugi strani pa je to velika potrata: zakaj se sploh ukvarjamo z vzorci, pri katerih se niti prva črka ne ujema s trenutno črko našega niza s? Če imamo 25 črk in so vse enako pogoste, se (pri vsakem konkretnem i) s 96 % vzorcev sploh ne splača ukvarjati, ker se ne začnejo na pravo črko. (V praksi niso vse črke enako pogoste, naša ugotovitev pa še vseeno drži: večina vzorcev se ne začne na črko s_i, pa katerakoli črka to že pač je.)

Torej si je pametno pripraviti za vsako črko seznam vzorcev, ki se začnejo nanjo. Ko se potem sprehajamo po nizu s, moramo pri vsakem i pregledati le še vzorce, ki se res začnejo na s_i.

```
for vsako črko c do prvi[c] := 0;
for j := 1 to n do begin nasl[j] := prvi[p^j_1]; prvi[p^j_1] := j end;
for i := 1 to m do begin
  j := prvi[s_i];
  while j > 0 do begin
    k := 1;
    while k < |p^j| do
      if s_{i+k} = p^j_{k+1} then k := k + 1 else break;
    if k = |p^j| then
      vzorec p^j se pojavlja v nizu s na mestu i;
    j := nasl[j];
  end; (* while *)
end; (* for *)
```

Sezname vzorcev smo predstavili z dvema tabelama: $prvi[c]$ je indeks prvega[5] vzorca, ki se začne na črko c; $nasl[j]$ pa je indeks naslednjega vzorca, ki se začne na isto črko kot p^j (torej na črko p_1^j). Pri vsakem začetnem položaju niza s gremo potem z zanko „**while** $j > 0$“ po seznamu vzorcev, ki se začnejo na s_i. Pri vsakem od njih potem primerjamo črke niza s črkami vzorca, enako kot prej, le da lahko začnemo pri $k = 1$ namesto $k = 0$, saj zdaj vemo, da se prva črka že ujema.

Izkaže se (glej tabelo na str. 68), da ta drobna izboljšava že močno pospeši našo rešitev; ne pa še dovolj. Kljub vsemu je vzorcev veliko, različnih črk pa malo in veliko se jih začne na isto črko. Naraven naslednji korak je, da si pripravimo po en seznam za vsak par črk: v njem bodo vzorci, ki se začnejo na tidve črki. Vzorce, ki so dolgi le eno črko, lahko obravnavamo kot poseben primer.

To je že skoraj dovolj hitro za naše potrebe. Če gremo še en korak naprej in si pripravimo za vsako trojico črk seznam vzorcev, ki se začnejo na te tri črke, pa smo že na konju. (Zdaj moramo kot poseben primer obravnavati tudi vzorce dolžine 2, ne le vzorce dolžine 1.) Tabela *prvi* je zdaj že precej velika: imeti mora po en element za vsako možno trojico črk. Besedilo naloge pravi, da pridejo v poštev le črke s številskimi kodami od 32 do 127, torej potrebujemo tabelo $96^3 = 884\,736$ elementov; če smo bolj ležerni in zaradi lažjega preračunavanja iz trojic znakov v indekse ravnamo tako, kot da bi bile možne vse kode od 0 do 127, potrebujemo tabelo velikosti $128^3 = 2\,097\,152$ (tako počne tudi spodnji program). Če je vsak element dolg 4 byte, je to skupaj 8 MB, kar je še vedno znosno. Lahko pa se zadovoljimo tudi z manjšo razpršeno tabelo: velika naj bo recimo p elementov in vzorce, ki se začnejo na trojico $c_1c_2c_3$, hranimo v seznamu, na katerega kaže $prvi[(c_1 \cdot 128^2 + c_2 \cdot 128 + c_3) \bmod p]$. Za p vzemimo kakšno praštevilo okoli 10^5, pa imamo že kar lepe možnosti, da se nam bodo vzorci lepo razpršili po tabeli. To bi prišlo še posebej prav, če bi hoteli delati s kakšnim večjim naborom znakov (kjer bi navadna tabela namesto 96^3 ali 128^3 elementov zahtevala pač a^3 elementov, če je a število vseh možnih znakov — pri npr. 500 ali 1000 znakih takšna tabela ne bi bila več mačji kašelj). Paziti moramo še na to, da ko potem zares primerjamo črke vzorca s črkami niza s, ne smemo začeti npr. pri $k = 3$, ampak pri $k = 0$, ker zdaj ne vemo več zagotovo, ali se vzorec res ujema z nizom s v prvih treh črkah (ker se lahko več trojic $c_1c_2c_3$ preslika v isti seznam v tabeli *prvi*).

```
program PogresaneOsebe;
const MaxN = 10000; MaxD = 20; MaxM = 10000000;
      Modulo = 100043; { Število seznamov, v katere bomo razdelili vzorce
                         (glede na prve tri znake). }
type SporociloT = packed array [1..MaxM] of char;
var
  S: ↑SporociloT; { Sporočilo, v katerem iščemo pojavitve vzorcev. }
  Ps: array [1..MaxN] of string; { Vzorci, katerih pojavitve iščemo. }
  n, m: integer; { n = število vzorcev; m = dolžina sporočila. }
  StPojavitev: integer; { Število vseh najdenih pojavitev vzorcev. }
  { Prvi[x] = prvi vzorec, čigar začetne tri črke imajo razprševalno kodo x. }
  Prvi: array [0..Modulo − 1] of integer;
  { Nasl[i] = naslednji vzorec, čigar začetne tri črke imajo isto razprševalno kodo }
  Nasl: array [1..MaxN] of integer; { kot začetne tri črke vzorca i. }
```

[5] Prvega v seznamu; drugače pa je to zaradi načina, kako naš gornji postopek sestavlja ta seznam, med vzorci $p^1, \ldots, p^n$ pravzaprav zadnji tak, ki se začne na črko c.

```
  { Kratki[c] = kolikokrat se c sam zase pojavi kot vzorec. }
  Kratki: array [Chr(0)..Chr(127)] of integer;
  { Kratki2[c, cc] = kolikokrat se pojavi vzorec c + cc (dolg 2 znaka). }
  Kratki2: array [Chr(0)..Chr(127), Chr(0)..Chr(127)] of integer;
  i, j, h, d, dd: integer; P: string; T: text; c, cc, ccc: char;
begin
  { Postavimo vse števce kratkih vzorcev na 0. }
  for c := Chr(0) to Chr(127) do begin
    Kratki[c] := 0;
    for cc := Chr(0) to Chr(127) do Kratki2[c, cc] := 0;
  end; {for c}
  { Seznami daljših vzorcev so na začetku prazni. }
  for h := 0 to Modulo - 1 do Prvi[h] := 0;
  { Preberimo vzorce iz vhodne datoteke. }
  Assign(T, 'osebe.in'); Reset(T); ReadLn(T, n);
  for i := 1 to n do begin
    ReadLn(T, Ps[i]); c := Ps[i, 1];
    { Kratke vzorce štejemo v tabelah Kratki in Kratki2. }
    if Length(Ps[i]) = 1 then Kratki[c] := Kratki[c] + 1
    else begin
      cc := Ps[i, 2];
      if Length(Ps[i]) = 2 then Kratki2[c, cc] := Kratki2[c, cc] + 1
      else begin { Daljše vzorce dodajmo vsakega v ustrezni seznam. }
        ccc := Ps[i, 3]; h := ((Ord(c) shl 14) or (Ord(cc) shl 7) or Ord(ccc)) mod Modulo;
        Nasl[i] := Prvi[h]; Prvi[h] := i;
      end; {if}
    end; {if}
  end; {for i}
  { Preberimo še sporočilo. }
  m := 0; New(S);
  while not Eoln(T) do begin m := m + 1; Read(T, S↑[m]) end;
  Close(T);
  { Poiščimo pojavitve vzorcev. }
  StPojavitev := 0;
  for j := 1 to m do begin
    c := S↑[j];
    { Preštejmo pojavitve kratkih vzorcev. }
    StPojavitev := StPojavitev + Kratki[c];
    if j < m then begin cc := S↑[j + 1]; StPojavitev := StPojavitev + Kratki2[c, cc] end;
    if j >= m - 1 then continue;
    { Preglejmo vzorce, ki se začnejo na S↑[j] + S↑[j + 1] + S↑[j + 2]. }
    ccc := S↑[j + 2]; h := ((Ord(c) shl 14) or (Ord(cc) shl 7) or Ord(ccc)) mod Modulo;
    i := Prvi[h];
    while i > 0 do begin
      P := Ps[i]; d := Length(P); i := Nasl[i];
      if j + d - 1 > m then continue;
      { Preverimo, če se vzorec P res pojavlja na tem mestu v sporočilu. }
      dd := 1;
      while dd <= d do
        if S↑[j + dd - 1] <> P[dd] then break
        else dd := dd + 1;
      if dd > d then StPojavitev := StPojavitev + 1;
    end; {while}
  end; {for j}
```

```
  { Izpišimo rezultat. }
  Assign(T, 'osebe.out'); Rewrite(T); WriteLn(T, StPojavitev); Close(T);
end. {PogresaneOsebe}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>
#define MaxN 10000000
#define MaxD 20
#define MaxM 10000
#define Modulo 100043
char *BeriVrstico(FILE *f, int maxDolz)
{
  char *p = (char *) malloc(maxDolz + 2), *r;
  fgets(p, maxDolz + 2, f);
  r = strchr(p, '\n'); if (r) *r = 0;
  return p;
}
int main()
{
  int m, n, h, i, j, k, *prvi, *nasl, *kratki, *kratki2, rezultat;
  char*s, *p[MaxM], *pp, c, c2, c3;
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("osebe.in", "rt");
  fscanf(f, "%d\n", &n);
  for (i = 0; i < n; i++) p[i] = BeriVrstico(f, MaxD);
  s = BeriVrstico(f, MaxN); m = strlen((char *) s);
  fclose(f);
  /* Pripravimo si sezname vzorcev.
     prvi[h] = prvi vzorec, ki se začne na tri črke s hash kodo h;
     nasl[i] = naslednji vzorec, čigar prve tri črke imajo isto hash kodo kot i-jeve.
     kratki in kratki2 štejeta vzorce dolžine 1 in 2. */
  prvi = (int *) malloc(Modulo * sizeof(int));
  nasl = (int *) malloc(n * sizeof(int));
  kratki = (int *) malloc(128 * sizeof(int));
  kratki2 = (int *) malloc(128 * 128 * sizeof(int));
  for (h = 0; h < Modulo; h++) prvi[h] = -1;
  for (h = 0; h < 128; h++) kratki[h] = 0;
  for (h = 0; h < 128 * 128; h++) kratki2[h] = 0;
  for (i = 0; i < n; i++)
  {
    c = p[i][0]; c2 = p[i][1];
    if (! c2) { kratki[c]++; continue; }
    c3 = p[i][2]; if (! c3) { kratki2[(int) c << 7 | c2]++; continue; }
    /* Izračunajmo hash kodo iz črk c, c2 in c3 in dodajmo i v seznam h. */
    h = ((int) c << 14 | (int) c2 << 7 | p[i][2]) % Modulo;
    nasl[i] = prvi[h]; prvi[h] = i;
  }
  /* Poiščimo pojavitve vzorcev. */
  for (j = 0, rezultat = 0; j < m; j++)
  {
    /* Najprej preštejmo pojavitve kratkih vzorcev (dolžine 1 in 2). */
    c = s[j]; rezultat += kratki[c];
    c2 = s[j + 1]; if (! c2) continue;
    rezultat += kratki2[(int) c << 7 | c2];
```

```
    c3 = s[j + 2]; if (! c3) continue;
    /* Tu so v nizu s črke c, c2 in c3; preglejmo seznam vzorcev,
       ki bi se utegnili začeti na to trojico črk. */
    h = ((int) c << 14 | (int) c2 << 7 | c3) % Modulo;
    for (i = prvi[h]; i >= 0; i = nasl[i]) {
      k = 0; pp = p[i]; /* Ali se tu pojavlja vzorec i? */
      while (s[j + k] == pp[k] && pp[k]) k++;
      if (! pp[k]) rezultat++; }
  }
  /* Izpišimo rezultat. */
  f = fopen("osebe.out", "wt");
  fprintf(f, "%d\n", rezultat);
  fclose(f);
  /* Pospravimo za sabo. */
  for (i = 0; i < n; i++) free(p[i]);
  free(s); free(prvi); free(nasl); free(kratki); free(kratki2);
  return 0;
}
```

Mimogrede, nobenega posebnega razloga ni, da bi morali za razvrščanje vzorcev v sezname uporabljati ravno prve tri črke. Lahko bi dovolili poljubne tri zaporedne črke vzorca, le da bi si morali potem tudi zapomniti njihov položaj znotraj vzorca (da bi potem pri primerjanju z znaki niza s vedeli, kje začeti). Lahko bi vzeli vedno zadnje tri črke, vendar se to dostikrat ne obnese, ker ima veliko besed isto končnico. Pametneje je na primer vedno vzeti tisto trojico zaporednih črk vzorca, ki se v besedilu pojavlja najredkeje — tako bo čim več vzorcev prišlo v take sezname, ki jih bo treba pregledovati le redkokdaj.

Rešitve z razpršeno tabelo in z bisekcijo. Malo drugačen pristop k rešitvi je ta, da za vse dovolj kratke podnize sporočila pogledamo, če so slučajno enaki kakšnemu od vzorcev. Pri naši nalogi si to lahko privoščimo, ker vemo, da so vzorci dolgi največ 20 znakov.

```
naj bo d dolžina najdaljšega vzorca;
for i := 1 to m do
  for k := 1 to min{d, m − i + 1} do begin
    t := s_i s_{i+1} ... s_{i+k−1};
    if je t enak kakšnemu od vzorcev p^1, ..., p^n then
      našli smo pojavitev vzorca dolžine k v nizu s na mestu i;
  end; (* for *)
```

Vprašanje je, kako za niz t preveriti, če je enak kakšnemu od vzorcev. Če bi imeli vse vzorce v seznamu in niz t primerjali z vsakim od njih, bi dobili podobno neučinkovit postopek, kot je bil tisti prvi naivni z začetka naše rešitve.

Bolje je, če shranimo vzorce v razpršeni tabeli; potem moramo za vsak t izračunati le še njegovo razpršilno kodo in ga nato primerjati s tistimi vzorci, ki imajo enako razpršilno kodo kot t (največkrat ne bo sploh nobenega takega). Da ne bomo imeli preveč dela z računanjem razpršilnih kod, se zgledujmo po Rabin-Karpovem algoritmu in uporabimo razpršilno fukcijo oblike

$$h(t_1 t_2 \ldots t_k) = (t_1 \cdot a^{k-1} + t_2 \cdot a^{k-2} + \ldots + t_{k-1} \cdot a + t_k) \bmod b$$

za neki primerni konstanti a in b. Lepo pri tej funkciji je to, da zaradi lepih lastnosti operacije mod (ostanek po deljenju) velja

$$h(t_1t_2\ldots t_kt_{k+1}) = (h(t_1t_2\ldots t_k)\cdot a + t_{k+1}) \bmod b.$$

Ko se torej kandidat za vzorec, t, podaljša za en znak, ni težko izračunati njegove nove razpršilne kode s pomočjo razpršilne kode, ki jo je imel, preden smo mu dodali novi znak.

Ponavadi se bo pri večini položajev i in dolžin k izkazalo, da takrat dobljeni niz t ni enak nobenemu od vzorcev $p^1, \ldots, p^n$. To pa pomeni, da bo naš tu opisani algoritem po nepotrebnem zapravil veliko časa za neuspešne poizvedbe po razpršeni tabeli. Preprosta izboljšava, s katero se lahko izognemo precejšnjemu delu teh neuspešnih poizvedb, je tale: v neki tabeli si za vsako možno trojico črk zapomnimo, ali se sploh kakšen vzorec začne na te tri črke. Gornji postopek potem spremenimo tako, da pri $k = 3$ najprej preveri, ali se kakšen vzorec začne na $s_is_{i+1}s_{i+2}$; če takega vzorca ni, lahko nad tem i-jem že kar takoj obupamo in se premaknemo na položaj $i + 1$. Primere, ko je $k = 1$ ali $k = 2$, lahko obravnavamo enako kot doslej, lahko pa si pri njih pomagamo z dvema tabelama, ki za vsako črko oz. par črk povesta, ali je kakšen vzorec dolžine 1 oz. 2 ravno enak tisti črki oz. paru črk.

Namesto z razpršeno tabelo si lahko pomagamo tudi z bisekcijo: vzorce hranimo v navadni tabeli, vendar jih na začetku uredimo v leksikografski vrstni red. Ko je treba preveriti, ali je nek podniz t sporočila s enak kakšnemu vzorcu, lahko to zdaj namesto s poizvedbo v razpršeni tabeli preverimo z bisekcijo po urejenem zaporedju vzorcev. Uporabimo lahko tudi enako izboljšavo kot v prejšnjem odstavku, torej da si za vsako trojico črk zapomnimo, ali se sploh kakšen vzorec začne na te tri črke.

Vendar pa je bisekcija načeloma počasnejša od poizvedbe po razpršeni tabeli: bisekcija po seznamu n vzorcev načeloma zahteva $O(\log n)$ primerjav vzorcev in iskanega niza t, poizvedba po razpršeni tabeli pa, če so se nam vzorci lepo razpršili po njej, le $O(1)$ takšnih primerjav. Lahko pa bisekcijo precej izboljšamo, če jo delamo „po črkah“ namesto po celih vzorcih:

naj bo d dolžina najdaljšega vzorca;
naj bo $p^1, \ldots, p^n$ leksikografsko urejen seznam vzorcev;
for $i := 1$ **to** m **do begin**
  $l := 1$; $r := n + 1$;
  for $k := 1$ **to** $\min\{d, m - i + 1\}$ **do begin**
    (* *Invarianta:* p^l *je prvi vzorec, ki je leksikografsko večji od ali enak podnizu* $s_i \ldots s_{i+k-2}$*; in* p^r *je prvi vzorec, ki je leksikografsko večji od* $s_i \ldots s_{i+k-2}$ *in se ne začne na ta niz. Če primernih vzorcev ni, sta lahko* l *in/ali* r *enaka* $n + 1$*.* *)
    naj bo l' prvi tak vzorec izmed $p^l, \ldots, p^{r-1}$,
      pri katerem je $p^{l'}_k \geq s_{i+k-1}$ (če takega ni, vzemimo $l' = r$);
    naj bo r' prvi tak vzorec izmed $p^l, \ldots, p^{r-1}$,
      pri katerem je $p^{r'}_k > s_{i+k-1}$ (če takega ni, vzemimo $r' = r$);
    if $l' = r'$ **then break**;
    $l := l'$; $r := r'$;
    if je p^l dolg natanko k znakov **then**

```
        našli smo pojavitev vzorca p^l v nizu s na mestu i;
    end; (* for k *)
  end; (* for i *)
```

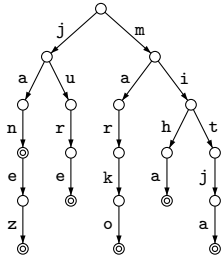
Na začetku vsake iteracije notranje zanke (po k) nam torej l in r povesta, kateri vzorci se začnejo na niz $s_i \dots s_{i+k-2}$ (to je niz dolžine $k-1$; pri $k=1$ si moramo ta pogoj razlagati tako, da nam l in r povesta, kateri vzorci se začnejo na prazen niz, to pa so seveda vsi vzorci, zato je prav, da je takrat $l=1$, $r=n+1$). Naloga trenutne iteracije notranje zanke je, da ugotovi, kateri od teh vzorcev se začnejo tudi na $s_i \dots s_{i+k-1}$, ne le na $s_i \dots s_{i+k-2}$. V ta namen mora opazovati k-ti znak teh vzorcev. Krajišči tako zožanega intervala bosta l' in r', ki ju poiščemo z bisekcijo znotraj intervala $\{l, \dots, r-1\}$. Lahko se izkaže, da primernega vzorca sploh ni (dobimo $l'=r'$), kar pomeni, da se nam tudi z večjimi k (pri trenutnem i) ni treba ukvarjati in lahko skočimo ven iz notranje zanke (in gremo na $i+1$). Če pa primerni vzorci obstajajo ($l'<r'$), moramo preveriti, če se kakšen med njimi ne le začne na $s_i \dots s_{i+k-1}$, ampak je temu podnizu kar enak. To je lahko le prvi vzorec z intervala $\{l', \dots, r'-1\}$, saj so vzorci urejeni leksikografsko; torej za $p^{l'}$ preverimo, če je dolg točno k znakov, in če je, povečajmo števec pojavitev.

Lepo pri tem postopku je, da se širina intervala $l..r$, po katerem moramo delati bisekcijo, hitro oža in temu primerno bisekcija zdaj ni več tako počasna. Čim pridemo v sporočilu do takega podniza, na katerega se ne začne noben vzorec, bosta l in r postala enaka in naša zanka po k ne bo po nepotrebnem pregledovala še daljših podnizov, ampak se bo takoj končala.

Temu postopku lahko še vedno dodamo tudi prej omenjene optimizacije, da kratke vzorce (dolžine 1 in 2) obravnavamo posebej in da imamo tabelo, ki za vsako trojico znakov pove, ali se kakšen vzorec začne na te tri znake. Pravzaprav je še bolje, če nam ta tabela za vsako začetno trojico znakov pove kar indeks prvega takega vzorca v seznamu, ki je leksikografsko večji ali enak od te trojice. Tako bomo lahko to tabelo uporabljali kot „kazalo“ v naše zaporedje vzorcev: pri vsakem i bomo lahko začeli s $k=3$ in z dvema pogledoma v kazalo bomo lahko takoj dobili primerni vrednosti l in r; tako bomo začeli s prijetno ozkim intervalom vzorcev, ne pa s celim intervalom $1..n$.

Rešitve z drevesom. Še bolj elegantno pa lahko nalogo rešimo takole. Za začetek zložimo vzorce v drevo: črke so na povezavah, za vsak vzorec p^j pa imamo od korena navzdol zaporedje povezav s črkami $p^j_1, \dots, p^j_{|p_j|}$. Če se več vzorcev začne na isto črko (ali nekaj istih črk), si delijo tudi skupno prvo povezavo (ali prvih nekaj povezav). V vsakem vozlišču je označeno, ali se pri njem končuje kakšna črka. Na sliki na str. 63 je primer takšnega drevesa za množico šestih vzorcev {`miha`, `jure`, `mitja`, `janez`, `marko`, `jan`}. Mimogrede, takemu drevesu se v angleščini reče *trie* (ker se uporablja za re*trie*val).

Kako si lahko s takšnim drevesom pomagamo pri iskanju vzorcev v nizu s? Recimo, da nas zanima, ali se pri znaku s_i začne kakšna pojavitev kakšnega vzorca. Očitno pridejo v poštev le tisti vzorci, ki se začnejo na s_i; torej se premaknimo iz korena drevesa v tisto vozlišče, v katerega vodi povezava z oznako s_i. Naslednja črka vzorca (če hočemo, da se pojavlja na tem položaju v nizu s) mora biti s_{i+1}, torej se spustimo v drevesu še en nivo nižje po povezavi s to črko; in tako naprej. Ko



pridemo do vozlišča, pri katerem se končuje nek vzorec, vemo, da se črke niza s od s_i do trenutne črke ujemajo s tem vzorcem, torej smo našli pojavitev tega vzorca. Prej ali slej pa pridemo v situacijo, ko spuščanja po drevesu ne moremo nadaljevati, ker manjka primerna povezava; to pomeni, da se noben vzorec ne začne na takšno zaporedje znakov iz niza s. Takrat s trenutnim položajem i končamo in lahko se posvetimo položaju $i + 1$. Na primer: če imamo drevo z gornje slike, niz s pa je `mirko`, bomo lahko (pri $i = 1$) šli iz korena po povezavi `m` in nato še po povezavi `i`, iz tistega vozlišča pa potem ni povezave z oznako `r`; takrat bi se ustavili in nadaljevali z $i = 2$ (in spet šli od korena drevesa naprej, vendar bi zdaj že takoj ugotovili, da iz korena ni nobene povezave z oznako `i`).

Ostane še čisto praktično vprašanje, kako v našem programu predstaviti vozlišča drevesa. Lahko ima vsako vozlišče tabelo otrok; ta tabela bi imela toliko celic, kolikor je črk, in vsaka celica bi vsebovala kazalec na pravega otroka (tistega, na katerega kaže povezava, označena s to črko) ali pa prazen kazalec (**nil**), če take povezave iz tega vozlišča ni. Težava je, da imamo pri naši nalogi načeloma 96 možnih znakov, torej bi bile te tabele velike in povečini prazne, celotno drevo pa bi zasedlo veliko pomnilnika.

Druga možnost je, da ima vsako vozlišče seznam otrok; to si lahko predstavljamo tudi tako, da ima vsako vozlišče poleg kazalca na seznam otrok (ki je v bistvu kazalec na prvega otroka tega vozlišča) tudi kazalec na svojega naslednjega brata (ti kazalci pa tvorijo seznam otrok očeta tega vozlišča). Tako ne tratimo pomnilnika po nepotrebnem, slabost te rešitve pa je, da so v višjih nivojih drevesa, predvsem v korenu, lahko ti seznami precej dolgi (ker vendarle obstaja veliko različnih črk, na katere se začne vsaj kakšen vzorec); zato bomo porabili precej časa pri preverjanjih, ali iz trenutnega vozlišča vodi kakšna povezava z določeno črko ali ne.

Tretja možnost je razpršena tabela, v kateri za vsak par ⟨številka vozlišča, črka⟩ hranimo številko otroka, v katerega kaže povezava s tisto črko iz tistega vozlišča. To je elegantno, zahteva pa nekaj dela z implementacijo razpršene tabele (če je nimamo ravno pri roki v kakšni standardni knjižnici).

Četrta možnost je hibrid prvih dveh: v korenu drevesa (in mogoče še v kakšnem nivoju pod njim) uporabimo tabelo otrok, nižje po drevesu (kjer ima večina vozlišč le malo otrok, ponavadi celo samo enega ali nobenega) pa sezname otrok. Zadeva je zelo podobna, kot če bi vzeli npr. zgornji program in vsak seznam vzorcev, na katerega kaže posamezni element tabele *prva*, organizirali v drevo. Nekaj takega bomo uporabili tudi v spodnjem programu: imeli bomo po eno drevo za vsako možno začetno črko vzorca, v vsakem takšnem drevesu pa bomo uporabljali sezname

otrok. Stvar je torej taka, kot če bi imeli eno samo veliko drevo, v katerem je koren predstavljen s tabelo, nižje ležeča vozlišča pa s seznami otrok.[6]

```
program PogresaneOsebeDrevo;
const MaxN = 10000; MaxD = 20; MaxM = 10000000;
type
  VozlisceP = ↑VozlisceT;
  VozlisceT = record
    c: char; { Znak na povezavi od očeta do tega vozlišča. }
    Koncno: integer; { Koliko vzorcev se konča pri tem vozlišču? }
    Otrok, Brat: VozlisceP; { Prvi otrok in naslednji brat tega vozlišča. }
  end; {VozlisceT}
  SporociloT = packed array [1..MaxM] of char;
var
  S: ↑SporociloT; { Sporočilo, v katerem iščemo pojavitve vzorcev. }
  { Koreni[c] = koren drevesa, v katerem so vsi vzorci, ki se začnejo na c. }
  Koreni: array [Chr(0)..Chr(127)] of VozlisceP;

  { Ta podprogram alocira in inicializira novo strukturo VozlisceT. }
  function NovoVozlisce(c: char; Otrok, Brat: VozlisceP): VozlisceP;
  var N: VozlisceP;
  begin
    New(N); N↑.c := c; N↑.Koncno := 0;
    N↑.Otrok := Otrok; N↑.Brat := Brat;
    NovoVozlisce := N;
  end; {NovoVozlisce}

  { Ta podprogram sprosti pomnilnik N-ja, njegovih bratov in njihovih potomcev. }
  procedure PospraviVozlisce(N: VozlisceP);
  begin
    if N↑.Otrok <> nil then PospraviVozlisce(N↑.Otrok);
    if N↑.Brat <> nil then PospraviVozlisce(N↑Brat);
    Dispose(N);
  end; {PospraviVozlisce}

  { Doda niz S[i..Length(S)] v drevo s korenom Oce. }
  procedure Dodaj(Oce: VozlisceP; S: string; i: integer);
  var c: char; Otrok: VozlisceP;
  begin
    if i > Length(S) then
      { Prišli smo že do konca niza, zato samo označimo, da se
        v vozlišču Oce konča nek vzorec. }
      begin Oce↑.Koncno := Oce↑.Koncno + 1; exit end;
    c := S[i];
    { Veja za naš niz se mora iz vozlišča Oce nadaljevati v otroka, na
      katerega kaže povezava z oznako c. Poglejmo, če tak otrok že obstaja. }
    Otrok := Oce↑.Otrok;
    while Otrok <> nil do
      if Otrok↑.c = c then break
      else Otrok := Otrok↑.Brat;
    { Če primernega otroka še ni, ga ustvarimo zdaj. }
    if Otrok = nil then begin
```

[6] Obstajajo še drugi, bolj zapleteni načini za predstavitev takega drevesa, ki omogočajo še hitrejše poizvedovanje po njem; glej npr. J.-I. Aoe, K. Morimoto, T. Sato, *An efficient implementation of trie structures*, Software — Practice and Experience, 22(9):695–731, September 1992.

```
        Otrok := NovoVozlisce(c, nil, Oce↑.Otrok);
        Oce↑.Otrok := Otrok;
      end; {if}
      { Nadaljujmo pri tem otroku in z naslednjim znakom niza. }
      Dodaj(Otrok, S, i + 1);
    end; {Dodaj}

var n, m, i, j, jj, StPojavitev: integer;
    T: text; P: string; c: char; Vozlisce: VozlisceP;
begin {PogresaneOsebeDrevo}
  { Imeli bomo po eno drevo (trie) za vsak možni začetni znak vzorca.
    Na začetku ima vsako drevo le koren. }
  for c := Chr(0) to Chr(127) do Koreni[c] := NovoVozlisce(c, nil, nil);
  { Preberimo vzorce iz vhodne datoteke in jih dodajmo v drevesa. }
  Assign(T, 'osebe.in'); Reset(T); ReadLn(T, n);
  for i := 1 to n do begin
    ReadLn(T, P);
    Dodaj(Koreni[P[1]], P, 2);
  end; {for i}
  { Preberimo še sporočilo. }
  m := 0; New(S);
  while not Eoln(T) do begin m := m + 1; Read(T, S↑[m]) end;
  Close(T);
  { Poiščimo pojavitve vzorcev. }
  StPojavitev := 0;
  for j := 1 to m do begin
    { Preštejmo pojavitve vzorcev, ki se začnejo pri znaku S↑[j].
      Če je to znak c, bomo ustrezne vzorce našli v drevesu, na katerega kaže Koreni[c]. }
    c := S↑[j]; Vozlisce := Koreni[c]; jj := j;
    while Vozlisce <> nil do begin
      { Spremenljivka Vozlisce trenutno kaže na vozlišče, ki ustreza
        vzorcu S↑[j] S↑[j + 1] ... S↑[jj]. Če tak vzorec res obstaja v
        naši vhodni datoteki, povečajmo število pojavitev. }
      StPojavitev := StPojavitev + Vozlisce↑.Koncno;
      { Premaknimo se na naslednji znak sporočila. }
      jj := jj + 1; if jj > m then break;
      c := S↑[jj];
      { Naslednji znak sporočila je c, torej se moramo premakniti iz trenutnega
        vozlišča v otroka, na katerega kaže povezava z oznako c (če tak otrok obstaja). }
      Vozlisce := Vozlisce↑.Otrok;
      while Vozlisce <> nil do
        if Vozlisce↑.c = c then break
        else Vozlisce := Vozlisce↑.Brat;
    end; {while}
  end; {for j}
  { Izpišimo rezultat. }
  Assign(T, 'osebe.out'); Rewrite(T); WriteLn(T, StPojavitev); Close(T);
  { Pospravimo za sabo. }
  for c := Chr(0) to Chr(127) do PospraviVozlisce(Koreni[c]);
end. {PogresaneOsebeDrevo}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>
#define MaxN 10000000
```

```
#define MaxD 20
#define MaxM 10000

typedef struct Vozlisce
{
  char c; int koncno;
  struct Vozlisce *otrok, *brat;
}
Vozlisce;

Vozlisce* NovoVozlisce(char c, Vozlisce *brat)
{
  Vozlisce *v = (Vozlisce *) malloc(sizeof(Vozlisce));
  v->c = c; v->koncno = 0; v->otrok = 0; v->brat = brat;
  return v;
}

void PospraviVozlisce(Vozlisce *v)
{
  if (v->otrok) PospraviVozlisce(v->otrok);
  if (v->brat) PospraviVozlisce(v->brat);
  free(v);
}

/* Doda v drevo s korenom „oce“ zaporedje vozlišč, ki ustreza nizu s. */
void Dodaj(Vozlisce *oce, const char *s)
{
  Vozlisce *otrok;
  /* Če smo že na koncu niza s, le povečajmo števec v korenu. */
  if (! *s) { oce->koncno++; return; }
  /* Sicer poiščimo primernega otroka za nadaljevanje veje. */
  for (otrok = oce->otrok; otrok; otrok = otrok->brat)
    if (otrok->c == *s) break;
  /* Če takega otroka še ni, ga dodajmo zdaj. */
  if (! otrok) oce->otrok = otrok = NovoVozlisce(*s, oce->otrok);
  Dodaj(otrok, ++s); /* Nadaljujmo pri otroku in pri naslednjem znaku niza. */
}

/* Prebere vrstico in odreže znak za konec vrstice.
   Shrani jo v niz s, če pa je ta 0, alocira nov niz. */
char *BeriVrstico(FILE *f, char *s, int maxDolz)
{
  char *p = s ? s : (char *) malloc(maxDolz + 2), *r;
  fgets(p, maxDolz + 2, f);
  r = strchr(p, '\n'); if (r) *r = 0;
  return p;
}

int main()
{
  int m, n, h, i, j, k, rezultat;
  Vozlisce *drevo[128], *v, *otrok;
  char *s, p[MaxD + 2], *S;
  FILE *f;

  /* Na začeku so drevesa prazna. */
  for (j = 0; j < 128; j++) drevo[j] = NovoVozlisce((char) j, 0);

  /* Preberimo vzorce in jih dodajmo v drevesa. */
  f = fopen("osebe.in", "rt"); fscanf(f, "%d\n", &n);
  for (i = 0; i < n; i++) {
```

```
    BeriVrstico(f, p, MaxD);
    Dodaj(drevo[p[0]], p + 1); }
  /* Preberimo niz, v katerem bomo iskali pojavive vzorcev. */
  s = BeriVrstico(f, 0, MaxN); m = strlen(s);
  fclose(f);
  /* Poiščimo pojavitve vzorcev. */
  for (j = 0, rezultat = 0; j < m; j++)
  {
    S = s + j; v = drevo[*S++];
    /* Pojdimo dol po drevesu in sledimo povezavam, ki ustrezajo črkam sporočila.
       Ustavimo se, ko pridemo do konca niza ali pa ko zmanjka primernih povezav. */
    while (v) {
      rezultat += v->koncno; /* Odkrili smo pojavitev nekega vzorca. */
      if (! *S) break; /* Prišli smo do konca sporočila. */
      /* Poiščimo otroka, v katerega kaže povezava, označena
         z naslednjo črko sporočila. */
      for (otrok = v->otrok; otrok; otrok = otrok->brat)
        if (otrok->c == *S) break;
      v = otrok; S++; }
  }
  /* Izpišimo rezultat. */
  f = fopen("osebe.out", "wt"); fprintf(f, "%d\n", rezultat); fclose(f);
  /* Pospravimo za sabo. */
  for (j = 0; j < 128; j++) PospraviVozlisce(drevo[j]);
  free(s); return 0;
}
```

Lepo pri tej rešitvi je, da nam zagotavlja, da bo šla notranja zanka pri vsakem i največ tako daleč, kolikor je globoko drevo; to pa je največ toliko, kolikor je dolg najdaljši vzorec. To je kot nalašč za našo nalogo, kjer je vzorcev sicer veliko, vendar so kratki.[7]

Primerjava hitrosti različnih rešitev. Za konec si oglejmo še primerjavo časov izvajanja tu opisanih rešitev na naših desetih testnih primerih. Primeri so podrobneje opisani v tabeli na str. 68; sporočilo (niz s) je bilo pri vseh testnih primerih neko besedilo v angleščini, pri čemer pa smo vse velike črke spremenili v male. Za

[7] Tu opisano rešitev z drevesom se da še izboljšati. Naš postopek, ko ne najde več primerne povezave v drevesu, poveča i na $i+1$ in začne spet pri korenu drevesa. V resnici pa ne bi bilo treba začeti pri korenu drevesa. Vsako vozlišče drevesa predstavlja nek niz w, ki ga dobimo, če staknemo črke na povezavah do tega vozlišča. Za vsak tak w lahko ob gradnji drevesa poiščemo najdaljšo tako končnico niza w, ki tudi sama predstavlja neko vozlišče drevesa (recimo tej končnici x); v vozlišču w si nato zapomnimo kazalec na vozlišče x. Če se nam potem med preiskovanjem niza zatakne pri vozlišču w, se premaknemo v x (i pa povečamo za $|w| - |x|$). Izkaže se, da je časovna zahtevnost te rešitve le še $O(m)$, ne več $O(mq)$ kot pri naši zgoraj opisani rešitvi (če je m dolžina niza s, q pa je dolžina najdaljšega vzorca). To izboljšavo sta opisala A. Aho in M. Corasick leta 1975 (*Efficient string matching: an aid to bibliographic search*, CACM, 18(6):333–340, June 1975); predstavljamo si jo lahko kot razširitev znanega Knuth-Morris-Prattovega algoritma (za iskanje enega vzorca) na problem iskanja več vzorcev. Lahko pa namesto Knuth-Morris-Pratta za izhodišče vzamemo Boyer-Mooreov algoritem in na problem iskanja več vzorcev posplošimo tega. Takšen algoritem sta opisala S. Wu in U. Manber leta 1994 (*A fast algorithm for multi-pattern searching*). Lep pregled in eksperimentalna primerjava različnih algoritmov za iskanje več vzorcev pa je L. Salmela, J. Tarhio, J. Kytöjoki, *Multipattern string matching with q-grams*, J. of Experimental Algorithmics, 11:1.1, 2006.

Testni primer	Sporočilo	Dolžina (v tisočih znakov)	Št. vzorcev
1	*Alica v čudežni deželi*	142	100
2–3	*Hamlet*	176	1000
4–5	*Hamlet*	176	4000
6	Edward Gibbon, *Zgodovina zatona in propada Rimskega cesarstva* (različno dolgi odlomki)	1990	10000
7		1992	10000
8		4979	10000
9		7967	10000
10		9470	10000

Opis testnih primerov pri 3. nalogi za 3. skupino.

Algoritem	Čas izvajanja [s]	Št. točk
naivna rešitev:		
— zunanja zanka po položaju v s, notranja po vseh vzorcih	1043	30
— zunanja zanka po vzorcih, notranja po položaju v s	937	
seznami vzorcev:		
— za vsako začetno črko	96,1	50
— za vsak začetni par črk	13,5	70
— za vsako začetno trojico črk	2,9	100
— za vsako končno trojico črk	2,5	
— za najredkejšo trojico črk	1,5	
razpršena tabela	6,5	
— s tabelo, ali se kak vzorec začne na neko trojico črk	2,6	
bisekcija		
— po celih vzorcih	58,6	
— po črkah, brez kazala	11,9	
— s kazalom za vsako začetno črko	5,9	
— s kazalom za vsak par začetnih črk	2,7	
— s kazalom za vsako trojico začetnih črk	1,6	
drevo:		
— koren ima tabelo otrok, drugod so seznami otrok	2,6	100
— koren in otroci imajo tabele otrok, drugod so seznami	1,6	
— podatki o povezavah so v razpršeni tabeli	2,3	
— razpršena tabela + Aho-Corasick	1,0	

Primerjava časa izvajanja različnih rešitev 3. naloge za 3. skupino.

Algoritme smo implementirali v C++ in jih prevedli z MS Visual Studiom 2005 z vsemi optimizacijami. Za vsak algoritem je prikazan skupni porabljeni čas CPU pri izvajanju tega algoritma na vseh desetih testnih primerih. V to ni vštet čas, porabljen za branje vhodnih datotek in zapisovanje izhodnih datotek. (Računalnik, na katerem so bili izmerjeni ti časi, je hitrejši od tistega, ki se je uporabljal na tekmovanju kot ocenjevalni strežnik.)

Zadnji stolpec kaže število točk, ki bi jih na tekmovanju dosegla malo manj učinkovita pascalska implementacija nekaterih tu prikazanih algoritmov. (Kjer je točk manj kot 100, je to posledica tega, da program pri nekaterih testnih primerih prekorači časovno omejitev 10 sekund na ocenjevalnem računalniku.)

---

vzorce smo vzeli pri vsakem testnem primeru nekaj naključno izbranih besed iz sporočila. Gornja tabela prikazuje skupni čas izvajanja raznih tu opisanih rešitev na vseh desetih testnih primerih.

4. Lemingi

Pomagajmo si z dejstvom, da lahko lemingi padajo le navzdol. V neki tabeli (v spodnjem programu je to Dosegljiva) si bomo za vsako črto označili, ali jo leming lahko doseže (in ostane živ) ali ne. Na začetku razglasimo za dosegljivo le črto 1, na kateri leming začne svojo pot. Potem pregledujmo črte od višjih proti nižjim; če je neka črta dosegljiva, poglejmo, kje bi pristali, če bi padli z nje (posebej za padec z levega roba in za padec z desnega roba); in če nek padec ni preglobok, označimo za dosegljivo tudi tisto črto, na kateri bi leming ob tem padcu pristal.

Koristno je tudi tla obravnavati kot eno od črt: je na višini 0 in dovolj široka, da je pri nobenem padcu ne moremo zgrešiti (spodnji program jo shrani kar v celici Crte[0]).

Črte bi lahko uredili po višini s katerimkoli od znanih postopkov za urejanje; ker pa črt ni veliko, uporablja spodnji program še preprostejši način: vsakič pregleda celo tabelo in poišče najvišjo črto v njej; ko tisto črto obdela, postavi njeno višino na -1, tako da bo v bodoče vedel, da se z njo ne sme več ukvarjati. (Tudi pri razmišljanju o padcih mu ne bo več hodila v napoto, saj zdaj pravzaprav leži pod tlemi.)

Ker črt ni veliko, nam tudi pri ugotavljanju, kje pristane leming ob padcu, ni treba kompliciratí: pregledamo lahko vse črte in pri vsaki preverimo, če pokriva primerno x-koordinato in če je nižja od črte, s katere leming pada. Med črtami, ki ustrezajo tem pogojem, si zapomnimo najvišjo — ta je potem tista, na kateri bi naš leming pri takem padcu zares pristal.

```
program Lemingi;
const MaxN = 1000; MaxKoord = 1000000;
type CrtaT = record a, b, h: integer end;
var n, d, i, j, k, c, Konec, x, Rezultat: integer; T: text;
    Crte: array [0..MaxN] of CrtaT;
    Dosegljiva: array [0..MaxN] of boolean;
begin
    { Preberimo vhodne podatke. }
    Assign(T, 'lemingi.in'); Reset(T); ReadLn(T, n, d);
    with Crte[0] do begin a := -1; b := MaxKoord + 1; h := 0 end;
    for i := 1 to n do with Crte[i] do ReadLn(T, a, b, h);
    Close(T);
    { Za začetek vemo le za črto 1, da je dosegljiva. }
    for i := 0 to n do Dosegljiva[i] := false;
    Dosegljiva[1] := true;
    { Pregledujmo črte po padajoči višini. }
    for i := 1 to n do begin
      k := 1; for j := 2 to n do if Crte[j].h > Crte[k].h then k := j;
      { k je najvišja še neobdelana črta. }
      if Dosegljiva[k] then begin
        Rezultat := Crte[k].h; { Na koncu bo v Rezultat najnižja dosegljiva črta. }
        { Poglejmo, kam se da pasti s črte k. Posebej bomo gledali
          padec z levega konca in padec z desnega konca. }
        for Konec := 1 to 2 do begin
          if Konec = 1 then x := Crte[k].a else x := Crte[k].b;
          c := 0; for j := 1 to n do
            if (Crte[j].h < Crte[k].h) and (Crte[j].h >= Crte[c].h)
            and (Crte[j].a < x) and (x < Crte[j].b) then c := j;
```

```
          { Padli bi na črto c. Ali padec preživimo? }
          if Crte[c].h >= Crte[k].h − d then Dosegljiva[c] := true;
        end; {for Konec}
      end; {if Dosegljiva[k]}
      Crte[k].h := −1; { Označimo si, da smo jo že obdelali. }
    end; {for i}
    if Dosegljiva[0] then Rezultat := 0;
    { Izpišimo rezultat. }
    Assign(T, 'lemingi.out'); Rewrite(T); WriteLn(T, Rezultat); Close(T);
end. {Lemingi}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>
#include <stdbool.h>
#define MaxN 1000
#define MaxKoord 1000000
int main()
{
  struct { int a, b, h; bool dosegljiva; } crte[MaxN + 1];
  int n, d, i, j, k, x, c, konec, rezultat;
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("lemingi.in", "rt");
  fscanf(f, "%d %d", &n, &d);
  crte[0].a = −1; crte[0].b = MaxKoord + 1; crte[0].h = 0;
  for (i = 1; i <= n; i++) fscanf(f, "%d %d %d", &crte[i].a, &crte[i].b, &crte[i].h);
  fclose(f);
  /* Označimo za dosegljivo le prvo črto (na kateri leming začne svojo pot). */
  for (i = 0; i <= n; i++) crte[i].dosegljiva = (i == 1);
  /* Pojdimo od najvišje črte proti najnižji. */
  for (i = 0; i < n; i++)
  {
    /* Poiščimo najvišjo še neobdelano črto. */
    for (k = 1, j = 2; j <= n; j++) if (crte[j].h > crte[k].h) k = j;
    if (crte[k].dosegljiva) /* Dosegljiva je — poglejmo, kam se pade z nje. */
    {
      rezultat = crte[k].h;
      /* Posebej glejmo padec z levega in padec z desnega konca. */
      for (konec = 0; konec < 2; konec++)
      {
        x = konec ? crte[k].a : crte[k].b; /* Kam pademo s tega konca črte k? */
        for (c = 0, j = 1; j <= n; j++) if (crte[j].a < x && x < crte[j].b)
          if (crte[j].h < crte[k].h && crte[j].h > crte[c].h) c = j;
        /* Pademo na c; označimo jo za dosegljivo, če padec preživimo. */
        if (crte[c].h >= crte[k].h − d) crte[c].dosegljiva = true;
      }
    }
    crte[k].h = −1; /* Označimo, da smo jo že obdelali. */
  }
  if (crte[0].dosegljiva) rezultat = crte[0].h; /* Dosegljiva so tudi tla. */
  /* Izpišimo rezultat. */
  f = fopen("lemingi.out", "wt");
  fprintf(f, "%d\n", rezultat); fclose(f); return 0;
}
```

Časovna zahtevnost tega postopka je $O(n^2)$, kar je za tako majhno število črt povsem ustrezno. Kot zanimivost si oglejmo še, kako bi lahko problem z malo bolj zapletenim postopkom rešili tudi precej hitreje — v času $O(n \log n)$. Za začetek uredimo x-koordinate levih in desnih robov vseh črt; pobrišimo morebitne duplikate in dobimo neko zaporedje $x_0 \leq x_1 \leq \ldots \leq x_k$, pri čemer je $k < 2n$. Za vsako črto si zapomnimo, kje v tem zaporedju sta koordinati njenega levega in desnega roba: pri črti i naj bo recimo levi rob na x_{u_i}, desni pa na x_{v_i}.

Zdaj pregledujmo črte po naraščajoči y-koordinati; pri vsaki črti se vprašamo, na kateri od dosedanjih (nižje ležečih) črt bi leming pristal, če bi padel z njenega levega oz. desnega roba. V ta namen si pripravimo nekaj tabel $T^0, T^1, \ldots$, pri čemer ima T^d natanko $\lfloor k/2^d \rfloor$ celic in i-ta med njimi predstavlja interval od $x_{i \cdot 2^d}$ do $x_{(i+1) \cdot 2^d}$. (Takih tabel potrebujemo do $d = \lfloor \log_2 k \rfloor$; od tam naprej bi bile tako ali tako prazne.) V celici $T^d[i]$ bomo hranili višino najvišje take črte (izmed doslej pregledanih), ki v celoti pokriva interval te celice, ne pokriva pa v celoti intervala njej nadrejene celice $T^{d+1}[i \operatorname{div} 2]$. Ko dodamo novo (višjo) črto, je treba popraviti največ dve celici v vsaki tabeli: na primer, če imamo črto, ki gre od x_{10} do x_{35}, moramo popraviti le celice T^0_{34} (zaradi intervala $[x_{34}, x_{35}]$), T^1_5 (zaradi intervala $[x_{10}, x_{12}]$), T^1_{16} (zaradi intervala $[x_{32}, x_{34}]$), T^2_3 (zaradi intervala $[x_{12}, x_{16}]$) in T^4_2 (zaradi intervala $[x_{16}, x_{32}]$). Dodajanje nove črte v naš sistem tabel nam bo torej vzelo le $O(\log n)$ časa.

Ko nas potem za nek padec z neke znane x-koordinate zanima, na kateri črti bi ob takem padcu pristali, je treba le pogledati v vsaki tabeli tisto celico, katere interval pokriva našo x-koordinato; v vsaki od teh celic dobimo neko višino in zapomnimo si najvišjo med njimi. Tako nam tudi poizvedba v tabelo vzame $O(\log n)$ časa. Ker moramo dodajanje in poizvedbo v tabelo izvesti za vsako črto po enkrat, nam ta del postopka vzame $O(n \log n)$ časa; toliko pa ga gre tudi za urejanje x- in y-koordinat črt na začetku postopka.

5. Štoparji

Problemov razporejanja (scheduling), med katere sodi tudi naša naloga, se pogosto lotevamo s požrešnimi algoritmi: potnike pregledujemo v nekem izbranem vrstnem redu in se pri vsakem odločimo, ali bi ga sprejeli ali ne; ko pa ga enkrat sprejmemo, ga obdržimo do konca našega postopka in se kasneje nikoli ne vrnemo k njemu in ne razmišljamo, ali ga mogoče vendarle raje ne bi sprejeli. Pri naši nalogi je smiselno predvsem preverjati, ali bi se s sprejemom novega potnika kdaj zgodilo, da bi imeli v avtomobilu več kot k potnikov hkrati; če se to zgodi, ga ne smemo sprejeti, drugače pa ga načeloma lahko. Tako pridemo do naslednjega postopka:

```
1  S := {};
2  pregleduj potnike v nekem izbranem vrstnem redu:
3     naj bo i trenutni potnik;
4     če je pri trenutni množici potnikov S avtomobil kakšen del intervala [s_i, e_i)
      že polno zaseden (s k potniki), nadaljuj pri naslednjem potniku;
5     sicer: S := S ∪ {i}; (* dodaj potnika i v množico S *)
```

Na koncu postopka je v množici S neka skupina potnikov, ki jih lahko prepeljemo, ne da bi jih bilo v avtomobilu kdaj več kot k naenkrat. Velikost množice S pa je v splošnem odvisna od tega, v kakšnem vrstnem redu smo potnike obravnavali.

Nekaj vrstnih redov, ki nam lahko hitro pridejo na misel, je na primer: potnike lahko pregledujemo v naraščajočem vrstnem redu trajanja vožnje ($e_i - s_i$); po naraščajočem času vstopa, s_i; po naraščajočem času izstopa, e_i; lahko pa po padajočem trajanju vožnje, padajočem času vstopa ali padajočem času izstopa; ali pa potnike pregledujemo celo v naključnem vrstnem redu. Lahko tudi preizkusimo več tu navedenih vrstnih redov (še posebej več naključnih vrstnih redov), pri vsakem najbrž dobimo malo drugačno množico S, na koncu pa vrnemo največjo med njimi.

Če bomo to res naredili, bomo sčasoma opazili, da najboljše rešitve (torej največje množice S) vedno dobimo takrat, ko pregledujemo potnike po naraščajočem vrstnem redu časa izstopa (e_i). Zato lahko posumimo, da ta različica požrešnega algoritma mogoče vrača kar najboljše možne rešitve sploh,[8] torej da pri vsakem testnem primeru vrne največjo možno množico S, pri kateri se v avtomobilu nikoli ne pelje več kot k potnikov hkrati. Kasneje se bomo prepričali, da je to res, še prej pa si oglejmo primer implementacije tega algoritma.

```
program Stoparji;
const MaxM = 1000; MaxN = 100; MaxK = 100;
var F: text;
    i, t, u, n, m, k: integer;
    { Casi vstopa in izstopa vseh potnikov. }
    Si, Ei: array [1..MaxM] of integer;
    { Zasedenost avtomobila: od mesta t do t + 1 se pelje Z[i] potnikov. }
    Z: array [1..MaxN - 1] of integer;
    { Prepeljani potniki. }
    S: array [1..MaxM] of boolean; nPrepeljanih: integer;
begin
  { Preberimo vhodne podatke. }
  Assign(F, 'stoparji.in'); Reset(F); ReadLn(F, n, k, m);
  for i := 1 to m do ReadLn(F, Si[i], Ei[i]);
  Close(F);
  for t := 1 to n - 1 do Z[t] := 0;
  for i := 1 to m do S[t] := false;
  { Preglejmo vse potnike. }
  nPrepeljanih := 0;
  for t := 1 to n do
    { Preglejmo vse potnike, ki izstopijo ob času t. }
    for i := 1 to m do if Ei[i] = t then begin
      { Ali lahko prepeljemo še tega potnika? }
      S[i] := true;
```

[8] Do tega suma lahko pridemo tudi takole. Za nekatere od tu predlaganih vrstnih redov obravnavanja potnikov se da precej hitro sestaviti tudi zelo neugodne testne primere, pri katerih bi vrnil požrešni algoritem zelo slabo rešitev; za vrstni red po naraščajočem času izstopa pa nam ne bo uspelo sestaviti takega testnega primera, ki ga požrešni algoritem ne bi rešil optimalno.

Drugače je, če gledamo potnike na primer po naraščajočem času vstopa ali pa po padajočem trajanju vožnje. Takrat lahko dobimo s požrešnim algoritmom zelo slabe rešitve, npr. če imamo k potnikov s $s_i = 1$ in $e_i = n$ — vse jih bomo pobrali takoj na začetku, zapolnili z njimi ves prostor v avtomobilu in jih odložili šele na koncu, četudi bi se dalo vmes mogoče prepeljati precej več drugih potnikov s krajšimi vožnjami.

Neugodne primere se da sestaviti tudi za požrešni postopek, ki pregleduje potnike po naraščajočem trajanju vožnje; imamo lahko npr. k potnikov s $s_i = t - 1$, $e_i = t + 1$, nato k potnikov s $s_i = t - 3$, $e_i = t$ in še k potnikov s $s_i = t$, $e_i = t + 3$. (Ta vzorec se lahko ponavlja pri $t = 4, 10, 16, \ldots$) Požrešni algoritem bi se zakopal v prvih k (ker imajo krajšo vožnjo, peljejo se le dve mesti namesto treh) in z njimi takoj zapolnil avto; če pa bi se tem potnikom odpovedal, bi lahko prepeljal vseh ostalih $2k$ potnikov.

```
        for u := Si[i] to Ei[i] − 1 do
          if Z[u] = k then begin S[i] := false; break end;
        if not S[i] then continue;
        { Lahko ga prepeljemo; popravimo tabelo S. }
        for u := Si[i] to Ei[i] − 1 do Z[u] := Z[u] + 1;
        nPrepeljanih := nPrepeljanih + 1;
      end; {for i, if}
  { Izpišimo rezultate. }
  Assign(F, 'stoparji.out'); Rewrite(F); WriteLn(F, nPrepeljanih);
  for i := 1 to m do if S[i] then WriteLn(F, i);
  Close(F);
end. {Stoparji}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>
#include <stdbool.h>

#define MaxM 1000
#define MaxN 100
#define MaxK 100

int main()
{
  int n, m, k, Si[MaxM], Ei[MaxM], i, t, u, nPrepeljanih;
  int Z[MaxK]; /* Zasedenost avtomobila: od mesta t do t + 1 se pelje Z[t] štoparjev. */
  bool S[MaxM]; /* S[i] pove, ali bomo štoparja i prepeljali ali ne. */

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("stoparji.in", "rt");
  fscanf(f, "%d %d %d", &n, &k, &m);
  for (i = 0; i < m; i++) fscanf(f, "%d %d", &Si[i], &Ei[i]);
  fclose(f);

  /* Na začetku naj bo množica prepeljanih štoparjev prazna. */
  for (t = 1; t < n; t++) Z[t] = 0;
  for (i = 0; i < m; i++) S[i] = false;
  nPrepeljanih = 0;

  /* Pregledujmo štoparje po naraščajočem času izstopa. */
  for (t = 1; t <= n; t++) for (i = 0; i < m; i++) if (Ei[i] == t)
  {
    /* Ali lahko sprejmemo tega štoparja? */
    S[i] = true;
    for (u = Si[i]; u < Ei[i]; u++) if (Z[u] == k) { S[i] = false; break; }
    if (! S[i]) continue; /* Ne moremo ga sprejeti. */
    for (u = Si[i]; u < Ei[i]; u++) Z[u]++;
    nPrepeljanih++;
  }

  /* Izpišimo rezultat. */
  f = fopen("stoparji.out", "wt"); fprintf(f, "%d\n", nPrepeljanih);
  for (i = 0; i < m; i++) if (S[i]) fprintf(f, "%d\n", i + 1);
  fclose(f); return 0;
}
```

Gornji program deluje za potrebe naše naloge čisto dovolj hitro, dalo pa bi se ga na razne načine še izboljšati. Trenutno je njegova časovna zahtevnost $O(mn)$: tolikokrat se izvede na primer stavek „**if** Ei[i] = t“; izpolnjen pa je ta pogoj v m primerih, namreč

za vsakega potnika po enkrat; takrat se potem izvedeta med drugim tudi notranji zanki po u, ki vzameta vsakič še $O(n)$ časa.

Urejanja potnikov po času izstopa bi se lahko lotili tudi tako, da bi si z enim samim prehodom čez vse potnike (lahko že ob branju vhodne datoteke) za vsak možen čas izstopa pripravili seznam potnikov, ki izstopajo takrat; to bi nam vzelo le $O(n+m)$ časa namesto $O(nm)$. Lahko pa bi seveda uporabili katerega od običajnih algoritmov za urejanje, npr. quicksort, kar bi nam vzelo $O(m \log m)$ časa; to bi bilo koristno, če je n velik v primerjavi z m-jem (ali pa če imamo namesto indeksov mest res prave čase, ki niso nujno cela števila).

Notranje zanke po u se lahko rešimo na primer z drevesom segmentov.[9] Namesto ene same tabele Z z $n-1$ celicami si mislimo več tabel, $Z^0, Z^1, \ldots$, pri čemer ima tabela Z^d le $\lfloor (n-1)/2^d \rfloor$ celic (oštevilčenih od 1 naprej). Celica $Z^d[t]$ predstavlja pot od mesta $1+(t-1)\cdot 2^d$ do $1+t\cdot 2^d$. O drevesu govorimo zato, ker so ti intervali organizirani hierarhično: interval celice $Z^d[t]$ je sestavljen ravno iz intervalov celic $Z^{d-1}[2t-1]$ in $Z^{d-1}[2t]$. V vsaki celici naj bosta dve števili: eno (recimo z_t^d) je število takih potnikov, ki se peljejo z nami po celem intervalu, ki ga predstavlja ta celica, ne pa po celem intervalu katere od hierarhično nadrejenih celic v drevesu; drugo (recimo $\hat{z}_t^d$) pa je $z_t^d + \max\{\hat{z}_{2t-1}^{d-1}, \hat{z}_{2t}^{d-1}\}$, kar je ravno največje število potnikov, ki se kdaj znotraj intervala te celice znajdejo skupaj v našem avtomobilu (če ne štejemo potnikov, ki se peljejo z nami po celem intervalu katere od hierarhično nadrejenih celic). Takšno drevo tabel ima $O(\log n)$ nivojev in ko preverjamo, kakšno je največje hkratno število potnikov na nekem intervalu, moramo pregledati le konstantno mnogo celic na vsakem nivoju; enako je tudi s popravljanjem tabele, ko dodamo v množico S novega potnika. Zato nam zdaj ukvarjanje z vsakim potnikom vzame le $O(\log n)$ časa, ne več $O(n)$ kot pri zgornjem programu.

Če je n večji od m, lahko vse čase s_i in e_i tudi uredimo naraščajoče, odstranimo duplikate (ostane največ $2m$ časov) in si nato za vsakega potnika zapomnimo položaj njegovih časov s_i in e_i v tako dobljenem vrstnem redu; učinek je tak, kot da smo dosedanji interval časov $1..n$ skrčili na interval $1..2m$, ne glede na to, kako velik je bil prvotno n. Takšno urejanje časov nam vzame $O(m \log m)$ časa. Če upoštevamo vse dosedanje izboljšave, vidimo, da lahko dosežemo časovno zahtevnost $O(m \log \min\{n, m\})$.

Dokaz optimalnosti. Prepričajmo se, da vrača naš požrešni algoritem, ki pregleduje potnike v naraščajočem vrstnem redu časa izstopa, optimalne rešitve.[10] Recimo, da ni tako; da torej pri nekem testnem primeru opisani algoritem vrne množico S, optimalna rešitev pa je neka večja množica $S^\star$, torej $|S^\star| > |S|$. Oštevilčimo pri tem testnem primeru potnike v naraščajočem vrstnem redu izstopnih časov, torej tako, da bo $e_1 \le e_2 \le \ldots \le e_m$.

Če primerjamo množici S in $S^\star$, se mogoče pri nekaterih potnikih ujemata (torej take potnike vsebujeta obe ali pa nobena); ker pa je $S^\star$ večja od S, gotovo nastopi pri vsaj enem potniku r tudi neujemanje, namreč takšno, da $S^\star$ tega potnika vsebuje, S

[9] Prav tak prijem smo uporabili že na koncu rešitve 4. naloge za 3. skupino (str. 71). Glej tudi rešitev naloge 1998.2.3, str. 359–61 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004*.

[10] Bralec, ki mu je ta dokaz preveč kosmat, ga lahko brez posebne škode preskoči. V nadaljevanju si bomo ogledali še en algoritem, ki daje enako dobre rešitve, pa še lepši dokaz optimalnosti ima.

pa ne. Če je takih potnikov več, vzemimo za r tistega z najmanjšo številko. Seveda je v splošnem mogoče, da pred r-jem (torej med potniki $1, 2, \ldots, r-1$) pride tudi do kakšnega neujemanja druge vrste (torej da se kakšen od teh potnikov pojavlja v S, ne pa v $S^\star$).

Če je možnih več enako velikih optimalnih rešitev, vzemimo za $S^\star$ tisto, ki ima največji presek s S; če je tudi takih več, vzemimo med njimi tisto, ki ima pred prvim neujemanjem tipa „potnik je v $S^\star$, ne pa v S“ najmanjše število neujemanj tipa „potnik je v S, ne pa v $S^\star$“.

Zdaj smo prišli do neujemanja pri potniku r, ki leži v $S^\star$, ne pa v S. Potem je množica $(S - S^\star) \cap \{1, \ldots, r-1\}$ množica vseh potnikov, za katere smo pred r-jem opazili neujemanja (in so ti potniki le v S, ne pa v $S^\star$); recimo, da je teh potnikov t, in označimo jih s $q_1, \ldots, q_t$. Vsa ta števila so $< r$ in zato so vsi $e_{q_i} \le e_r$. Med temi potniki označimo s q tistega, ki ima največji začetni čas.

Ali je mogoče, da je $s_q \ge s_r$? To bi (ker smo v prejšnjem odstavku videli tudi $e_q \le e_r$) pomenilo, da je q-jev časovni interval podmnožica r-jevega; če bi potem v $S^\star$ zamenjali r-ja s q-jem, bi ostala množica $S^\star$ veljavna in enako velika kot prej, njen presek s S pa bi se povečal, kar je protislovje, ker smo rekli, da bomo vzeli tako $S^\star$, pri kateri je presek največji možni.

Torej je $s_q < s_r$. Še vedno pa mora veljati (ker bi drugače prišli v protislovje), da bi množica $S^\star$ postala neveljavna, če bi v njej potnika r zamenjali s q-jem. Pri zamenjavi r-ja s q-jem se zgodi naslednje: v času $(s_q, \min\{e_q, s_r\})$ je v avtomobilu en potnik več kot prej, v času (e_q, e_r) pa en potnik manj kot prej. Tole slednje ne more povzročiti, da množica postane neveljavna, če je bila prej veljavna; neveljavnost mora torej povzročiti dejstvo, da si v vsaj nekem delu intervala $(s_q, \min\{e_q, s_r\})$ ne moremo privoščiti v avtomobilu enega potnika več, kot smo ga imeli v prvotni $S^\star$. Nekje na tem intervalu je torej obdobje, ko je avtomobil že v množici $S^\star - \{r\}$ polno zaseden. Glede potnikov od 1 do $r-1$ ne vsebuje ta množica nič takega, česar ne vsebuje tudi S, poleg tega pa S vsebuje še potnika q, ki pokriva celoten interval $(s_q, \min\{e_q, s_r\})$ in ga množica $S^\star - \{r\}$ ne vsebuje. Torej samo potniki s številkami od 1 do $r-1$ vsekakor ne zmorejo sami popolnoma zapolniti avtomobila v nobenem delu intervala $(s_q, \min\{e_q, s_r\})$. Ravno tako seveda k zasedenosti tega intervala ne more prispevati potnik r (ker za kaj takega vstopi prepozno, namreč šele ob času s_r); torej mora množica $S^\star - \{r\}$ vsebovati tudi nekega potnika $u > r$, ki pokriva vsaj del intervala $(s_q, \min\{e_q, s_r\})$. Zato pa mora seveda tega potnika vsebovati tudi $S^\star$.

Ali se lahko zgodi, da je $s_u \le s_q$? Ker je $u > r$, je tudi $u > q$ in zato $e_u \ge e_q$; skupaj s $s_u \le s_q$ bi to pomenilo, da je q-jev časovni interval podmnožica u-jevega, torej bi v množici $S^\star$ lahko zamenjali u-ja s q-jem, množica pa bi ostala veljavna in enako velika kot prej. Njen presek s S-jem bi se mogoče povečal (če $u \notin S$); toda to je protislovje (zaradi tega, kako smo izbrali $S^\star$); torej bi v resnici njen presek s S-jem ostal enak; toda zdaj bi se vsaj število neujemanj tipa „potnik je v S, ne pa v $S^\star$“ pred prvim neujemanjem tipa „potnik je v $S^\star$, ne pa v S“ (to je tisto pri r) zmanjšalo (ker bi izginilo neujemanje pri q, ki bi ga po novem vsebovali obe množici), kar pa je spet protislovje (zaradi tega, kako smo izbrali $S^\star$).

Torej mora biti $s_u > s_q$. Obenem je seveda $s_u < s_r$, saj smo pri definiciji u-ja rekli, da mora u pokrivati vsaj del intervala $(s_q, \min\{e_q, s_r\})$. Kaj se zgodi, če v $S^\star$ zamenjamo u s q-jem? Edini časovni interval, v katerem bi bil avtomobil po taki

spremembi bolj zaseden kot prej, je $(s_q, \min\{e_q, s_u\})$. Če ne bi bil avtomobil v tem intervalu zdaj nikoli prezaseden, bi bila nova množica $S^\star - \{u\} \cup \{q\}$ veljavna, enako velika kot $S^\star$, obenem pa bi imela pred prvim neujemanjem tipa „potnik je v $S^\star$, ne pa v S“ (to je tisto pri r) manj neujemanj tipa „potnik je v S, ne pa v $S^\star$“ kot prej (ker neujemanja pri q ni več); to pa je protislovje zaradi načina, kako smo izbrali $S^\star$.

Torej mora biti v množici $S^\star$ že zdaj takšen nabor potnikov, da je avtomobil vsaj nek del intervala $(s_q, \min\{e_q, s_u\})$ polno zaseden. Ker je $s_u < s_r$, mora biti ta del polno zaseden tudi v $S^\star - \{r\}$ (saj r na tisti interval pred časom s_r nima vpliva); ker izmed potnikov od 1 do $r-1$ množica $S^\star$ ne vsebuje nobenega takega, ki ni prisoten tudi v $S - \{q\}$, in ker pri $S - \{q\}$ avtomobil v tistem intervalu gotovo nikoli ni polno zaseden (ker sicer S ne bi mogel vsebovati še potnika q), sledi, da mora vsebovati $S^\star$ še nekega potnika s številko, večjo od r, ki pokrije vsaj del intervala $[s_q, s_u)$. Recimo temu potniku v; da bo lahko v posegel v čas pred s_u, mora biti v seveda različen od u in veljati mora $s_v < s_u$. Obenem zaradi $v > r > q$ sledi tudi $e_u \geq e_q$. Torej, če bi bil $s_v \leq s_q$, bi bil q-jev časovni interval podmnožica v-jevega, torej bi bila $S^\star - \{v\} \cup \{q\}$ veljavna množica potnikov, kar pa je protislovje zaradi enakih razlogov kot prej pri $S^\star - \{u\} \cup \{q\}$.

Torej mora biti $s_v > s_q$. Obenem je seveda $s_v < s_u$, saj smo pri definiciji v-ja rekli, da mora v pokrivati vsaj del intervala $(s_q, \min\{e_q, s_u\})$. Zdaj lahko ponovimo razmislek iz prejšnjih dveh odstavkov, le da namesto u-ja pišemo v. Iz tega vidimo, da mora obstajati v $S^\star$ še nek potnik w, za katerega je $w > r$ in $s_q < s_w < s_v$; in potem še nek potnik x, za katerega je $x > r$ in $s_q < s_x < s_w$; in tako naprej. Tak razmislek lahko ponavljamo poljubno dolgo, kar pa je nemogoče, saj morajo biti vsi ti potniki različni (ker tvorijo njihovi začetni časi strogo padajoče zaporedje), mi pa imamo le končno mnogo potnikov. Tako smo vidimo, da se protislovju ne moremo izogniti; naša predpostavka, da S ni optimalna rešitev, je torej napačna.

Malo drugačen požrešen algoritem. Problem, s katerim se ukvarjamo v tej nalogi, se v angleščini pogosto imenuje *interval scheduling* in z njim se je ukvarjalo že precej avtorjev. Zanimivo je, da je prvi požrešni algoritem za ta problem opisal šele K. Bouzina v 90. letih prejšnjega stoletja.[11] Za razliko od zgoraj opisanega algoritma Bouzinov algoritem ni čisto požrešen, saj potnike včasih tudi pobriše iz množice S; ima pa to prednost, da je dokaz optimalnosti preprostejši, pa tudi učinkovita implementacija je (vsaj pod določenimi pogoji) preprostejša. Bouzinov algoritem deluje takole:

```
1  S := {};
2  pregleduj potnike v naraščajočem vrstnem redu času vstopa, s_i:
3     naj bo i trenutni potnik;
4     S := S ∪ {i}; (* dodaj potnika i v množico S *)
5     če je zdaj S taka, da je v avtomobilu kdaj k + 1 potnikov hkrati:
6        pobriši iz S potnika z najkasnejšim časom izstopa, e_j;
```

Časovna zahtevnost je zdaj: $O(m \log m)$ ali pa $O(n + m)$ za urejanje potnikov po vstopnem času (pri naši nalogi tega sicer ne potrebujemo, ker nam že besedilo naloge

[11] Khalid Bouzina: *On interval scheduling problems: a contribution.* Ph. D. thesis, Case Western Reserve University, 1994 (str. 40). Na str. 31 omenja, da so pred tem poznali za ta problem le bolj zapletene in manj učinkovite algoritme, ki so temeljili na pretokih po grafih in so imeli časovno zahtevnost $O(m^2 \log m)$.

zagotavlja, da bomo potnike dobili urejene naraščajoče po času vstopa); nato pa imamo m iteracij glavne zanke. Tu imamo več možnosti: (1) Za preverjanje, ali je avtomobil kdaj prepoln, lahko uporabimo drevo segmentov, enako kot zgoraj pri našem prvotnem požrešnem algoritmu. Drevo segmentov bi lahko prilagodili tudi za ugotavljanje, kateri potnik ima najkasnejši čas izstopa; vsaka iteracija glavne zanke bi nam zdaj vzela $O(\log \min\{m, n\})$ časa. (2) Če je število mest, n, majhno, je namesto drevesa segmentov lažje hraniti natančen seznam potnikov za vsak časovni interval (od enega mesta do naslednjega); vsaka iteracija glavne zanke nam bo vzela $O(n)$ časa. (3) Še ena možnost pa je, da vzdržujemo seznam potnikov, ki se trenutno peljejo v avtomobilu. Na začetku vsake iteracije pomečemo iz seznama tiste, ki so izstopili ob času s_i (ali prej), nato pa vanj dodamo potnika i. Če nato iz množice S kakšnega potnika pobrišemo, je to gotovo nek tak potnik, ki je ob tem času tudi v avtomobilu in ga je zato treba zbrisati tudi iz tistega seznama. Korist tega seznama trenutnih potnikov je, da nam že preprost pogled na dolžino tega seznama pove, kdaj je avto prepoln. Tako nam vsaka iteracija glavne zanke vzame le $O(k)$ časa; to je torej koristno, če imamo malo prostora za potnike.

Časovna zahtevnost je tako v splošnem $O(m \log m)$; pri majhnem n lahko dobimo $O(m \log n)$, z veliko preprostejšo implementacijo pa $O(mn)$; pri majhnem k lahko dobimo $O(mk)$, v kar pa ni vštet čas urejanja potnikov po času vstopa.

Dokaz pravilnosti in optimalnosti. Opazimo lahko, da na začetku vsake iteracije glavne zanke velja invarianta, da v S ni nobene skupine $k + 1$ (ali več) potnikov, ki bi morali kdajkoli hkrati sedeti v avtomobilu. Iz te invariante sledi, da je tudi ob koncu algoritma množica S dopustna rešitev (da nikoli ni v avtomobilu več kot k potnikov hkrati). Prepričajmo se, da je tudi najboljša možna.

Recimo, da ni tako; naj bo $S^\star$ najboljši možni izbor potnikov in predpostavimo, da je $|S^\star| > |S|$. Če je možnih več tako dobrih množic $S^\star$, izberimo med njimi tisto, ki ima največji presek z množico S.

Ker je $|S^\star| > |S|$, mora $S^\star$ vsebovati nekega potnika j, ki ga ni v množici S (če je takih več, naj bo j med njimi tisti, ki vstopi najbolj zgodaj). Torej je S takrat, ko smo j-ja vrgli ven, vseboval še k drugih potnikov, ki so se nekje prekrivali z j-jem, njihovi izstopni časi pa so manjši ali enaki kot pri j; recimo tem potnikom $j_1, \ldots, j_k$. Kasneje smo mogoče kakšnega od teh j_i vrgli ven, ampak če se je to zgodilo, pomeni, da smo takrat sprejeli v S nekega takega potnika t, ki se prekriva z j_i, ki ima $s_t \geq s_j$ (ker naš algoritem pač gleda potnike po naraščajočem času vstopa in ker smo j-ja nekoč prej že vrgli iz S, torej smo ga že takrat obdelali) in ki ima $e_t \leq e_{j_i}$ (drugače ne bi vrgli iz množice potnika j_i, pač pa potnika t). Kot smo videli prej, je $e_{j_i} \leq e_j$; torej je tudi $e_t \leq e_j$. Skupaj s $s_t \geq s_j$ sledi, da se tudi t-jev interval prekriva z j-jevim, prav tako kot se je j_i-jev interval prekrival z j-jevim. Torej lahko v nadaljevanju namesto potnika j_i gledamo potnika t. Tako nadaljujemo do konca postopka in vidimo, da tudi na koncu množica S vsebuje k potnikov, ki se prekrivajo z j-jem, izstopili pa so prej kot j (ali ob istem času).

Med temi k potniki $j_1, \ldots, j_k$ je gotovo vsaj en tak, ki ga ni v množici $S^\star$ (zakaj? ker $S^\star$ vsebuje potnika j in če bi vsebovala še vse $j_1, \ldots, j_k$, bi bilo tu skupaj $k + 1$ potnikov, ki se vsaj nekaj časa vsi peljejo skupaj, torej $S^\star$ ne bi bila dopustna rešitev); recimo temu potniku t. Ločimo dva primera:

(1) Če je $s_t \geq s_j$: ker je t eden od $j_1, \ldots, j_k$ in ker imajo vsi ti izstopne čase

$e_{j_i} \leq e_j$, sledi, da bi v množici $S^\star$ lahko j-ja vrgli ven in namesto njega sprejeli t-ja; avtomobil ne bi bil s tem nikoli bolj zaseden, kot je bil pred to spremembo.

(2) Če je $s_t < s_j$: recimo, da bi iz $S^\star$ nagnali j-ja in namesto njega sprejeli t-ja. Ali bi bile lahko zaradi tega kakšne težave s prezasedenostjo avtomobila? Bile bi le v primeru, če bi v stari $S^\star$ obstajalo k potnikov, ki se peljejo hkrati nekje znotraj intervala (s_t, s_j) — to je namreč edini interval, v katerem bi bil avtomobil po zamenjavi j-ja s t-jem (v množici $S^\star$) bolj zaseden kot prej. Toda vsi ti potniki bi torej gotovo vstopili takrat kot t ali pa še prej. Torej tudi vsi vstopijo pred j-jem. Spomnimo se, da smo j izbrali tako, da je to med vsemi potniki iz $S^\star - S$ tisti z najzgodnejšim časom vstopa. Torej vsi potniki, ki jih vsebuje $S^\star$ in vstopijo prej kot j, pripadajo tudi množici S. Torej, če bi obstajala taka skupina $k+1$ potnikov (s t-jem vred), ki se peljejo hkrati nekje na intervalu $[s_t, s_j)$, bi bili vsi ti potniki tudi v S in torej S ne bi bila dopustna rešitev, kar je protislovje.

V obeh primerih torej vidimo, da bi $S^\star$ ostala dopustna rešitev (in enako dobra po številu potnikov), če bi iz nje vrgli potnika j in dodali potnika t. Toda tako popravljena $S^\star$ bi imela z množico S zdaj en skupen element več kot prej. Mi pa smo prej rekli, da smo izbrali $S^\star$ tako, da ima (med optimalnimi rešitvami) največji možni presek z množico S. Tako smo prišli v protislovje, torej sledi, da je S že tudi sama v resnici optimalna rešitev.

Naloge so sestavili: lemingi, štoparji — Nino Bašić; šahovnica — Nino Bašić in Mark Martinec; poraba goriva — Primož Gabrijelčič; vsote — Uroš Jovanovič; DKIM — Mark Martinec; nizi — Mojca Miklavec; ceste — Marjan Šterk; hanojski stolpi, pleskarji — Mitja Trampuš; enačbe — Miha Vuk; bančni računi, pogrešane osebe — Klemen Žagar; zlogovna pisava, nebotičniki — Janez Brank; komparatorji — Janez Brank in Mark Martinec; algoritmi — Mojca Miklavec in Janez Brank. Hvala Klemnu Kendi za implementacijo rešitve nebotičnikov.

Urednik bi se rad tudi še posebej zahvalil Sebastjanu Misleju za obilno pomoč v zvezi z logotipom podjetja Quintelligence.

REŠITVI NALOG ZA OGREVANJE

1. Nizi

Naloga je sestavljena iz dveh delov: (1) preveriti moramo, če se kot podniz v sporočilu pojavlja kakšna od oblik besede *vojna* (to pa so: *vojna, vojne, vojni, vojno, vojn, vojnama, vojnah, vojnam* in *vojnami*) in (2) če se pojavlja kot cela beseda (ne pa le kot podniz neke daljše besede). Pogoj (2) je pomemben zato, da ne bomo cenzurirali pisem z neškodljivimi besedami, kot je *ovojnica*.

Pogoj (1) zahteva iskanje pojavitev neke skupine krajših nizov kot podnizov v nekem daljšem nizu; v malo splošnejši obliki smo srečali ta problem pri 3. nalogi za 3. skupino (str. 23; kup idej za rešitev je na str. 56–68). Pri naši nalogi pa s tem ne bomo preveč kompliciral in bomo preprosto za začetek poiskali črke *vojn*, nato pa preverili, če tem črkam sledi ena od primernih končnic.

Pri pogoju (2) lahko npr. preverimo, da znaka pred in za najdeno pojavitvijo podniza *vojna* (ali neke njene druge oblike) nista črki; posebej moramo obravnavati primer, ko se beseda *vojna* pojavlja na začetku ali na koncu sporočila.

```
function JeCenzuraPotrebna(S: string): boolean;
const Crke = ['A'..'Z', 'a'..'z', 'Č', 'č', 'Š', 'š', 'Ž', 'ž'];
      Koncnice: array [1..9] of string= ('', 'a', 'ah', 'am', 'ama', 'ami', 'e', 'i', 'o');
var i, j, k, L, LK: integer;
begin
  L := Length(S);
  { Postavimo niz S v same male črke. }
  for i := 1 to L do if S[i] in ['A'..'Z'] then S[i] := Chr(Ord(S[i]) - Ord('A') + Ord('a'));
  { Poiščimo pojavitve besede vojna v nizu S. }
  for i := 1 to L - 3 do begin
    { Ali se tu pojavljajo črke „vojn“? }
    if not ((S[i] = 'v') and (S[i + 1] = 'o') and (S[i + 2] = 'j') and (S[i + 3] = 'n'))
      then continue;
    { Ali je to na začetku besede? }
    if i > 1 then if S[i - 1] in Crke then continue;
    { Preverimo, če sledi primerna končnica. }
    for j := 1 to 9 do begin
      k := 0; LK := Length(Koncnice[j]);
      if k + LK > L then
        continue; { Končnica je predolga, tu smo že preveč pri koncu niza S. }
      while k < LK do
        if S[i + 4 + k] = Koncnice[j, 1 + k] then k := k + 1 else break;
      if k < LK then continue; { Očitno smo opazili neko neujemanje. }
      { Preverimo, če je tukaj konec besede. }
      if i + 4 + LK <= L then if S[i + 4 + LK] in Crke then
        continue; { Ni konec besede. }
      JeCenzuraPotrebna := true; exit; { Našli smo pojavitev neke oblike besede „vojna“. }
    end; {for j}
  end; {for i}
  JeCenzuraPotrebna := false; { Če smo prišli do sem, očitno nismo našli nobene pojavitve. }
end; {JeCenzuraPotrebna}
```

Še rešitev v C-ju:

```
#include <stdlib.h>
#include <string.h>
```

```
#include <ctype.h>
#include <stdbool.h>

bool JeCenzuraPotrebna(const char *S)
{
  static const char *koncnice[] = { "", "a", "ah", "am", "ama", "ami", "e", "i", "o", 0 };
  int i, j, k, L = strlen(S); char *s; const char *p;

  /* Pripravimo si kopijo niza S, v kateri bodo vse črke male. */
  s = (char *) malloc(L + 1);
  for (i = 0; i <= L; i++) s[i] = tolower(S[i]);

  /* Sprehodimo se po nizu s. */
  for (i = 0; i + 4 <= L; i++)
  {
    /* Ali se tu pojavljajo črke „vojn“? */
    if (s[i] != 'v' || s[i + 1] != 'o' || s[i + 2] != 'j' || s[i + 3] != 'n') continue;

    /* Ali je to na začetku besede? */
    if (i > 0 && isalpha(s[i - 1])) continue;

    /* Poglejmo, če sledi primerna končnica. */
    i += 4;
    for (j = 0; koncnice[j]; j++)
    {
      p = koncnice[j]; k = 0;

      /* Primerjajmo niz s (od i-tega mesta naprej) in končnico p. */
      while (*p && s[i + k] == *p) p++, k++;
      if (*p) continue; /* Očitno smo opazili neko neujemanje. */

      if (! isalpha(s[i + k])) /* Našli smo pojavitev neke oblike besede „vojna“. */
        { free(s); return true; }
    }
  }
  free(s); /* Pospravimo za sabo. */
  return false; /* Če smo prišli do sem, očitno nismo našli nobene pojavitve. */
}
```

2. Algoritmi

Podnalogi (a) in (b) sta si zelo podobni, le da uporablja prva desetiški sestav, druga pa sedmiškega. Torej ju lahko rešimo obe na en mah, če tudi osnovo uporabljenega številskega sestava gledamo kot parameter — recimo mu b; pri podnalogi (a) bomo torej vzeli $b = 10$, pri podnalogi (b) pa $b = 7$.

Preprosta, vendar zelo neučinkovita rešitev lahko pregleduje cela števila v naraščajočem vrstnem redu, od 0 naprej; vsako število pretvori v niz (glede na izbrano osnovo b), ob tem pa sešteva dolžine dobljenih nizov, tako da vidi, koliko števk se je doslej že nabralo. Takoj ko opazimo, da je skupno število števk doseglo (ali preseglo n), vemo, da je n-ta števka celotnega zaporedja nekje v pravkar pretvorjenem številu (tudi ni težko ugotoviti, katera po vrsti v tem številu je). Neugodno pri tej rešitvi je, da poraba časa narašča sorazmerno z n-jem — če nas zanima npr. števka na mestu $n = 10^{10}$, se bo naš podprogram izvajal kar nekaj časa.

Do učinkovitejše rešitve pridemo, če malo bolje pogledamo, kaj prispevajo posamezna števila v naše neskončno zaporedje števk:

- števila $0, 1, \ldots, b - 1$ prispevajo po eno števko;

- števila $b, b+1, \ldots, b^2-1$ prispevajo po dve števki;
- števila $b^2, b^2+1, \ldots, b^3-1$ prispevajo po tri števke;
- itd.

Pri tem tudi vidimo, da je v prvi skupini b števil, v drugi jih je b^2-b, v tretji jih je b^3-b^2 in tako naprej. (Konkretni primer: pri $b=10$ imamo najprej 10 števil, ki prispevajo po eno števko; nato 90 števil (namreč $10, 11, \ldots, 99$), ki prispevajo po dve števki; nato 900 števil (namreč $100, 101, \ldots, 999$), ki prispevajo po tri števke; itd.)

Ker torej za vsako skupino točno vemo, koliko števil je v njej in koliko števk prispeva vsako od njih, tudi vemo, koliko števk prispeva cela skupina: v k-ti skupini je $b^{k-1}(b-1)$ števil, vsako od njih je dolgo k števk, skupaj torej prispevajo $b^{k-1}(b-1)k$ števk. (Izjema je prva skupina, v kateri je b števil, ne pa $b-1$, kar bi sicer sledilo iz gornje formule.) Zdaj nam ni treba pregledovati vsakega števila posebej, ampak cele skupine; tako bomo zelo hitro prišli do pravega k. Ko bomo vedeli, v kateri skupini je iskana števka in katera po vrsti je med števkami te skupine, pa tudi ne bo težko izračunati, v katerem številu leži ta števka in katera po vrsti je med števkami tega števila.

```
program Butalci;

function Champernowne(b, n: integer): integer;
var k, StStevil, a: integer;
begin
  { Ničlo na začetku obravnavajmo kot poseben primer. }
  if n = 0 then begin Champernowne := 0; exit end;
  { Poglejmo, v kateri skupini leži n-ta števka. }
  k := 1; StStevil := b - 1; a := 1;
  while n > k * StStevil do begin
    { n ne leži v k-ti skupini (ki vsebuje števila od a do a + StStevil - 1,
      vsako pa je dolgo k števk); premaknimo se mimo te skupine. }
    n := n - k * StStevil;
    { Zdaj bomo pri (k + 1)-vi skupini. }
    k := k + 1; a := a + StStevil;
    StStevil := StStevil * b;
  end; {while}
  { Števka, ki nas zanima, je torej n-ta v k-ti skupini. Ker ima vsako število
    v tej skupini k števk, leži (n - 1) div k števil v celoti levo od n-te števke. }
  a := a + (n - 1) div k;
  n := n - ((n - 1) div k) * k;
  { Zanima nas torej n-ta števka števila a; vendar je to n-ta z leve,
    pri pretvarjanju števil v nize pa je lažje gledati števke z desne
    (tam so enice, ki jih najlažje dobimo iz števila a). }
  n := k + 1 - n;
  { Zdaj nas zanima nas n-ta števka števila a, gledano z desne. }
  while n > 1 do
    { Odrežimo a-ju najbolj desno števko, n pa zmanjšajmo za 1. }
    begin a := a div b; n := n - 1 end;
  { Zdaj nas zanima preprosto najbolj desna števka števila a. }
  Champernowne := a mod b;
end; {Champernowne}
```

```
var n: integer;
begin {Butalci}
  ReadLn(n);
  WriteLn(Champernowne(10, n)); { ali pa Champernowne(7, n) }
end. {Butalci}
```

Zapišimo to rešitev še v C-ju:

```
#include <stdio.h>

int Champernowne(int b, int n)
{
  int k, StStevil, a;

  /* Ničlo na začetku obravnavajmo kot poseben primer. */
  if (n == 0) return 0;

  /* Poglejmo, v kateri skupini leži n-ta števka. */
  k = 1; StStevil = b - 1; a = 1;
  while (n > k * StStevil)
  {
    /* n ne leži v k-ti skupini (ki vsebuje števila od a do a + StStevil - 1,
       vsako pa je dolgo k števk); premaknimo se mimo te skupine. */
    n -= k * StStevil;
    /* Zdaj bomo pri (k + 1)-vi skupini. */
    k++; a += StStevil; StStevil *= b;
  }

  /* Števka, ki nas zanima, je torej n-ta v k-ti skupini.
     Ker ima vsako število v tej skupini k števk, leži (n - 1) div k
     števil v celoti levo od n-te števke. */
  a += (n - 1) / k;
  n -= ((n - 1) / k) * k;

  /* Zanima nas torej n-ta števka števila a; vendar je to n-ta z leve,
     pri pretvarjanju števil v nize pa je lažje gledati števke z desne
     (tam so enice, ki jih najlažje dobimo iz števila a). */
  n = k + 1 - n;

  /* Zanima nas n-ta števka števila a, gledano z desne. */
  while (n > 1)
    a /= b, n--; /* Odrežimo a-ju najbolj desno števko, n pa zmanjšajmo za 1. */

  /* Zdaj nas zanima preprosto najbolj desna števka števila a. */
  return a % b;
}

int main()
{
  int n; scanf("%d", &n);
  printf("%d\n", Champernowne(10, n)); return 0;
}
```

Mimogrede, s takšnimi zaporedji se ne ukvarjajo le Butalci, pač pa tudi matematiki. Če za ničlo na začetku zaporedja vrinemo decimalno vejico, lahko vse skupaj preberemo kot neko realno število, ki ga imenujemo Champernownova konstanta C_b (gl. npr. MathWorld *s. v.* "Champernowne Constant").

REZULTATI

Spodnje tabele prikazujejo vrstni red vseh tekmovalcev, ki so sodelovali na letošnjem tekmovanju. Poleg skupnega števila doseženih točk je za vsakega tekmovalca navedeno tudi število točk, ki jih je dosegel pri posamezni nalogi. V prvi in drugi skupini je mogoče pri vsaki nalogi doseči največ 20 točk, v tretji skupini pa največ 100 točk.

Načeloma se v prvi skupini podeli tri prve, tri druge in tri tretje nagrade; v drugi skupini dve prvi, dve drugi in dve tretji nagradi; v tretji skupini pa po eno prvo, drugo in tretjo nagrado. Letos smo v drugi skupini izjemoma podelili eno drugo in tri tretje nagrade, ker sta si dva tekmovalca delila četrto mesto (po točkah), razlika med njima in tistim na tretjem mestu pa je bila precejšnja.

Nesorazmerje v številu nagrad po skupinah (devet v prvi, šest v drugi in tri v tretji) izvira še iz časov, ko je bilo v prvi skupini veliko več tekmovalcev kot v drugi, v drugi pa več kot v tretji. Ker je zadnja leta število tekmovalcev v vseh treh skupinah približno enako (letos jih je bilo v prvi celo manj kot v ostalih dveh), nameravamo v prihodnje število nagrad po skupinah izenačiti, tako da bomo v vsaki skupini podelili šest nagrad.

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1	1	Peter Koželj	1	ZRI + Gim. Bežigrad	19	20	19	20	20	98
1	2	Žiga Ham	1	ZRI	20	20	20	20	16	96
1	3	Matjaž Payrits	2	Gimnazija Bežigrad	20	15	20	15	20	90
2	4	Jošt Stergar	1	Gimnazija Bežigrad	15	18	19	20	17	89
2	5	Robert Slak	4	ŠC Novo mesto	18	20	17	15	16	86
2	6	Rok Kralj	1	Gimnazija Vič	15	14	17	20	18	84
3	7	Jure Henigsman	4	ŠC Novo mesto	0	13	16	20	18	67
3	8	Andrej Slapnik	2	ŠC Velenje — PTERŠ	19	12	9	15	7	62
3	9	Marko Zabreznik	2	ŠC Velenje — PTERŠ	15	5	17	15	5	57
	10	Matija Rezar	1	Gimnazija Kranj	19	0	18	0	16	53
	10	Ludvik Zobec	2	Licej Fr. Prešeren, Trst	5	4	17	10	17	53
	12	Jernej Legiša	2	Licej Fr. Prešeren, Trst	0	5	20	20	7	52
	13	Marko Štemberger	3	ŠC Novo mesto	12	17	0	8	12	49
	14	Robi Cvirn	2	ŠC Celje — PTEKŠ	0	18	19	0	0	37
	15	David Kropivšek	3	SERŠ Maribor	5	16	0	0	10	31
	16	Nace Kranjc	2	Gim. Jurija Vege Idrija	5	9	0	0	14	28
	17	Leon Pajk	3	ŠC Novo mesto	2	6	0	7	12	27
	18	David Belovič	4	Gimnazija Ormož	2	5	1	0	18	26
	18	Andrej Viršček	1	ZRI + ŠC Ljubljana	2	1	1	10	12	26
	20	Damjan Kovačič	3	Gimnazija Brežice	0	18	1	0	0	19
	20	David Pozar	2	Licej Fr. Prešeren, Trst	5	2	0	0	12	19
	22	Andrej Dolinar	4	Gimnazija Ormož	1	5	1	0	10	17
	23	Miha Lavtar	3	TŠC Kranj	0	0	0	0	16	16
	24	Robert Zorko	3	Gimnazija Brežice	2	0	0	10	0	12
	25	Anže Rožman	3	SŠER Ljubljana	0	0	0	0	0	0

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1	1	Domen Blenkuš	3	ZRI	15	20	18	20	20	93
1	2	Blaž Peterlin	3	ŠC Novo mesto	17	20	16	20	19	92
2	3	Nace Hudobivnik	2	Škof. klas. gimn. Lj.	16	10	18	20	19	83
3	4	Gašper Černevšek	3	SŠER Ljubljana	8	10	11	20	19	68
3	4	Jakob Drešar	4	Gimnazija Kranj	18	0	14	18	18	68
3	6	Marko Balkovec	3	ŠC Novo mesto	10	10	7	16	20	63
	7	Matej Jakop	4	ŠC Celje	5	20	15	5	15	60
	8	Sandi Verdev	3	ŠC Celje, Gimnazija Lava	15	5	4	12	17	53
	9	Klemen Kobetič	3	ŠC Novo mesto	10	0	6,5	20	14	50,5
	10	Bernard Klinc	4	SERŠ Maribor	15	1	12	20	0	48
	11	Naško Janež	1	Gim. Jurija Vege Idrija	0	15	0	18	14	47
	12	Patrick Zver	3	SPTŠ Murska Sobota	8	8	4,5	20	5	45,5
	13	Matic Redek	4	ŠC Novo mesto	5	12	8	3	15	43
	14	Jurij Zibler	4	Gimnazija Kranj	8	20	2,5	0	10	40,5
	15	Amela Rakanović	4	ZRI + SŠ Srečka Kosovela Sežana + Gim. Vič	4	12	1,5	0	14	31,5
	16	Rok Slamek	2	SPTŠ Murska Sobota	5	5	0	0	20	30
	17	Silvo Debelak	3	SPTŠ Murska Sobota	0	10	0	19	0	29
	18	David Vidmar	3	ŠC Novo mesto	9	0	0	17	2	28
	19	Simon Lušenc	4	ŠC Celje, Gimnazija Lava	10	7	1	7	0	25
	20	Daniel Brulc	4	ŠC Novo mesto	0	10	0	0	14	24
	21	Rok Herman	4	ŠC Celje, Gimnazija Lava	4	16	0	0	0	20
	21	Roman Šuster	3	SPTŠ Murska Sobota	0	8	2	0	10	20
	21	Tomaž Turner	2	SPTŠ Murska Sobota	5	0	1	0	14	20
	24	Zoran Felbar	2	SPTŠ Murska Sobota	1	0	3	0	13	17
	25	Miha Jerič	3	SPTŠ Murska Sobota	0	10	1	0	5	16
	26	Sandi Srkoč	2	SPTŠ Murska Sobota	2	0	2	0	10	14
	27	Matjaž Lovše	3	ŠC Novo mesto	8	0	4	0	0	12
	28	Blaž Berglez	3	SC Celje — PTEKŠ	0	0	8	0	0	8
	28	Davor Gaberšek	3	ŠC Celje — PTEKŠ	7	1	0	0	0	8
	28	Gašper Govek	3	ŠC Celje — PTEKŠ	0	0	3	5	0	8
	31	Gregor Povše	3	ŠC Celje — PTEKŠ	0	0	1	0	0	1

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1	1	Jan Berčič	3	ZRI	100	94	70	100	16	380
2	2	Tomaž Hočevar	4	ZRI	97	40	47	100	93	377
3	3	Žiga Osolin	4	Gimnazija Bežigrad	97	37	30	90	92	346
	4	Aleš Bizjak	4	ŠC Krško-Sevnica	90	85	0	0	21	196
	5	Damir Srpčič	4	ŠC Novo mesto	0	24	0	77	41	142
	6	Matic Vrščaj	4	SŠER Ljubljana	0	27	0	90	3	120
	7	Nejc Ramovš	4	ŠC Novo mesto	0	10	0	80	0	90
	8	David Vodnik	3	Gimnazija Kranj	0	0	0	81	0	81
	9	Gašper Ažman	4	Gimnazija Vič	0	0	0	80	0	80
	10	Miha Lunar	3	SŠER Ljubljana	0	0	0	0	63	63
	11	Danijel Žlaus	4	ŠC Celje	0	21	24	0	0	45
	12	Štefan Kohek	4	SERŠ Maribor	0	1	27	0	0	28
	13	Aljaž Čeru	3	SŠER Ljubljana	10	4	0	0	0	14
	14	David Mohar	2	SŠER Ljubljana	0	12	0	0	0	12
	15	Jernej Kavka	3	SERŠ Maribor	0	10	0	0	0	10
	16	David Božjak	3	SŠER Ljubljana	0	7	0	0	0	7
	16	Aleš Grašič	3	SERŠ Maribor	0	7	0	0	0	7
	18	Marko Cungl	4	SERŠ Maribor	0	0	0	0	0	0
	18	Armin Djogić	3	SŠER Ljubljana	0	0	0	0	0	0
	18	Miha Kočevar	4	ŠC Celje — PTEKŠ	0	0	0	0	0	0
	18	Primož Korošec	3	SŠER Ljubljana	0	0	0	0	0	0
	18	Primož Perhač	4	ŠC Celje — PTEKŠ	0	0	0	0	0	0
	18	Matej Rojko	3	SŠER Ljubljana	0	0	0	0	0	0

NAGRADE

Za nagrado so najboljši tekmovalci vsake skupine prejeli naslednjo strojno opremo:

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Peter Koželj	500 GB zunanji disk
1	1	Žiga Ham	2 GB iPod nano
1	1	Matjaž Payrits	2 GB iPod nano
1	2	Jošt Stergar	320 GB zunanji disk
1	2	Robert Slak	320 GB zunanji disk
1	2	Rok Kralj	8 GB USB flash disk
1	3	Jure Henigsman	8 GB USB flash disk
1	3	Andrej Slapnik	Logitech QuickCam Pro 5000
1	3	Marko Zabreznik	Logitech QuickCam Pro 5000
2	1	Domen Blenkuš	500 GB zunanji disk
2	1	Blaž Peterlin	500 GB zunanji disk
2	2	Nace Hudobivnik	16 GB USB flash disk
2	3	Jakob Drešar	16 GB USB flash disk
2	3	Gašper Černevšek	320 GB zunanji disk
2	3	Marko Balkovec	320 GB zunanji disk
3	1	Jan Berčič	4 GB iPod nano
3	2	Tomaž Hočevar	4 GB iPod nano
3	3	Žiga Osolin	500 GB zunanji disk
Tekmovanje programov — Lov na zaklade			
$20 \times 20 \times 10$		Tomaž Hočevar	Amebisov prevajalni sistem Presis
$100 \times 100 \times 100$		Tomaž Hočevar	500 GB zunanji disk
Off-line naloga — Avtomobili			
		Tomaž Hočevar	Powerball

Nagrajenci so prejeli tudi naslednje knjižne nagrade:

R. B. Banks: *Ledene gore, padajoče domine*	(4×)
B. Bryson: *Kratka zgodovina skoraj vsega*	(4×)
R. Dawkins: *Sebični gen*	(4×)
S. Hawking: *Na ramenih velikanov*	(3×)
S. Hawking: *Vesolje v orehovi lupini*	(3×)
R. Jamnik: *Teorija iger.*	(4×)
J. S. Rosenthal: *Ko strela udari: skrivnostni svet verjetnosti*	(4×)
S. Singh: *Knjiga šifer*	(3×)
G. Staguhn: *V iskanju najmanjšega delca sveta*	(3×)
J. Strnad: *Mala kvantna fizika*	(4×)
R. J. Wilson, J. J. Watkins: *Uvod v teorijo grafov*	(4×)

Poleg tega je vsak od nagrajencev prejel tudi izvod knjige *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004* (v dveh zvezkih, IJS, 2006).

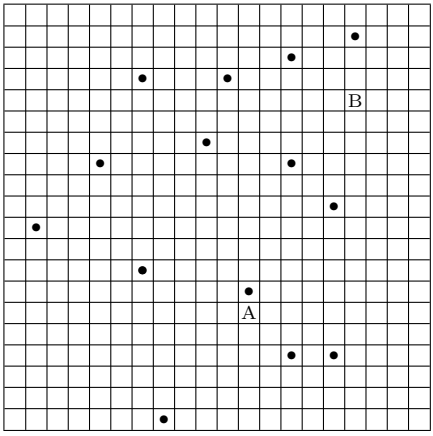
SODELUJOČE ŠOLE IN MENTORJI

Gimnazija Bežigrad, Ljubljana	Bojan Šernek, Andrej Šuštaršič
Gimnazija Brežice	Tea Habinc
Gimnazija Jurija Vege, Idrija	Danica Bogataj
Gimnazija Kranj	Matevž Jekovec
Gimnazija Ormož	Lenka Keček Vavpotič
Gimnazija Vič	
Licej France Prešeren, Trst	Valentina Busechian
Srednja elektro-računalniška šola Maribor (SERŠ)	Slavko Nekrep, Manja Sovič Potisk, Jože Štrucl
Srednja poklicna in tehniška šola (SPTŠ) Murska Sobota	Simon Horvat, Milan Petrijan
Srednja šola Srečka Kosovela, Sežana	
Srednja šola za elektrotehniko in računalništvo (SŠER), Ljubljana	Andrej Bagon, Matjaž Kodela, Tea Lončarič, Nataša Makarovič, Darjan Toth
Šolski center Celje, Poklicna in tehniška elektro in kemijska šola (PTEKŠ)	Jurij Gregl, Damjan Luc, Boštjan Resinovič
Šolski center Celje, Splošna in strokovna gimnazija Lava	Borut Slemenšek
Šolski center Ljubljana, Splošna in strokovna gimnazija Ljubljana	
Šolski center Krško-Sevnica	
Šolski center Novo mesto	Mile Božič, Andrej Jakobčič, Albert Zorko
Šolski center Velenje, Poklicna in tehniška elektro in računalniška šola (PTERŠ)	Damjan Borovnik, Miran Zevnik
Škofijska klasična gimnazija, Ljubljana	Helena Medvešek
Tehniški šolski center (TŠC) Kranj	Gabrijela Kranjc
Zavod za računalniško izobraževanje (ZRI), Ljubljana	Jelko Urbančič

TEKMOVANJE PROGRAMOV — LOV NA ZAKLADE

Podobno kot lani smo tudi letos organizirali tekmovanje programov. Lani je bilo za takšno tekmovanje veliko zanimanja, vendar pa so potem svoje programe zares poslali le trije tekmovalci. To smo si poskušali delno razložiti s tem, da je bila lanska naloga mogoče pretežka oz. je zahtevala precej ukvarjanja z geometrijo, preračunavanjem koordinat v ravnini ipd., kar je mogoče marsikoga odvrnilo od reševanja. Zato smo letos pripravili nalogo podobnega tipa, vendar lažjo.[12]

Opis naloge smo objavili oktobra 2006 skupaj z razpisom za tekmovanje v znanju. Tekmovalci so imeli čas do 23. marca 2007 (teden dni pred tekmovanjem), da pošljejo svoje programe. Letošnja naloga je od tekmovalcev zahtevala, da napišejo nadzorno logiko za igralca v preprosti računalniški igri: pobiranju zakladov. Igra poteka na karirasti mreži z $n \times n$ polji. Na nekaterih poljih mreže stojijo zakladi; začetni položaj zakladov je naključen, pri čemer pa se nikoli ne zgodi, da bi več zakladov stalo na istem polju. Tudi začetni položaj igralcev je naključen (sta pa vsak na svojem polju in to oba na takih poljih, na katerih ni nobenega zaklada). Čeprav sta igralca na začetku igre na različnih poljih, pa je dovoljeno tudi to, da se kdaj kasneje med igro znajdeta oba igralca hkrati na istem polju.



Primer možnega stanja igre pri igralni plošči velikosti 20×20. Na plošči je še 14 zakladov (označeni so s pikami •). Položaj igralcev je označen s črkama A in B.

Igra poteka v korakih. Igralec se lahko v vsakem koraku premakne s svojega trenutnega polja na eno od sosednjih osmih polj (torej tistih, ki imajo s trenutnim poljem skupno vsaj eno oglišče), pri čemer se seveda ne sme premakniti izven mreže; lahko pa tudi ostane na trenutnem polju. Premiki igralcev so sinhronizirani: vedno se oba premakneta hkrati. Če se igralec premakne na polje, na katerem je doslej stal zaklad, se šteje, da je igralec ta zaklad pobral (igralec dobi eno točko, zaklad pa izgine). Če oba igralca hkrati pobereta isti zaklad, dobi vsak od njiju polovico točke.

[12] Letos se je igra dogajala na diskretni (karirasti) mreži; igralca sta si enakovredna in je dovolj, če napišemo en sam program (lani sta imeli mačka in miš bistveno različne cilje in vzorce obnašanja, zato je bilo treba za vsako napisati čisto drugačno nadzorno logiko); gibanje igralcev pa je tako omejeno, da lahko trivialno izračunamo, v koliko časa bi lahko igralec prišel do nekega ciljnega položaja.

Igra se konča, ko so pobrani vsi zakladi, ali pa po 10 000 korakih (karkoli je prej). Zmagovalec je tisti igralec, ki ima največ točk.

Tekmovanje programov smo izvedli v dveh težavnostnih kategorijah, ki se razlikujeta po velikosti igralne površine in po začetnem številu zakladov: lažja kategorija je potekala na mreži 20×20 z 10 zakladi, težja pa na mreži 100×100 s 100 zakladi. Vsak tekmovalec je moral napisati podprogram za krmiljenje igralca; le-ta je ob vsakem koraku dobil podatke o položaju obeh igralcev in vseh še preostalih zakladov. Ob inicializaciji dobi tekmovalčev podprogram tudi podatke o velikosti mreže in številu zakladov, tako da lahko, če hoče, svojo strategijo prilagodi tema parametroma.

Glede porabe časa in pomnilnika smo postavili precej darežljive omejitve: igralec lahko pri posamezni igri porabi do 100 sekund procesorskega časa (na 2-gigaherčnem opteronu) in do 4 GB pomnilnika, vendar so bile rešitve, ki so jih poslali naši tekmovalci, precej manj požrešne (procesorjevega časa so porabile največ kakšno sekundo, pomnilnika pa skoraj zanemarljivo malo).

Po roku za oddajo rešitev smo izvedli navzkrižno tekmovanje: za vsak par igralcev smo izvedli veliko število iger z različnimi začetnimi razporedi zakladov in igralcev (za vsak par igralcev smo izvedli 60 000 iger na mreži 20×20 z 10 zakladi in 6 000 iger na mreži 100×100 s 100 zakladi). Izkaže se, da je izid posamezne igre pogosto močno odvisen od srečnega naključja (koliko zakladov je blizu začetnega položaja posameznega igralca), zato smo za vsak izbrani začetni razpored zakladov in začetni položaj igralcev izvedli dve igri, pri čemer se je druga igra razlikovala od prve le v tem, da je bil začetni položaj obeh igralcev ravno zamenjan.

Rezultati

Dobili smo prispevke sedmih tekmovalcev (šestih dijakov in enega študenta), kar je velika izboljšava v primerjavi z lanskim tekmovanjem programov, na katerem so sodelovali le trije. Naslednji tabeli kažeta povprečno število točk, ki jih je dobil posamezni tekmovalec v vseh odigranih igrah (torej z vsemi ostalimi tekmovalci).

Lažja kategorija: 20×20 z 10 zakladi

Mesto	Ime	Šola	Točke
1	Tomaž Hočevar	ZRI	5,246
2	Žiga Osolin	Gimnazija Bežigrad	5,178
3	Jure Vrščaj	FRI	5,105
4	Aleš Bizjak	ŠC Krško-Sevnica	5,079
5	Matic Vrščaj	SŠER Ljubljana	4,842
6	Jan Berčič	ZRI	4,839
7	Rok Kralj	Gimnazija Vič	4,710

Težja kategorija: 100×100 s 100 zakladi

Mesto	Ime	Šola	Točke
1	Tomaž Hočevar	ZRI	53,447
2	Jure Vrščaj	FRI	52,307
3	Žiga Osolin	Gimnazija Bežigrad	50,801
4	Aleš Bizjak	ŠC Krško-Sevnica	49,433
5	Matic Vrščaj	SŠER Ljubljana	48,384
6	Jan Berčič	ZRI	48,368
7	Rok Kralj	Gimnazija Vič	47,357

Kot kažejo tudi ti rezultati, ima naša igra to neugodno lastnost, da so razlike v številu točk, ki jih dosežejo različne igralne strategije, precej majhne. Izkaže se, da doseže naivni igralec, ki se usmeri vedno proti najbližjemu zakladu, komaj kakšnih 10–15 % točk manj kot tisti, ki igra po strategiji min-max (ki je v nekem smislu najboljša možna; glej naslednji razdelek). To je verjetno predvsem posledica dejstva, da se igralci pri tej igri premikajo precej počasi (ker se lahko v vsakem koraku premaknemo za največ eno polje v navpični in največ eno v vodoravni smeri); tudi igralec z dobro strategijo torej porabi precej časa, da pride do zakladov, ki se jih je namenil pobrati; v tem času pa bo tudi njegov nasprotnik lahko že marsikaj pobral, četudi igra po precej slabši strategiji.

Rešitev za majhno število zakladov

Če sta tako igralna površina kot število zakladov majhni, lahko sestavimo zelo dobro igralno strategijo po načelu min-max. To načelo temelji na zamisli, da nasprotnika ne podcenjujemo, pač pa vedno domnevamo, da bo naredil takšno potezo, ki bo za nas najbolj neugodna. Mi pa poskušamo med potezami, ki so na voljo nam, vedno izbrati tako, da bo izid za nas najbolj ugoden (ob domnevi, da bo nasprotnik potem naredil tisto, kar bo za nas najbolj neugodno). Oglejmo si zdaj, kako lahko to načelo prelijemo v konkretno rešitev te naloge.

Stanje igre v kateremkoli trenutku lahko predstavimo s podatki o tem, kje sta oba igralca in kateri zakladi so še ostali nepobrani. Če imamo igralno površino velikosti $n \times n$ in je bilo na začetku igre k zakladov, je torej možnih n^2 položajev vsakega igralca, za množico še nepobranih zakladov pa je 2^k možnosti, kar nam da skupaj $n^4\,2^k$ stanj (v praksi so sicer nekatera od teh stanj nemogoča: igralec ne more stati na polju, kjer je nek še nepobran zaklad, saj bi bil moral ta zaklad pobrati, ko je stopil na tisto polje). Pri naši lažji kategoriji z igralno površino 20×20 in 10 zakladi dobimo malo manj kot 164 milijonov stanj; to je, kot bomo videli, še obvladljivo.

Recimo, da začnemo opazovati igro v nekem stanju s in da oba igralca (mi in naš nasprotnik) igrata, dokler niso pobrani vsi zakladi; vsak igra tako, kot se mu zdi, da bo zanj najbolje. Do konca igre pobereta oba igralca še preostale zaklade in tako dobita še po nekaj točk; naj bo zdaj $f(s)$ razlika med številom točk, ki jih je do konca igre dobil naš nasprotnik, in številom točk, ki smo jih do konca igre dobili mi. Očitno je za nas koristno, če uspemo igro speljati v stanja, ki imajo čim nižjo vrednost $f(s)$ — to namreč pomeni, da bo v nadaljevanju igre nasprotnik pobral malo zakladov, mi pa veliko.

Če je s stanje, v katerem so pobrani že vsi zakladi, je jasno, da bo $f(s) = 0$. Za ostala stanja pa lahko računamo $f(s)$ po načelu min-max:

$$f(s) = \min_{p \in \text{naše poteze}} \max_{r \in \text{nasprotnikove poteze}} (\delta(s,p,r) + f(\mu(s,p,r))).$$

Za nasprotnika torej domnevamo, da bo naredil tisto potezo r, ki je za nas najbolj neugodna (zato je tam max), mi pa poskušamo med svojimi potezami p izbrati tisto, ki bo pripeljala do za nas najugodnejšega rezultata (zato imamo tam min). Pri tem smo z $\delta(s,p,r)$ označili razliko med tem, koliko točk dobi nasprotnik, in tem, koliko jih dobimo mi, če je bila igra prej v stanju s in smo takrat naredili mi potezo p, nasprotnik pa potezo r. Z $\mu(s,p,r)$ pa smo označili stanje, v katerega pride igra, če v stanju s naredimo mi potezo p in nasprotnik potezo r. Možnih potez je za vsakega

igralca največ 9 (lahko ostane pri miru ali pa se premakne na eno od sosednjih osmih polj, če pri tem seveda ne bi padel čez rob igralne površine).

Težava s to formulo je, da ni tako eksplicitna, kot bi si želeli — v bistvu nam določa nek velik sistem enačb (za vsako stanje s je po ena enačba). V njej namreč $f(s)$ ne nastopa le na levi, ampak tudi na desni (če oba igralca ostaneta pri miru, se stanje ne spremeni in je $\mu(s,p,r) = s$), poleg tega pa se pogosto zgodi, da se formula za $f(s)$ sklicuje na vrednost nekega drugega stanja, $f(s')$, formula za $f(s')$ pa se sklicuje na $f(s)$ (ker se lahko igra večkrat vrne v isto stanje, npr. če se igralca premikata sem in tja, ne da bi pobrala kakšen zaklad). Ta sistem enačb bi lahko reševali npr. s tehnikami logičnega programiranja z omejitvami (CLP — *constraint logic programming*), vendar se izkaže, da je za 164 milijonov stanj ta pristop že prepočasen.

Lahko pa si problem malo poenostavimo in računamo nek približek funkcije f, ki bo za potrebe vodenja igralca še vedno zelo uporaben. Zgornja formula za $f(s)$ gleda le po en premik vsakega igralca; v enem premiku pa se število točk ponavadi nič ne spremeni (premiki, pri katerih vsaj eden od igralcev pobere kakšen zaklad, so relativno redki). Namesto tega si mislimo, da bi v formuli za f spremljali večje število korakov igre:

$$f(s) = \min_{p \in \text{naše poti}} \max_{r \in \text{nasprotnikove poti}} (\delta(s,p,r) + f(\mu(s,p,r))).$$

Za funkcijo f je načeloma čisto nepomembno, kakšni sta točni poti gibanja igralcev, p in r; važno je le, da sta enako dolgi in da znamo pravilno izračunati $\delta(s,p,r)$ (spremembo razlike v številu točk zaradi opravljenih potez) in novo stanje $\mu(s,p,r)$.

Smiselno je vzeti tako dolge poti p in r, da na koncu vsaj ene od teh dveh poti tisti igralec pobere nek zaklad; daljših poti pa raje ne glejmo, saj bi s tem le po nepotrebnem izgubljali čas (vsak daljši odlomek igre je mogoče razrezati na krajše dele, ki se končajo s pobiranjem nekega zaklada). In ker se zdaj na poteh p in r ne dogaja nič razen tega, da se na koncu ene (ali pa mogoče obeh) pobere nek zaklad, nas podroben potek teh poti res ne zanima več. Formulo lahko torej zapišemo takole:

$$f(s) = \min_{p \in P} \max_{r \in P} (\delta(s,p,r) + f(\mu(s,p,r))).$$

(Pri tem predstavlja P množico vseh n^2 možnih položajev na mreži.) Vendar pa bomo v formuli upoštevali le take pare (p,r), pri katerih velja: (1) vsaj na enem od teh dveh položajev je v stanju s še nepobran zaklad; (2) naš igralec lahko od svojega trenutnega položaja (v stanju s) do svojega cilja p pride, ne da bi pred prihodom na p pobral kak zaklad; (3) tudi nasprotnik lahko od svojega trenutnega položaja do svojega cilja r pride, ne da bi pred prihodom na r pobral kak zaklad. (S pogojema (2) in (3) ponavadi ne bo težav; lahko bi nastopile le v primeru, če je npr. eden od igralcev v stanju s vse naokoli čisto zagrajen z zakladi, njegov cilj p ali r pa je nekje daleč stran in igralec ne more do njega, ne da bi šel čez kakšno od polj z zakladi.)

To je že skoraj dobro, vendar pa imamo z računanjem $f(s)$ še vedno preveč dela, ker je še vedno potrebno (pri vsakem s) pregledati preveč parov (p,r). Delo si lahko še omejimo, če predpostavimo, da igralcema nikoli ni v interesu delati neumnosti, kot so npr. stanje pri miru ali brezciljno tavanje sem ter tja. Predpostavili bomo, da se igralec vedno giblje po najkrajši poti od svojega trenutnega položaja do enega od še nepobranih zakladov.

algoritem za izračun $f(s)$:
  naj bo p_0 položaj našega igralca v stanju s, r_0 pa položaj nasprotnika;
  za vsak zaklad p, ki v stanju s še ni pobran:
    za vsak zaklad r, ki v stanju s še ni pobran:
      $t := \min\{|p_0 - p|, |r_0 - r|\}$;
      p_1 := položaj našega igralca po t korakih,
          če gre po najkrajši poti od p_0 do p;
      r_1 := položaj nasprotnika po t korakih,
          če gre po najkrajši poti od r_0 do r;
      $\delta := 0$;
      če je na p_1 zaklad, zmanjšaj δ za 1;
      če je na r_1 zaklad, povečaj δ za 1;
      s' := novo stanje (naš igralec v p_1, nasprotnik v r_1, pobrani zakladi
        so tisti kot v s in še morebitna zaklada na p_1 in r_1);
      zapomni si $\phi(p, r) := f(s') + \delta$;
  vrni $f(s) := \min_p \max_r \phi(p, r)$

Pri tem sta $|p_0 - p|$ in $|r_0 - r|$ dolžini najkrajših poti od p_0 do p in od r_0 do r. Pri izbiri p_1 in r_1 je gornji opis malo nenatančen: v resnici ponavadi obstaja več enako dolgih najkrajših poti od npr. p_0 do p in položaj našega igralca po t korakih je načeloma odvisen od tega, katero od teh poti si je izbral. Če bi hoteli biti res natančni, bi morali pač pregledati vse primerne p_1 in r_1, vendar bi se nam s tem čas računanja funkcije f spet neugodno povečal. Privoščimo si naslednji približek: pri določanju p_1 in r_1 predpostavimo, da se igralec do svojega ciljnega zaklada vedno giblje po takih poteh, ki od začetka uporabljajo same diagonalne premike (vodoravne in navpične premike pa šele, ko se položaj igralca razlikuje od položaja ciljnega zaklada le v eni koordinati, ne pa v obeh).

Tako smo prišli do dovolj hitrega algoritma za računanje funkcije f; če si ob tem še zapomnimo, kateri je bil pri posameznem s najugodnejši zaklad p (torej tisti, pri katerem je bil dosežen minimum vrednosti $\max_r \phi(p, r)$ — temu p-ju recimo $p^\star(s)$), lahko ta podatek uporabimo za usmerjanje našega igralca med igro.

Opazimo lahko, da je stanje s' v gornjem algoritmu vedno tako, da stoji vsaj eden od igralcev na mestu, kjer je nekoč stal nek zaklad. Zato funkcije f ni več treba računati za vsa stanja, pač pa le za tista, kjer stoji eden od igralcev na mestu nekega zaklada. Takšnih stanj je približno $n^2 \cdot k \cdot 2^k$ (oz. malo manj), kar je precej manj od števila vseh stanj ($n^4 \cdot 2^k$). Med stanji, v katerih ne stoji nobeden igralcev na mestu, kjer je bil nekoč kak zaklad, nas zanima le tisto stanje (recimo mu s_0), v katerem se igra nahaja na začetku. Naš igralec se torej na začetku usmeri proti zakladu $p^\star(s_0)$, nato pa v vsakem koraku pogleda, če je trenutno stanje s takšno, da zanj pozna vrednosti $f(s)$ in $p^\star(s)$ (torej če je to eno od tistih stanj, v katerih stoji vsaj eden od igralcev na mestu, kjer je bil nekoč nek zaklad), in če je, za svoj cilj odslej vzame zaklad $p^\star(s)$ in se začne gibati proti njemu.

Na računalniku, na katerem smo poganjali prispevke naših tekmovalcev, je opisani algoritem za izračun funkcije f pri igralni površini 20×20 z 10 zakladi porabil približno pol minute. Igralec, temelječ na tej strategiji, je v boju z ostalimi šestimi dosegel povprečno 5,418 točk na igro in bi s tem prepričljivo zmagal. Velika slabost opisanega postopka pa je seveda ta, da število stanj narašča eksponentno hitro v

odvisnosti od števila zakladov, zato pri večjem številu zakladov tako poraba časa kot poraba pomnilnika hitro narasteta prek sprejemljivih meja. Pri igralni površini 100×100 s 100 zakladi bi imel na primer naš prostor stanj že kar $100^4\cdot2^{100}\approx1{,}2\cdot10^{38}$ stanj.

Rešitev za večje število zakladov

Oglejmo si, kako deluje rešitev Tomaža Hočevarja, saj tudi nam ni uspelo sestaviti ničesar bistveno boljšega. Igralec si vzdržuje seznam naslednjih šestih zakladov, ki jih namerava obiskati.[13] Ta seznam sestavi tako, da z rekurzijo (oz. razveji-in-omeji) preišče vsa zaporedja šestih nepobranih zakladov in izbere med njimi tisto, pri katerem bi bila skupna prehojena pot (od trenutnega položaja do prvega zaklada, nato od tam do drugega in tako naprej do šestega) najkrajša.

Igralec se vedno giblje proti prvemu zakladu s svojega seznama; ko ga pobere, ta zaklad izbriše s seznama in se odtlej giblje proti tistemu, ki je zdaj prvi (prej pa je bil drugi) in tako naprej. Če opazi, da je nasprotnik pobral kakšnega od zakladov s seznama, takoj zavrže obstoječi seznam in si pripravi novega (glede na sedanji položaj in sedanjo množico še nepobranih zakladov). Enako si seveda pripravi nov seznam tudi v primeru, da je starega čisto izpraznil, ker je uspel pobrati vse zaklade na njem.[14] Še en primer, v katerem obstoječi seznam ciljev zavrže in si izračuna novega, pa je, če opazi, da je prvi zaklad z našega seznama bližje nasprotniku kot nam in da nasprotniku ni bližje noben drug zaklad kot ta — v tem primeru obstaja tveganje, da nam bo nasprotnik pobral ta zaklad, še preden bi lahko mi prišli do njega, zato je pametno, če poskusimo spremeniti svoje cilje (pri naslednjem iskanju ciljev se nalašč tudi izogibamo zakladu, ki smo ga imeli doslej prvega na seznamu).

To rešitev lahko kombiniramo tudi z zamislimi na podlagi načela min-max iz prejšnjega razdelka: v „končnici“ igre, npr. ko ostane le še m (od prvotnih k) zakladov, lahko za preostanek igre naračunamo strategijo po načelu min-max in igramo po njej. Pregledati moramo približno $n^2\cdot m\cdot2^m$ stanj, kar je pri $m=8$ in igralnem polju 100×100 približno dvajset milijonov stanj; to je še obvladljivo. Koristi od boljše igre v končnici so sicer majhne, vendar ne nepomembne: kot smo videli že iz rezultatov na str. 89, so razlike med različnimi strategijami majhne; če lahko po zaslugi načela min-max v končnici poberemo povprečno en zaklad več, kot bi jih sicer, se

[13] Razen proti koncu igre — če je ostalo le m zakladov, si pripravimo seznam naslednjih $\min\{6, m/2\}$ zakladov. Namen te hevristike je, da proti koncu igre, ko je ostalo le še razmeroma malo zakladov, ne delamo računov brez krčmarja — dokler je zakladov veliko, je precej verjetno, da sta igralca vsak na svojem koncu igralne površine in pobirata vsak svoje zaklade, ko pa ostane le še peščica zakladov, se oba zapodita vanje in nima smisla delati preveč dolgoročnih načrtov, saj nam bo nasprotnik gotovo posegel vanje s tem, ko bo pobral kakšnega od zakladov, ki smo ga sicer nameravali pobrati sami.

Izkaže se tudi, da se uspeh našega igralca še malo poveča, če namesto naslednjih šestih zakladov načrtujemo še dlje v prihodnost, npr. naslednjih osem ali celo deset zakladov. Težava pri tem je, da čas, ki ga potrebujemo, da preiščemo vse vrstne rede obiskovanja naslednjih r zakladov, narašča eksponentno v odvisnosti od r-ja. Ker so bile omejitve porabe časa pri tej nalogi precej darežljive, bi bil $r=10$ še sprejemljiv, pri večjih r pa res že tvegamo, da bomo prekoračili časovno omejitev.

[14] Pri naših poskusih se je izkazalo za še bolje, če sestavimo nov seznam ciljev vsakič, ko poberemo kak zaklad (ostale cilje z dosedanjega seznama torej zavržemo, je pa seveda mogoče, da se bodo znašli tudi na novem seznamu). Program je zaradi pogostejšega sestavljanja novih seznamov ciljev sicer porabil približno štirikrat več časa kot prej, vendar pa je tudi pobral več zakladov (proti igralcu brez te izboljšave je zmagal z rezultatom 50,5670 : 49,4330 po 10000 igrah na mreži 100×100 s 100 zakladi).

nam lahko taka izboljšava že precej pozna pri uvrstitvi. Pri naših poskusih je igralec s to izboljšavo premagal igralca brez takšne izboljšave s povprečnim rezultatom 50,3020 : 49,6380 (po 1000 igrah na mreži 100×100 s 100 zakladi).

Če skombiniramo različne tu omenjene izboljšave, se lahko naš rezultat še izboljša: pri naših poskusih je igralec, ki načrtuje 10 ciljev vnaprej, sestavi nov seznam ciljev vsakič, ko pobere kak zaklad, in v končnici (ko ostane le še 6 zakladov) uporablja pristop min-max, premagal igralca, ki načrtuje le 6 ciljev vnaprej, sestavi nov seznam ciljev le, ko se stari povsem izprazni, in v končnici ne igra nič drugače kot sicer, z rezultatom 51,7945 : 48,2055 (po 1000 igrah na mreži 100×100 s 100 zakladi).

OFF-LINE NALOGA — AVTOMOBILI

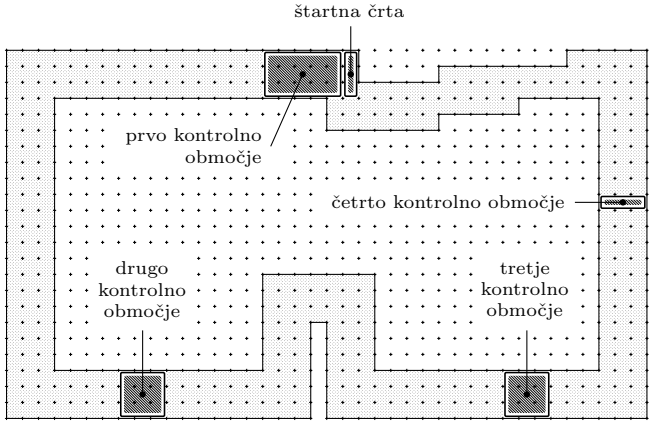
Na tekmovanju v znanju računalništva rešujejo tekmovalci pet nalog in imajo za to tri ali pet ur časa (odvisno od tega, v kateri skupini tekmujejo); v vsakem primeru morajo biti torej naloge take, da vsaj uspešnejši tekmovalci za posamezno nalogo ne bodo porabili več kot slabo uro časa. Za obširnejše naloge na tem tekmovanju ni prostora; je pa lahko reševanje kakšnih takih obširnejših nalog tudi zelo prijeten izziv, če si človek lahko vzame pri nalogi dovolj časa za razmislek, poišče in mogoče preizkusi več različnih poti do rešitve, poskuša svoje prvotne ideje še izboljšati in tako naprej. Off-line nalogo smo letos razpisali prav z namenom, da bi tekmovalce povabili k razmišljanju o malo večjem problemu, ki bi jim bil lahko zanimiv izziv, obenem pa ne bi bil tako zahteven, da ga ne bi mogli rešiti. Upali smo tudi, da bi lahko na ta način spodbudili ljudi, da bi o tematikah, ki jim je posvečeno naše tekmovanje, razmišljali tudi med šolskim letom, ne le tisto eno spomladansko soboto, ko zares poteka tekmovanje v znanju.

Nalogo (tako opis naloge kot vse testne primere) smo objavili v oktobru 2006 skupaj z razpisom tekmovanja v znanju računalništva, tekmovalci pa so imeli vse do vključno 30. marca 2007, torej do dneva pred tekmovanjem, možnost oddajati svoje rešitve prek naše spletne strani. Naš računalnik je sproti preverjal pravilnost vsake oddane rešitve in prikazoval razvrstitev tekmovalcev glede na kakovost prejetih rešitev.

Opis naloge

Na karirasti mreži velikosti $w \times h$ (w stolpcev, h vrstic) je definirano dirkališče. Vsako polje mreže je bodisi prevozno bodisi neprevozno. Nekatera izmed prevoznih polj tvorijo štartno črto, nekatera pa tvorijo kontrolna območja. Na mreži si mislimo tudi koordinatni sistem: stolpce oštevilčimo od leve proti desni s koordinatami od 0 do $w - 1$, vrstice pa od zgoraj navzdol s koordinatami od 0 do $h - 1$.

Primer:



Pot vozila po dirkališču predstavimo z zaporedjem koordinat obiskanih točk:

$$(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k).$$

Dolžino poti definiramo kot število korakov na njej; dolžina zgornje poti je torej k.

Cilj te naloge je poiskati čim krajšo pot, ki ustreza vsem naslednjim zahtevam:

- Vsako od polj (x_t, y_t) je prevozno (za $t = 0, 1, 2, \ldots, k$).
- Polje (x_0, y_0) leži na štartni črti.
- Definirajmo hitrost v času t takole:

  $$v_x(t) = x_t - x_{t-1}, \qquad v_y(t) = y_t - y_{t-1}.$$

  (Pri $t = 0$ si mislimo $v_x(t) = v_y(t) = 0$.)

  Potem zahtevamo, da sta

  $$|v_x(t) - v_x(t-1)| \text{ in } |v_y(t) - v_y(t-1)| \text{ iz množice } \{-1, 0, 1\}$$

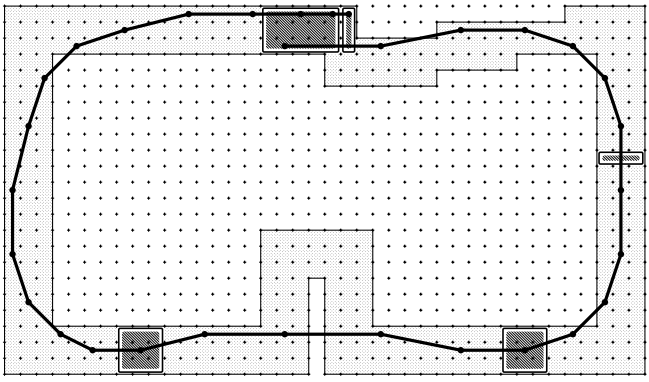
  in to za vse $t = 1, 2, \ldots, k$. Z drugimi besedami, hitrost se od enega koraka do naslednjega ne sme spremeniti za več kot za 1 v vsaki smeri.

- Pot mora obiskati vsa kontrolna območja v pravem vrstnem redu in se nato končati bodisi na štartni črti bodisi v prvem kontrolnem območju.

  Z drugimi besedami: ko enkrat obiščemo neko kontrolno območje i, od takrat naprej ne smemo več obiskati kontrolnih območij od 1 do $i - 1$. Ko zadnjič zapustimo zadnje kontrolno območje, se lahko od tam odpeljemo le še na štartno črto ali pa na prvo kontrolno območje, nato pa se moramo ustaviti.

Posebej poudarimo, da nas pri preverjanju, ali pot ustreza gornjim zahtevam, zanimajo le točke (x_t, y_t), na katerih se začenja ali končuje posamezni korak. Na primer: točki (x_{t-1}, y_{t-1}) in (x_t, y_t) morata biti prevozni, ni pa nujno, da cela daljica od (x_{t-1}, y_{t-1}) do (x_t, y_t) leži na prevoznem delu mreže (dovoljeno je torej „sekanje ovinkov“). Podobno, če niti (x_{t-1}, y_{t-1}) niti (x_t, y_t) ne ležita na nekem kontrolnem območju, potem se šteje, da v tem koraku tega kontrolnega območja nismo obiskali, četudi mogoče nek del daljice od (x_{t-1}, y_{t-1}) do (x_t, y_t) prečka kakšno od polj, ki pripadajo temu kontrolnemu območju.

Naslednja slika kaže primer poti, ki ustreza skoraj vsem zgoraj opisanim pogojem; z njo je narobe le to, da ne obišče četrtega kontrolnega območja.



Skupaj z razpisom naloge smo na naši spletni strani objavili tudi deset dirkališč (in še nekaj dodatnih, ki pa se niso upoštevala za potrebe točkovanja in razvrščanja tekmovalcev). Največje med njimi je bilo veliko 1000×1000 polj, ostala pa so bila velikosti 300×300 ali manjša.

Rezultati

Za to nalogo je bilo med tekmovalci manj zanimanja, kot smo upali, ko smo jo objavili. Vsega skupaj se je k sodelovanju pri tej nalogi prijavilo šest tekmovalcev, rešitve pa so poslali le trije. Tomaž Hočevar je že novembra 2006, le nekaj tednov po objavi naloge, poslal svoje rešitve vseh testnih primerov; pri vseh razen pri najtežjem testnem primeru so bile te rešitve optimalne. Potem dolgo ni bilo nobenega novega zanimanja za nalogo in nas je že močno skrbelo, da bo Tomaž edini udeleženec tega dela tekmovanja. Druge rešitve so se pojavile šele v zadnjih treh dneh pred tekmovanjem: Domen Blenkuš je poslal suboptimalno rešitev prvega testnega primera (baje jo je sestavil kar ročno), Jan Berčič pa je poslal optimalne rešitve devetih testnih primerov (vseh razen najtežjega). Stvari so postale celo precej napete, saj je Jan zadnjo izmed svojih rešitev poslal še ob osmih zvečer na dan pred tekmovanjem, le štiri ure pred iztekom roka za oddajo. Če bi do polnoči poslal še optimalno rešitev desetega testnega primera, bi lahko še prevzel vodstvo, vendar se to potem ni zgodilo.

Točkovanje je pri tej nalogi potekalo tako, da smo pri vsakem testnem primeru razvrstili prejete rešitve po dolžini poti in podelili tekmovalcem deset točk za prvo mesto, osem za drugo, sedem za tretje, ..., dve točki za osmo mesto in po eno točko za vsa nadaljnja mesta. Tako ali tako pa večina tega točkovalnega mehanizma ni prišla nikoli do izraza, saj pri nobenem testnem primeru nismo prejeli več kot treh rešitev. Končna razvrstitev je torej naslednja:

Tomaž Hočevar (ZRI + Gimnazija Vič)	100 točk
Jan Berčič (ZRI + Gimnazija Bežigrad)	90 točk
Domen Blenkuš (ZRI + Gimnazija Vič)	7 točk

Rešitev

Oglejmo si, kako lahko pri tej nalogi na učinkovit način pridemo do optimalnih (najkrajših možnih) poti in pri tem še varčujemo s pomnilnikom, tako da bomo lahko rešili tudi precej velike testne primere.

Stanje igre v nekem trenutku lahko opišemo s peterico (x, y, v_x, v_y, c), pri čemer sta x in y koordinati avtomobila, v_x in v_y njegova hitrost (v_x je razlika med sedanjim x in med x-koordinato v prejšnjem trenutku, podobno pa je tudi z v_y), število c pa je številka zadnjega kontrolnega območja, ki ga je na svoji poti doslej obiskal. Kontrolna območja si mislimo oštevilčena od 1 do k. Če avtomobil ni obiskal še nobenega kontrolnega območja, predstavimo to s $c = 0$; če pa je obiskal že vse in je po tistem že tudi enkrat ponovno obiskal štartno črto ali pa prvo kontrolno območje, predstavimo to s $c = k + 1$.

Če poznamo neko stanje, ni težko našteti njegovih naslednikov, torej stanj, do katerih se da priti iz njega v enem časovnem koraku. Izbrati si moramo „pospešek“ v vsaki smeri, torej a_x in a_y iz $\{-1, 0, 1\}$. Novo stanje je potem (x', y', v'_x, v'_y, c') za $v'_x = v_x + a_x$, $v'_y = v_y + a_y$, $x' = x + v'_x$, $y' = y + v'_y$. Vrednost c' določimo takole: če je (x', y') na kontrolnem območju $c+1$, vzamemo $c' = c+1$; če je $c = k$ in je (x', y')

na štartni črti ali pa na kontrolnem območju 1, vzamemo $c' = c + 1$; če ne velja nič od tega dvojega, pač pa je (x', y') na enem od kontrolnih območij $1, 2, \ldots, c-1$, je premik neveljaven; sicer pa vzamemo $c' = c$. Premik je seveda neveljaven tudi v primeru, če je (x', y') neprevozno polje ali pa leži zunaj meja dirkališča. Vsako stanje ima tako največ devet naslednikov (ker imamo tri možnosti za a_x in tri za a_y). Za stanje $u = (x, y, v_x, v_y, c)$ označimo množico njegovih naslednikov z $N(u)$.

Naša pot se sme začeti v poljubnem stanju oblike $(x, y, 0, 0, 0)$, za katerega (x, y) leži na štartni črti; končati pa se mora v poljubnem izmed stanj oblike $(x, y, v_x, v_y, k+1)$. Pri tem naj ima pot čim manj korakov. Problema se lahko lotimo z iskanjem v širino:

$S := \{\}$; $Q :=$ prazna vrsta;
for each (x, y) s štartne črte:
  $u := (x, y, 0, 0, 0)$;
  $p[u] :=$ **nil**; dodaj u v Q in v S;
$t :=$ **nil**;
while $t =$ **nil and** Q ni prazna:
  vzemi iz vrste naslednje stanje, recimo u;
  for each $v \in N(u)$:
    if $v \notin S$:
      $p[v] := u$; dodaj v v Q in v S;
      if $v.c = k + 1$ **then** $t := v$;
if $t =$ **nil**:
  dirkališča se ne da obvoziti;
else:
  $v := t$; $\pi :=$ seznam z elementom v;
  while $p[v] \neq$ **nil**:
    $v := p[v]$; dodaj v na začetek seznama π;
  izpiši pot π;

Naš postopek si torej med iskanjem za vsako stanje v zapomni — v tabeli $p[v]$ — iz katerega stanja je vanj prišel. Ustavi se, čim pride do nekega končnega stanja; to stanje si zapomni v spremenljivki t in nato s pomočjo tabele p rekonstruira celoten potek poti. Načeloma se lahko zgodi tudi, da dirkališča sploh ni mogoče obvoziti (in pri tem upoštevati vse zahteve iz besedila naloge) — na primer, lahko bi se zgodilo, da bi bile v dirkališču vrzeli, ki bi jih morali preskočiti, vendar nimamo dovolj prostora, da bi vzeli zalet (in pridobili dovolj veliko hitrost za skok). V takih primerih primerne poti sploh ni in postopek se ustavi, ko se vrsta Q izprazni. Vendar pa med našimi testnimi primeri ni nobenega takega, pri katerem se dirkališča ne bi dalo obvoziti.

Ko se lotimo implementacije tega postopka v praksi, nam lahko povročata preglavice predvsem množica S (v kateri so vsa stanja, ki smo jih že kdaj zagledali in dodali v vrsto) ter tabela p (v kateri je za vsako stanje iz S napisano, iz katerega predhodnika smo v to stanje prišli). Obe skupaj bi lahko predstavili z razpršeno tabelo (*hash table*); težava pa je, da moramo pri nekaterih dirkališčih obiskati veliko stanj in bi takšna razpršena tabela zasedla preveč prostora. Težave nam bo povzročal predvsem deseti testni primer: tu imamo dirkališče velikosti 1000×1000, od tega je 326 422 polj tudi prevoznih, kontrolnih območij pa je 16. Izkaže se, da moramo pri iskanju v širino tu preiskati okoli 1,77 milijarde stanj, preden pridemo do najbližjega

ciljnega stanja. V razpršeni tabeli bi, tudi če zelo varčujemo, za vsako od teh stanj verjetno porabili v povprečju vsaj dobrih 5 bytov, torej bi taka tabela zasedla več kot 9 GB pomnilnika. To za današnje razmere sicer ni več nepredstavljivo veliko, je pa vseeno koristno razmisliti o kakšni varčnejši rešitvi.

Shranjevanje podatkov o predhodnikih. Tabele p ni nujno treba hraniti v pomnilniku; lahko jo sproti odlagamo na disk. Namesto prireditve $p[v] := u$ zapišimo na disk par (v, u). Zanko, s katero na koncu postopka rekonstruiramo pot π do najbližjega končnega stanja, moramo zdaj malo predelati, da bo brala pare (v, u) z diska. Ker smo stanja odkrivali (in zapisovali podatke o predhodnikih na disk) po naraščajoči oddaljenosti od začetnih stanj, pri rekonstrukciji poti pa rekonstruiramo pot od konca proti začetku, moramo tudi datoteko s podatki o predhodnikih zdaj brati od konca proti začetku.

$v := t$; $\pi :=$ seznam z elementom v;
while $v.c \neq 0$:
  preberi z diska nov par (v', u') (še enkrat poudarimo,
    da moramo te pare brati od konca proti začetku);
  if $v' = v$:
    $v := u'$; dodaj v na začetek seznama π;

Če hočemo varčevati tudi s prostorom na disku, namesto para (v, u) shranimo le $(v, a_x, a_y, u.c)$, pri čemer je $a_x = v.v_x - u.v_x$ in $a_y = v.v_y - u.v_y$ — to je dovolj, da izračunamo hitrost in položaj v stanju u. Lepo pri tem je, da so za a_x in a_y le po tri možnosti (namreč -1, 0 in 1), za $u.c$ pa $k + 2$ možnosti (od 0 do $k + 1$), zato lahko tudi pri 16 kontrolnih območjih vsa tri polja, a_x, a_y in $u.c$, predstavimo že s samo osmimi biti.

Varčnejša predstavitev množice že videnih stanj. Ostane nam še vprašanje, kako predstaviti množico S. Za začetek opazimo, da vseh možnih stanj ni ravno neomejeno veliko — za koordinati (x, y) je le toliko možnosti, kolikor je prevoznih polj na dirkališču; za c so možne vrednosti od 0 do $k + 1$; maksimalna hitrost pa je tudi omejena z velikostjo (in obliko) dirkališča, saj nimamo neomejeno veliko prostora za pospeševanje. Če začnemo v neki točki s hitrostjo $v_x = 0$ in ves čas pospešujemo v smeri x, bomo po T korakih dosegli hitrost $v_x = T$, naša x-koordinata pa se bo v tem času povečala za $1 + 2 + \ldots + T = T(T + 1)/2$. Ker je širina dirkališča le w enot, je hitrost $v_x = T$ dosegljiva samo, če je $T(T + 1)/2 < w$. Tako lahko že na samem začetku preprosto poiščemo neko zgornjo mejo za največjo možno hitrost v smeri x — recimo ji $v_x^{\max}$. Podobno storimo tudi za gibanje v smeri y in odslej vemo, da se bodo v množici S pojavljala le stanja, ki imajo $|v_x| \leq v_x^{\max}$ in $|v_y| \leq v_y^{\max}$. Pri dirkališču velikosti 1000×1000 to na primer pomeni, da gredo lahko hitrosti od -44 do $+44$. Število vseh možnih stanj je torej

$$(\text{št. prevoznih polj}) \times (2v_x^{\max} + 1) \times (2v_y^{\max} + 1) \times (k + 2).$$

Pri desetem testnem primeru, ki ima 326 422 prevoznih polj, dirkališče velikosti 1000×1000 in $k = 16$ kontrolnih območij, dobimo približno 46,5 milijard možnih stanj. Množico S lahko torej predstavimo s tabelo 46,5 milijard bitov, v kateri bo

vsak bit za eno od možnih stanj povedal, ali je tisto stanje prisotno v množici S ali ne. Takšna tabela bi zasedla približno 5,5 GB pomnilnika; če imamo računalnik z 8 GB pomnilnika, je to že čisto sprejemljivo.

Lahko pa gremo še korak naprej. Zgornji meji za hitrost avtomobila, $v_x^{\max}$ in $v_y^{\max}$, kakršni smo izračunali zgoraj, sta pogosto precej ohlapni: izračunali smo ju le na podlagi velikosti dirkališča, v resnici pa nam lahko oblika dirkališča (npr. zaradi ovinkov) preprečuje, da bi tako visoke hitrosti res dosegli. Množica S zato vsebuje le majhen delež vseh možnih stanj; za deseti testni primer smo na primer zgoraj videli, da S vsebuje 1,77 milijarde stanj, v prejšnjem odstavku pa smo ugotovili, da je možnih stanj tam kar 46,5 milijard.

Razdelimo množico S v mislih na nekaj disjunktnih podmnožic: $S_{v_x,v_y,c}$ naj vsebuje tista stanja iz S, ki imajo hitrost (v_x, v_y) in zadnje obiskano kontrolno območje c. Vsako od teh podmnožic lahko predstavimo s tabelo bitov, ki ima po en bit za vsako prevozno polje (x, y). Prihranek prostora dosežemo tako, da opazimo, da so mnoge od teh podmnožic čisto prazne, in zanje raje sploh ne alociramo tabele bitov, pač pa si pri vsaki od njih le zapomnimo, da je prazna. Pri desetem testnem primeru je na primer od $(2 \cdot 44 + 1) \times (2 \cdot 44 + 1) \times (16 + 2) = 142\,578$ podmnožic le 45 562 nepraznih; če za vsako od njih porabimo 326 422 bitov (toliko je namreč pri tem testnem primeru prevoznih polj), je skupna poraba pomnilnika le še 1,73 GB.

Iskanje v širino za vsako kontrolno območje posebej. Iskanje v širino obiskuje stanja po naraščajoči oddaljenosti od začetnih stanj: najprej obišče začetna stanja sama; nato obišče vsa stanja, ki so od začetnih oddaljena en korak; nato vsa stanja, ki so od začetnih oddaljena dva koraka; itd. Da nam bo nadaljnje razmišljanje lažje, preoblikujmo naš dosedanji postopek tako, da se bo to preiskovanje po naraščajočih oddaljenostih videlo bolj eksplicitno:

```
S := {}; Q := prazna vrsta;
for each (x, y) s štartne črte:
  u := (x, y, 0, 0, 0);
  dodaj u v Q in v S;
t := nil; d := 0;
while t = nil and Q ni prazna:
  Q' := prazna vrsta;
  P_{d+1} := nova datoteka (zaenkrat prazna);
  for each u ∈ Q:
    for each v ∈ N(u):
      if v ∉ S:
        dodaj v v Q' in v S;
        vpiši (v, u) v P_{d+1};
        if v.c = k + 1 then t := v;
  Q := Q';
  d := d + 1;
```

Namesto ene datoteke s podatki o predhodnikih imamo zdaj po eno datoteko za vsako možno oddaljenost d od začetnih stanj. Rekonstrukcija poti do stanja t poteka podobno kot prej:

```
v := t; π := seznam z elementom v;
while d > 0:
  preberi datoteko P_d; v njej se nahaja tudi podatek o
    v-jevem predhodniku, torej nek par (v, u);
  v := u; dodaj v na začetek seznama π;
  d := d − 1;
```

Za zdaj še nismo pridobili nobenega dodatnega prihranka v porabi pomnilnika; do tega lahko pridemo tako, da opazimo, da vzdolž vsake veljavne poti vrednost polja c v stanjih na poti zgolj narašča ali pa ostaja enaka, ne more pa se zmanjšati. Zato lahko najprej poiščemo najkrajše poti do vseh stanj s $c = 0$, nato najkrajše poti do vseh stanj s $c = 1$ in tako naprej; tak postopek bo še vedno vračal pravilne rezultate. Pri tem pa nam nikoli ne bo treba hraniti v pomnilniku cele množice S, ampak le množico tistih stanj iz S, ki imajo takšen c, s kakršnim se trenutno ukvarjamo. Naj bo torej S_c množica tistih stanj iz S, ki imajo zadnje obiskano kontrolno območje c. Naš postopek je zdaj tak:

```
S_0 := {}; Q := prazna vrsta;
for each (x, y) s štartne črte:
  u := (x, y, 0, 0, 0);
  dodaj u v Q in v S_0;
t := nil;
for c := 0 to k + 1:
  d := 0;
  if c > 0: S_c := {}; Q := prazna vrsta;
  while t = nil and (Q ni prazna or P_d ni prazna):
    if d > 0 then for each (u, u') ∈ P_d:
      if u.c = c − 1 then dodaj u v Q;
    Q' := prazna vrsta;
    če datoteka P_{d+1} še ne obstaja, ustvari novo prazno datoteko P_{d+1};
    for each u ∈ Q:
      if u.c = c or u.c = c − 1:
        for each v ∈ N(u):
          if v.c = c and v ∉ S_c:
            dodaj v v Q' in v S_c;
            vpiši (v, u) v P_{d+1};
            if v.c = k + 1 then t := v;
    Q := Q'; d := d + 1;
  pozabi množico S_c;
```

Tako lahko načeloma prihranimo nekaj pomnilnika, ker bo treba hraniti v njem le po eno množico S_c naenkrat, ne pa cele S. Za predstavitev množice S_c lahko uporabimo enak pristop kot zgoraj za S: razbijemo jo na podmnožice $S_{v_x,v_y,c}$ in predstavimo z bitnimi kartami le tiste med njimi, ki so neprazne. Če si na primer ogledamo deseti testni primer in njegovo S razbijemo na posamezne S_c, vidimo, da bi za največjo izmed njih (izkaže se, da je to S_1) potrebovali le še 138 MB pomnilnika.

No, v resnici ima ta naš razmislek pomanjkljivost: naš prvotni postopek z iskanjem v širino se je ustavil, čim je našel neko končno stanje; če je na primer najkrajša

pot od kakšnega začetnega stanja do najbližjega končnega stanja dolga d korakov, bi naš prvotni postopek obiskal vsa stanja, ki so dosegljiva iz kakšnega začetnega stanja v d ali manj korakih, ne bi pa obiskal nobenega takega stanja, ki je dosegljivo le v več kot d korakih. Naš novi postopek pa bi na primer pri $c = 0$ obiskal *vsa* stanja, ki imajo $c = 0$ in so dosegljiva iz kakšnega od začetnih stanj, ne glede na to, koliko korakov potrebujemo, da pridemo do njih. Podobno bi se zgodilo pri $c = 1$ in tako naprej. Zato so množice S_c, s katerimi imamo tu opravka, lahko precej večje od tistih, ki bi jih dobili, če bi vzeli množico S iz našega prvotnega postopka in jo razbili na posamezne S_c. Prihranek pomnilnika zato ne bo tolikšen, kot smo razmišljali v prejšnjem odstavku. Vendar bi tudi v najslabšem primeru vendarle prihranili kar precej pomnilnika; tudi če bi na primer neka S_c vsebovala toliko stanj, da bi morali predstaviti z bitnimi kartami vse njene podmnožice $S_{v_x,v_y,c}$ (torej za vse možne v_x in v_y), bi nam to zdaj vzelo le (št. prevoznih polj) $\times (2v_x^{\max} + 1) \times (2v_x^{\max} + 1)$ bitov, kar pri desetem testnem primeru znese približno 308 MB — še vedno lep prihranek v primerjavi z 1,73 GB iz prejšnjega razdelka.

Če bi hoteli porabo pomnilnika za posamezne S_c še omejiti, bi si lahko vnaprej postavili zgornjo mejo za d in se pri iskanju v širino vsakič ustavili, čim bi d dosegel to zgornjo mejo. Če si postavimo prenizko mejo, se seveda lahko zgodi, da ne bomo našli nobenega končnega stanja; v tem primeru mejo povečajmo in postopek ponovimo. (Mimogrede, ta prijem pri preiskovanju prostorov stanj se imenuje „iterativno poglabljanje“ — *iterative deepening*.) Če hočemo zagotoviti, da ne bomo nikoli obiskali kakšnega stanja, ki bi bilo od začetnih stanj oddaljeno dlje kot najbližje končno stanje, lahko na začetku postavimo mejo na 1 in jo nato povečujemo po 1; seveda pa bi se pri tem močno povečala časovna zahtevnost postopka (ker moramo vedno znova in znova pregledovati vse večji del prostora stanj).

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 107–113.

Letnik: ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Kako si izvedel za tekmovanje?
☐ od mentorja ☐ na spletni strani (kateri? _____)
☐ od prijatelja/sošolca ☐ drugače (kako? _____)
Kolikokrat si se že udeležil kakšnega tekmovanja iz računalništva pred tem tekmovanjem? _____
Katerega leta si se udeležil prvega tekmovanja iz računalništva? _____
Najboljša dosedanja uvrstitev na tekmovanjih iz računalništva (kje in kdaj)? _____

Koliko časa že programiraš? _____
Kje si se naučil? ☐ sam ☐ v šoli pri pouku ☐ na krožkih ☐ na tečajih ☐ poletna šola
☐ drugje: _____

Za programske jezike, ki jih obvladaš, napiši (začni s tistimi, ki jih obvladaš najbolje):

Jezik: _____
Koliko programov si že napisal v tem jeziku: ☐ do 10 ☐ od 11 do 50 ☐ nad 50
Dolžina najdaljšega programa v tem jeziku:
☐ do 20 vrstic ☐ od 21 do 100 vrstic ☐ nad 100
[Gornje rubrike za opis izkušenj v posameznem programskem jeziku so se nato še dvakrat ponovile, tako da lahko reševalec opiše do tri jezike.]

Ali si programiral še v katerem programskem jeziku poleg zgoraj navedenih? V katerih?

Kako vpliva tvoje znanje matematike na programiranje in učenje računalništva?
☐ zadošča mojim potrebam
☐ občutim pomanjkljivosti, a se znajdem
☐ je preskromno, da bi koristilo

Kako vpliva tvoje znanje angleščine na programiranje in učenje računalništva?
☐ zadošča mojim potrebam
☐ občutim pomanjkljivosti, a se znajdem
☐ je preskromno, da bi koristilo

Ali poznaš naslednje podatkovne strukture:

Drevo	☐ da	☐ ne
Hash tabela (asociativna tabela)	☐ da	☐ ne
S kazalci povezan seznam (linked list)	☐ da	☐ ne
Sklad (stack)	☐ da	☐ ne
Vrsta (queue)	☐ da	☐ ne

Ali poznaš naslednje algoritme:

Evklidov algoritem (za največji skupni delitelj)	☐ da	☐ ne
Eratostenovo rešeto (za iskanje praštevil)	☐ da	☐ ne
Poznaš formulo za vektorski produkt	☐ da	☐ ne
Rekurzivni sestop	☐ da	☐ ne
Dinamično programiranje	☐ da	☐ ne
Iskanje v širino (po grafu)	☐ da	☐ ne
Katerega od algoritmov za urejanje	☐ da	☐ ne

Katere(ga)? ☐ bubble sort (urejanje z mehurčki)
☐ insertion sort (urejanje z vstavljanjem)
☐ selection sort (urejanje z izbiranjem)
☐ quicksort
☐ kakšnega drugega: _____

[Le pri 1. in 2. skupini.] V besedilu nalog trenutno objavljamo deklaracije tipov in podprogramov v pascalu in C/C++.
— Ali razumeš kakšnega od teh jezikov dovolj dobro, da razumeš te deklaracije v besedilu naših nalog? ☐ da ☐ ne
— Ali bi raje videl, da bi objavljali deklaracije (tudi) v kakšnem drugem programskem jeziku? Če da, v katerem? _____

V rešitvah nalog trenutno objavljamo izvorno kodo v pascalu in C-ju.
— Ali razumeš kakšnega od teh jezikov dovolj dobro, da si lahko kaj pomagaš z izvorno kodo v naših rešitvah? ☐ da ☐ ne
— Ali bi raje videl, da bi izvorno kodo rešitev pisali v kakšnem drugem jeziku? Če da, v katerem? _____

[Le pri 3. skupini.] Doslej smo v tretji skupini podpirali reševanje nalog v pascalu, C, C++ in javi. Bi rad uporabljal kakšen drug programski jezik? Če da, katerega? _____

Katere od naslednjih jezikovnih konstruktov in programerskih prijemov znaš uporabljati?	ne poznam	da, slabo	da, dobro
Ali bi znal prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal prebrati kakšno celo število in kakšen niz iz datoteke ali pa ju zapisati v datoteko?	☐	☐	☐
Tabele (**array**):			
— enodimenzionalne	☐	☐	☐
— dvodimenzionalne	☐	☐	☐
— večdimenzionalne	☐	☐	☐
Znaš napisati svoj podprogram (**procedure**, **function**)	☐	☐	☐
Poznaš rekurzijo	☐	☐	☐
Kazalce, dinamično alokacijo pomnilnika (New/Dispose, GetMem/FreeMem, malloc/free, **new**/**delete**, ...)	☐	☐	☐
Zanka **for**	☐	☐	☐
Zanka **while**	☐	☐	☐
Gnezdenje zank (ena zanka znotraj druge)	☐	☐	☐
Naštevni tipi (*enumerated types* — **type** ImeTipa = (Ena, Dve, Tri) v pascalu, **typedef enum** v C/C++)	☐	☐	☐
Strukture (**record** v pascalu, **struct**/**class** v C/C++)	☐	☐	☐

and, **or**, **xor**, **not** kot aritmetični operatorji (nad biti celoštevilskih operandov namesto nad logičnimi vrednostmi tipa boolean) (v C/C++: `&`, `|`, `^`, `~`) ☐ ☐ ☐
Operatorja **shl** in **shr** (v C/C++: `<<`, `>>`) ☐ ☐ ☐

[Naslednja skupina vprašanj se je ponovila za vsako nalogo po enkrat.]
Zahtevnost naloge: ☐ prelahka ☐ lahka ☐ primerna ☐ težka ☐ pretežka ☐ ne vem
Naloga je (ali: bi) vzela preveč časa: ☐ da ☐ ne ☐ ne vem
Mnenje o besedilu naloge:
— dolžina besedila: ☐ prekratko ☐ primerno ☐ predolgo
— razumljivost besedila: ☐ razumljivo ☐ težko razumljivo ☐ nerazumljivo
Naloga je bila: ☐ zanimiva ☐ dolgočasna ☐ že znana ☐ povprečna
Si jo rešil?
- ☐ nisem rešil, ker mi je zmanjkalo časa za reševanje
- ☐ nisem rešil, ker mi je zmanjkalo volje za reševanje
- ☐ nisem rešil, ker mi je zmanjkalo znanja za reševanje
- ☐ rešil sem jo le delno, ker mi je zmanjkalo časa za reševanje
- ☐ rešil sem jo le delno, ker mi je zmanjkalo volje za reševanje
- ☐ rešil sem jo le delno, ker mi je zmanjkalo znanja za reševanje
- ☐ rešil sem celo

Ostali komentarji o tej nalogi: _____

Katera naloga ti je bila najbolj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

Katera naloga ti je bila najmanj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

Na letošnjem tekmovanju ste imeli tri ure / pet ur časa za pet nalog.
Bi imel raje: ☐ več časa ☐ manj časa ☐ časa je bilo ravno prav
Bi imel raje: ☐ več nalog ☐ manj nalog ☐ nalog je bilo ravno prav

Kakršne koli druge pripombe in predlogi. Kaj bi spremenil(a), popravil(a), odpravil(a), ipd., da bi postalo tekmovanje zanimivejše in bolj privlačno? _____

Kaj ti je bilo pri tekmovanju všeč? _____

Kaj te je najbolj motilo? _____

Če imaš kaj vrstnikov, ki se tudi zanimajo za programiranje, pa se tega tekmovanja niso udeležili, kaj bi bilo po tvojem mnenju treba spremeniti, da bi jih prepričali k udeležbi?

Poleg tekmovanja bi radi tudi v preostalem delu leta organizirali razne aktivnosti, ki bi vas zanimale, spodbujale in usmerjale pri odkrivanju računalništva. Prosimo, da nam pomagate izbrati aktivnosti, ki vas zanimajo in bi se jih zelo verjetno udeležili.

Udeležil bi se oz. z veseljem bi spremljal:

- ☐ izlet v kak raziskovalni laboratorij v Evropi (po možnosti za dva dni)
- ☐ poletna šola računalništva (1 teden na IJS, spanje v dijaškem domu)
- ☐ poletna praksa na IJS
- ☐ predstavitve novih tehnologij (.NET, mobilni portali, programiranje „vgrajenih računalnikov“, strojno učenje, itd.) (1× mesečno)
- ☐ predavanja o algoritmih in drugih temah, ki pridejo prav na tekmovanju (1× mesečno)
- ☐ reševanje tekmovalnih nalog (naloge se rešuje doma in bi bile delno povezane s temo, predstavljeno na predavanju; rešitve se preveri na strežniku) (1× mesečno)
- ☐ tvoji predlogi: _____

Vesel bi bil pomoči pri:

- ☐ iskanju štipendije
- ☐ iskanju podjetij, ki dijakom ponujajo njim prilagojene poletne prakse in druge projekte, kjer se ob mentorstvu lahko veliko naučijo.

Ali si pri izpolnjevanju ankete prišel do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 22 tekmovalcev prve skupine, 31 tekmovalcev druge in 23 tekmovalcev tretje. Ker se marsikomu ne da pisati odgovorov na vprašanja, smo pri letošnji anketi (še bolj kot lani) dali večji poudarek na vprašanjih z vnaprej ponujenimi odgovori, pri katerih je treba le odkljukati okence pred želenim odgovorom.

Mnenje tekmovalcev o nalogah

Tekmovalce smo spraševali: kako zahtevna se jim zdi posamezna naloga; ali se jim zdi, da jim vzame preveč časa; ali je besedilo primerno dolgo in razumljivo; ali se jim zdi naloga zanimiva; ali so jo rešili (oz. zakaj ne); in katera naloga jim je bila najbolj/najmanj všeč.

Rezultate vprašanj o zahtevnosti nalog kažejo grafi na str. 108. Tam so tudi podatki o povprečnem številu točk, doseženem pri posamezni nalogi, tako da lahko primerjamo mnenje tekmovalcev o zahtevnosti naloge in to, kako dobro so jo zares reševali.

V povprečju se je zdela 3. skupina ljudem težja kot prva in druga. Če pri vsaki nalogi pogledamo povprečje mnenj o zahtevnosti te naloge (1 = prelahka, 3 = primerna, 5 = pretežka) in vzamemo povprečje tega po vseh petih nalogah, dobimo: 3,18 v prvi skupini, 3,36 v drugi in 3,72 v tretji.

Ni pa videti kakšne opazne korelacije med tem, kako težka se je naloga zdela tekmovalcem, in tem, kako dobro so jo zares reševali (npr. merjeno s povprečnim številom točk pri tej nalogi).

Največ pripomb, češ da je težka (ali celo pretežka), je dobila naloga 3.1 (nebotičniki); to je po svoje nepričakovano, saj je ta naloga pravzaprav zelo šolski primer rekurzije, le-to pa tekmovalci v 3. skupini najbrž dobro poznajo.

Da bi bila kakšna naloga prelahka, se ne pritožujejo; še največ takih glasov so dobile ceste (1.3), ki so pravzaprav res lahka naloga (zanimivo pa je, da so se marsikomu vendarle zdele težke).

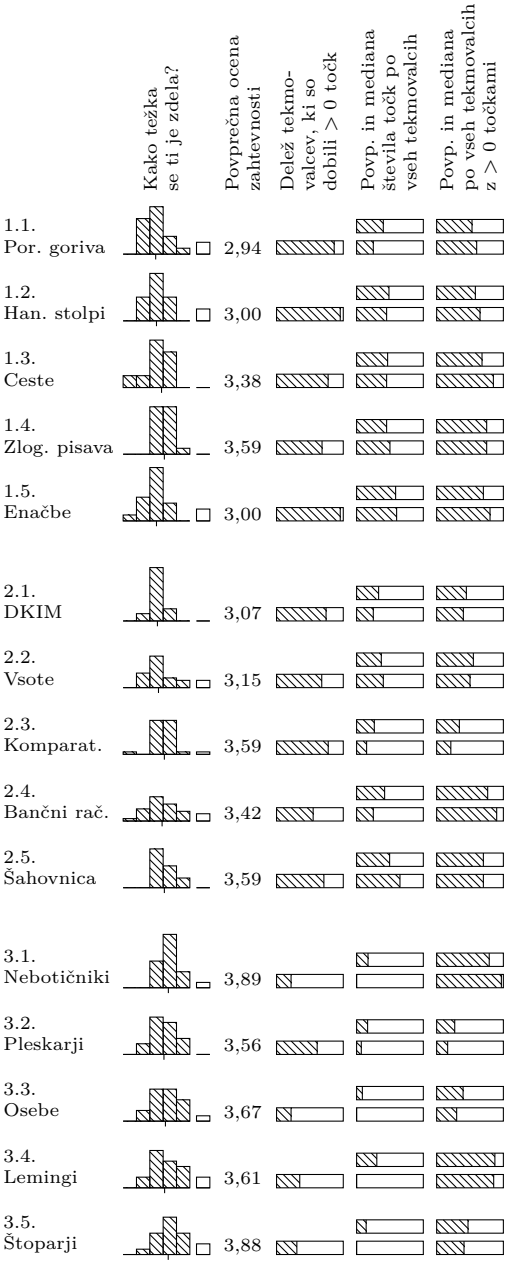
Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 109. Nad razumljivostjo besedil ni veliko pripomb, razen pri nalogi 2.2 (vsote), ki se je večini ljudi zdela težko razumljiva ali celo nerazumljiva. Nekaj pripomb glede razumljivosti je tudi pri lemingih (3.4) in štoparjih (3.5).

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, le pri vsotah (2.2) je nekaj več pripomb, češ da je besedilo prekratko. Res je, da je znatno krajše kot pri večini drugih nalog.

Naloge se jim večinoma zdijo zanimive; še največ pripomb, češ da je naloga dolgočasna, je bilo pri vsotah (2.2) in lemingih (3.4).

Pripomb, da bi naloga vzela preveč časa, večinoma ni bilo veliko; največ jih je bilo pri komparatorjih (2.3) in pogrešanih osebah (3.3); če pogledamo besedilo teh dveh nalog, vidimo, da sta verjetno res med najbolj zamudnimi na tem tekmovanju. Bolj presenetljivo je, da so se prezamudni marsikomu zdeli tudi nebotičniki (3.1).

Glasovi o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, so precej razpršeni med naloge; skoraj vsaka je bila nekaterim najbolj in nekaterim najmanj všeč. To je po svoje mogoče dobro, saj je znak, da se najde v naših nalogah za vsakogar nekaj. Primer izrazito priljubljenih nalog so bančni računi (2.4) in pleskarji (3.2), primer izrazito nepriljubljene pa komparatorji (2.3); bančni računi



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

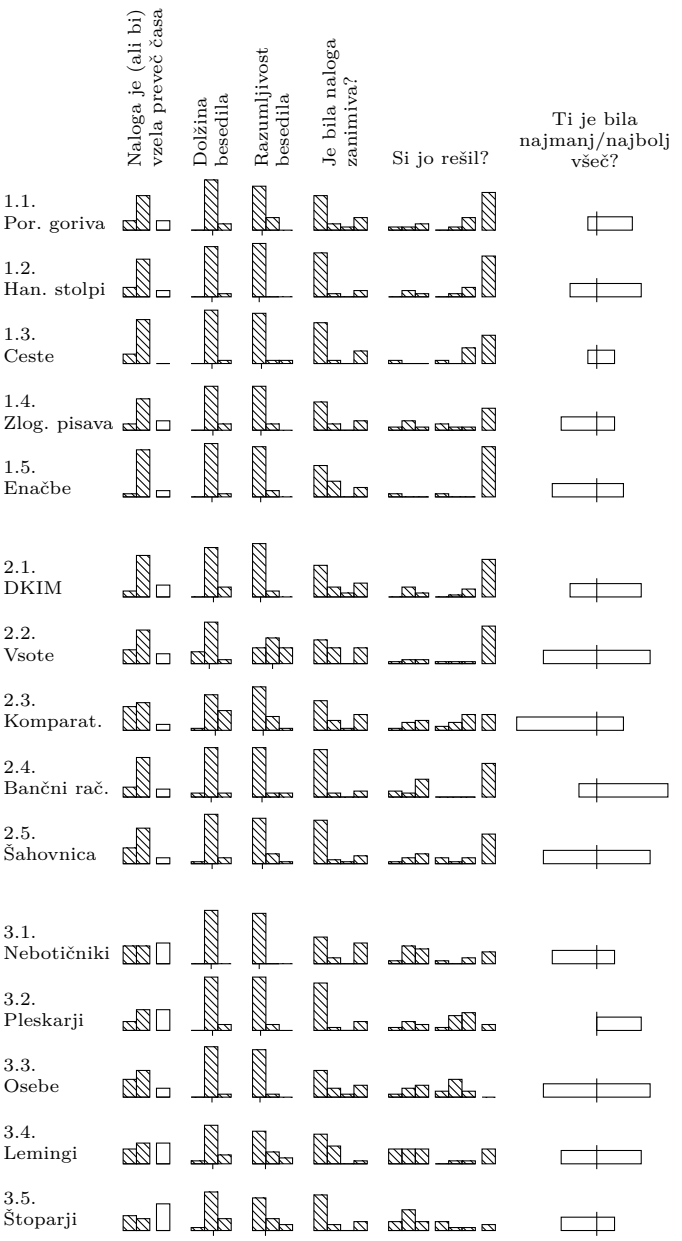
Pomen stolpcev v vsaki vrstici:

Na levi je skupina šestih stolpcev, ki kažejo, kako so tekmovalci v anketi odgovarjali na vprašanje o zahtevnosti naloge. Stolpci po vrsti pomenijo odgovore „prelahka“, „lahka“, „primerna“, „težka“, „pretežka“ in „ne vem“. Višina stolpca pove, koliko tekmovalcev je izrazilo takšno mnenje o zahtevnosti naloge. Desno od teh stolpcev je povprečna ocena zahtevnosti (1 = prelahka, 3 = primerna, 5 = pretežka). Povprečno oceno kaže tudi črtica pod to skupino stolpcev.

Sledi stolpec, ki pokaže, kolikšen delež tekmovalcev je pri tej nalogi dobil več kot 0 točk. Naslednji par stolpcev pokaže povprečje (zgornji stolpec) in mediano (spodnji stolpec) števila točk pri vsej nalogi. Zadnji par stolpcev pa kaže povprečje in mediano števila točk, gledano le pri tistih tekmovalcih, ki so dobili pri tisti nalogi več kot nič točk.

Mnenje tekmovalcev o nalogah

Višina stolpcev pove, koliko tekmovalcev je dalo določen odgovor na neko vprašanje.



Stolpci se od leve proti desni nanašajo na naslednja vprašanja in možne odgovore:

Naloga je (ali: bi) vzela preveč časa:

- da
- ne
- ne vem

Dolžina besedila:

- prekratko
- primerno
- predolgo

Razumljivost besedila:

- razumljivo
- težko razumljivo
- nerazumljivo

Naloga je bila:

- zanimiva
- dolgočasna
- že znana
- povprečna

Si jo rešil?

- nisem, zmanjkalo časa
- nisem, zmanjkalo volje
- nisem, zmanjkalo znanja
- delno, zmanjkalo časa
- delno, zmanjkalo volje
- delno, zmanjkalo znanja
- rešil sem celo

Katera naloga ti je bila najmanj všeč? Katera ti je bila najbolj všeč?

in komparatorji sta vsaka po svoje malo drugačni od ostalih nalog, komparatorji verjetno celo toliko (in na tak način), da mnogim preprosto niso bili všeč že zaradi tega. Zanimivo je tudi, da so bile nekatere naloge hkrati mnogim najljubše in mnogim drugim najmanj všeč; primeri so vsote (2.2), šahovnica (2.5) in pogrešane osebe (3.3).

Programersko znanje

Ko sestavljamo naloge, še posebej tiste za prvo skupino, nas pogosto skrbi, če tekmovalci poznajo ta ali oni jezikovni konstrukt ali programerski prijem. Zato smo jih že lani v anketi za nekaj stvari vprašali, če jih poznajo in bi jih znali uporabiti v svojih programih. Ta vprašanja smo letos še razširili z vprašanji o nekaterih znanih algoritmih in podatkovnih strukturah.

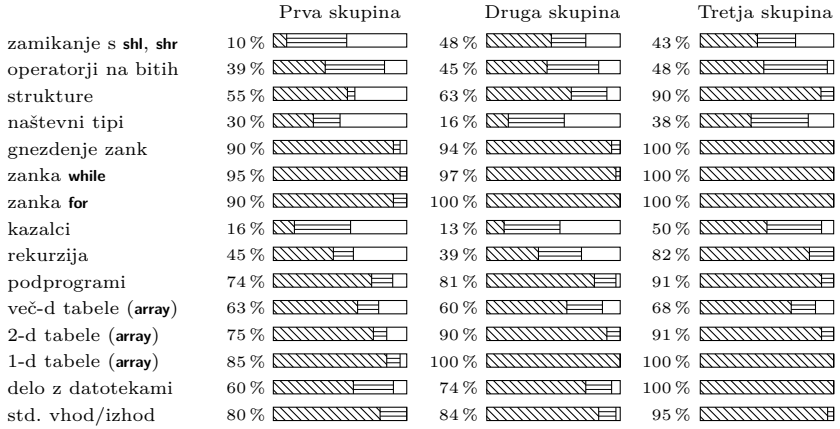


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Že lani smo bili rahlo presenečeni nad tem, kako veliko ljudi pravi, da določene prijeme poznajo. Letos smo zato poskušali namesto dveh možnih odgovorov ponuditi tri („ne poznam“, „poznam, slabo“ in „poznam, dobro“), vendar se slika zaradi tega ni kaj dosti spremenila. Stvari, ki jih poznajo slabše, so približno iste kot lani: rekurzija, kazalci, naštevni tipi in operatorji na bitih (nepoznavanje zadnjih dveh tu naštetih reči je še posebej izrazito, bolj kot lani).

Pri vprašanjih o algoritmih in podatkovnih strukturah prihaja do nekaj nepričakovanih rezultatov. Človek bi pričakoval, da če se premaknemo iz prve skupine v drugo in iz druge v tretjo, bi moralo poznavanje algoritmov in podatkovnih struktur naraščati. Pri nekaterih je to res, pri mnogih pa tudi ne. Primer tega slednjega je Evklidov algoritem; mogoče je to nekaj, česar se naučijo na začetku srednje šole in ga do časa, ko začnejo tekmovati v 3. skupini, že pozabijo? Še bolj presenetljivo je, da se nekaj podobnega zgodi tudi pri razpršenih tabelah, ki jih baje v 3. skupini pozna manj tekmovalcev kot v prvi.

Mogoče ni pametno, da smo vprašanje o algoritmih in podatkovnih strukturah zapisali kot „ali poznaš“ namesto npr. „ali znaš implementirati in uporabljati“. Naj-

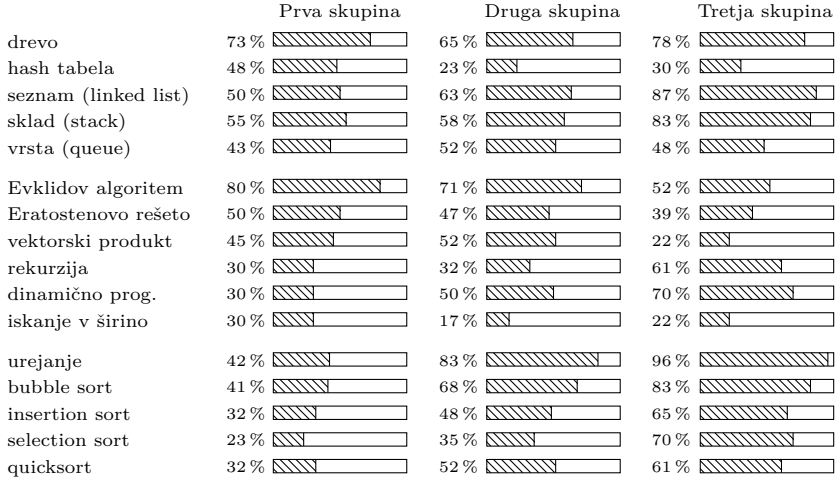


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

brž si lahko s tem razložimo npr. zelo visok delež tekmovalcev 1. skupine, ki pravijo, da poznajo drevo (v isti sapi pa, da ne poznajo seznamov in ne znajo delati s kazalci — to pa so stvari, ki ponavadi pridejo prav pri implementaciji drevesa).

Presenetljivo veliko tekmovalcev 3. skupine pravi, da poznajo dinamično programiranje (70 %); splošno prepričanje je drugače to, da dinamičnega programiranja samouki v srednješolskih letih ponavadi še ne poznajo; ne vemo pa, koliko ljudi se ima možnost srečati z njim v šoli. Kakorkoli že, prav možna razlaga te visoke številke je tudi ta, da so nas tekmovalci narobe razumeli in mislili, da sprašujemo, če poznajo dinamično delo s pomnilnikom (`malloc`, `free` ipd.).

Presenetljivo malo tekmovalcev 3. skupine pozna razpršene tabele, iskanje v širino in vektorski produkt. Slednje je škoda, ker bi prišel prav pri marsikateri geometrijski nalogi.

Spodbudno je, da skoraj vsi tekmovalci 3. skupine poznajo vsaj kakšnega od algoritmov za urejanje (ponavadi vsaj bubblesort).

Uporaba programskih jezikov

Večina tekmovalcev uporablja programske jezike pascal, C in C++. Lani smo prvič opazili, da je bil pascal v izraziti manjšini; ta trend se letos še nadaljuje in pomembno število uporabnikov pascala je le še v prvi skupini. To sicer ne pomeni, da pascala sploh ne poznajo; v anketi smo jih spraševali tudi, katere jezike znajo (in kako dobro) in jih dobra polovica obvlada tudi pascal, le da očitno C ali C++ poznajo še bolje in so zato na tekmovanju uporabljali enega od njiju. Veliko tekmovalcev pravi tudi, da znajo programirati v PHPju, ki v tem pogledu ne zaostaja veliko za pascalom; na tekmovanju pa ga je uporabljal en sam tekmovalec. Rahlo presenetljivo je, kako malo tekmovalcev zna javo in basic.

Letos se je prvič zgodilo, da je nekdo uporabljal C#. V tretji skupini tokrat ni nihče uporabljal jave (lani so jo trije).

	Leto in skupina									
	2007			2006			2004			2003
Jezik	1	2	3	1	2	3	1	2	3	3
pascal	$8\frac{1}{2}$	2	1	6	5	5	23	20	13	17
C	$5\frac{1}{2}$	11	$6\frac{1}{2}$	4	16	$1\frac{1}{2}$	13	$7\frac{1}{2}$	1	4
C++	7	14	$15\frac{1}{2}$	13	5	$10\frac{1}{2}$	5		6	5
java		$2\frac{1}{2}$				3		$\frac{1}{2}$		
PHP	1			1						
basic		1			1					
C#		$\frac{1}{2}$								
nič	3			1	2		3	2		

Število tekmovalcev, ki so uporabljali posamezni programski jezik.
Nekateri uporabljajo po dva različna jezika (pri različnih nalogah) in se štejejo polovično k vsakemu jeziku. „Nič“ pomeni, da tekmovalec ni napisal nič izvorne kode.

Glede štetja C in C++ v gornji tabeli je treba pripomniti, da je razlika med njima tako ali tako zelo majhna: tisti, ki delajo v C++, uporabljajo večinoma le malo stvari, ki jih C++ ima, C pa ne. To so ponavadi <iostream> in vhodno/izhodna tokova cin in cout namesto C-jevih funkcij scanf, printf in podobnih. Precej pogosto uporabljajo tudi razred string.

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu in C/C++. Tekmovalce smo v anketi vprašali, če te deklaracije razumejo ali pa bi morale biti še v kakšnem drugem jeziku; veliki večini sedanje deklaracije zadostujejo (17/20 v prvi skupini in 30/31 v drugi). Nekateri predlagajo še deklaracije v javi, C# in pythonu.

V rešitvah nalog objavljamo izvorno kodo v pascalu in C-ju. Tekmovalce vseh treh skupin smo v anketi vprašali, če kakšnega od teh dveh jezikov razumejo dovolj, da si lahko kaj pomagajo s to izvorno kodo, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Velika večina je s sedanjima jezikoma zadovoljna (18/20 v prvi skupini, 25/30 v drugi in 22/23 v tretji). 14 ljudi je predlagalo, da bi bile rešitve tudi v C++, osem pa v javi. Nekoliko presenetljivo je, da je pet ljudi napisalo, da sedanjih rešitev ne razumejo in da bi jih radi videli v C++; mar je mogoče, da nekdo, ki razume C++, ne razume naših sedanjih rešitev v Cju?

Rešitev v C *in* v C++ tudi v prihodnje najbrž ne bomo objavljali, lahko pa bi poskusili razmisliti (in najprej v anketi malo potipati), kako bi bilo, če bi namesto rešitev v C-ju objavljali rešitve v C++.

Letnik

Po pričakovanjih so tekmovalci zahtevnejših skupin v povprečju v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot lani. Največ tekmovalcev hodi v tretji ali četrti letnik; dijakov prvega letnika pa je izrazito malo, pa še ti so skoraj vsi v prvi skupini. Je pa zanimivo, da je v prvi skupini tudi nekaj dijakov 4. letnika.

Skupina	Št. tekmovalcev po letnikih				
	1	2	3	4	povprečje
prva	6	8	7	4	2,4
druga	1	5	16	9	3,1
tretja		1	10	12	3,5

Druga vprašanja

Podobno kot lani je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!). Ostali so večinoma izvedeli za tekmovanje prek spletnih strani, od tega jih malo več kot polovica navaja našo stran (`rtk.ijs.si`), drugi pa portal Slo-Tech (`www.slo-tech.com`), ki je prijazno objavil novico o našem tekmovanju in reklamno pasico s povezavo na našo spletno stran.

Tudi pri vprašanju, kje so se naučili programirati, so odgovori podobni kot lani. Večina je samoukov, precej pa je tudi takih, ki so se programirati naučili v šoli; v drugi skupini slednji celo prevladujejo nad samouki.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. V tretji in še zlasti drugi skupini je precej tudi takih, ki si želijo manj nalog (pri enakem času reševanja). Glede časa pa, kolikor si že želijo sprememb, si želijo več časa za reševanje nalog, ne manj.

	Kje si izvedel za tekmovanje				Kje si se naučil programirati					Čas reševanja			Število nalog			Potekmovalne dejavnosti								
Skupina	od mentorja	na spletni strani	od prijatelja/sošolca	drugače	sam	pri pouku	na krožkih	na tečajih	poletna šola	hočem več časa	hočem manj časa	je že v redu	hočem več nalog	hočem manj nalog	je že v redu	izlet v tuji laboratorij	poletna šola	praksa na IJS	predstavitve tehnologij	predavanja o algoritmih	reševanje nalog	iskanje štipendije	iskanje podjetij	
I	19	1	1	2	15	7	3	2	2	2	0	15	1	4	10	6	7	10	11	8	10	6	6	
II	27	4	2	1	17	20	3	1	2	5	4	20	1	11	17	10	9	13	12	9	8	9	15	
III	16	5	1	2	17	10	4			3	4	9	3	9	5	5	5	6	9	10	7	6	8	

Iz odgovorov na vprašanje, kakšne potekmovalne dejavnosti bi jih zanimale, je težko zaključiti kaj posebej konkretnega. Večina je odkljukala kar vse možnosti ali pa nobene (ker npr. tistega dela ankete sploh niso izpolnjevali).

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb.

CVETKE

V tem razdelku je zbranih nekaj zabavnih odlomkov iz rešitev, ki so jih napisali tekmovalci. V oklepajih pred vsakim odlomkom sta skupina in številka naloge.

(1.1) Nek tekmovalec je napisal rešitev v pascalu in uporabil deset ločenih spremenljivk namesto tabele z desetimi elementi. Koda je temu primerno mazohistična.

```
s1 := s2; s2 := s3; s3 := s4; s4 := s5; s5 := s6; s6 := s7; s7 := s8; s8 := s9; s9 := s10;
                                                                        s10 := pot;
```

(1.2) Nek tekmovalec je ignoriral dejstvo, da je število ploščic lahko tudi različno od 5, in si zato privoščil mazohistične pogoje v stilu:

```
if (h[0] > h[1] > h[2] > h[3] > h[4] &
    h[5] > h[6] > h[7] > h[8] > h[9] &
    h[10] > h[11] > h[12] > h[13] > h[14) { ... }
```

(1.3) Naslednja rešitev je pravzaprav konceptualno dobro zastavljena, le zankam se izogiba precej bolj, kot je v pascalu to mogoče:

```
program Ceste;
type tabela1000 = array [1..1000] of integer;
var km: tabela1000;
  x
begin
  i := 1;
  for x := 1 to StCest do
    km[1..(DolzinaCeste(i) − 1] = km[1..(DolzinaCeste(i) − 1] + 1;
  WriteLn(km);
end.
```

(1.4) Nek tekmovalec je raje naštel soglasnike kot samoglasnike:

```
if (t0 == 'b' | 'c' | 'd' | 'f' | 'g' | 'h' | 'j' | 'k' | 'l' | 'm' | 'n' |
                                    'p' | 'r' | 's' | 't' | 'v' | 'z') { } else { p++; };
```

(1.4) Nek drug tekmovalec je za preverjanje, ali je nek znak sploh črka, napisal tole:

```
if Znak in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
                                  'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'] then
```

Najbrž ni vedel, da lahko v pascalu to množico zapišemo tudi kot `['a'..'z']`. Zanimivo je tudi to, da čeprav množice očitno pozna, jih v tej isti rešitvi, ko je bilo treba preverjati, ali je neka črka samoglasnik, ni uporabil, ampak si je definiral takšen niz:

```
                { abcdefghijklmnopqrstuvwxyz }
const Tabela = '10001000100000100000100000'
```

(1.4) Pomlad tudi črkam ne prizanaša...

če sta soglasnik, soglasnik
– vrinemo mednju samoglasnik
– spet poparčkamo

(1.*) En tekmovalec iz prve skupine je izpolnil le obrazec z osebnimi podatki, vsi drugi papirji, ki jih je oddal, pa so bili popolnoma prazni — na njih ni bilo napisano nič, kot da ni niti poskušal rešiti nobene naloge...

(2.1) Nekatere rešitve po nepotrebnem preberejo celotno besedilo naenkrat v pomnilnik:

```
/* glede na to, koliko spremenljivk je uporabljenih, bo poraba spomina
dokaj velika, ampak ker imajo dandanes vsi računalniki vsaj 128 MB
spomina, to naj ne bi predstavljalo problema */
```

To je po svoje res, bolj nerodno je to, da njegova rešitev uporablja tabele 1000×1000 znakov in se tako omeji na besedila, dolga največ 1000 vrstic (s po največ 1000 znaki v vsaki vrstici — naloga sicer zagotavlja celo to, da jih bo največ 200 v vsaki vrstici).

(2.1) Tretji **if** je zelo koristen:

```
if (isspace(vrstica[u]))
  ...
else if (b == 2)
  ...
else if ((isspace(vrstica[u])) && (vrstica[u] == '\n'))
  ...
```

(2.2) Cela rešitev te naloge:

> Minister bi po opravljenem seznamu klube v prvi tabeli razvrstil po uspešnosti, zatem pa po številu članov. Enako bi storil v drugi tabeli. Sledijo poizvedbe, ki bi ustrezale število članov za n sedežev. Uvedli bi lahko tudi minimalno število let, kolikor klub pravzaprav obstaja.

(2.2) Cela rešitev:

> Na potovanje ni možno poslati klubov z različnih seznamov, saj se marajo le klubi na istem seznamu.

(2.2) Nek tekmovalec je že napisal zametek precej pravilne rešitve (le malo neučinkovita bi bila), nato pa vse skupaj prečrtal in napisal novo, ki temelji na popolnoma napačnem razumevanju naloge:

> Rešitev se da preprosto izračunati... če je vsota vseh članov iz klubov na 1. seznamu plus vsota vseh članov iz klubov na drugem seznamu deljivo s številom sedežev v oklepnikih in je ta rezultat enak številu klubov 1. seznama, lahko minister pošlje po 1 klub iz vsakega seznama.
>
> Recimo, da je v vsakem klubu 5 igralcev in na vsakem od seznamov imamo po 3 klube. Oklepniki pa imajo 10 sedežev.
>
> $$\frac{5 \cdot 3 + 5 \cdot 3}{10} = 3$$
> $$\frac{15 + 15}{10} = 3$$
> $$\underline{\underline{\frac{30}{10} = 3}}$$

(2.2) Cela rešitev te naloge:

> Ker podatkov o uspešnosti klubov nimamo, bo težko oz. nemgoče na izlet poslati dva najuspešnejša. Pravzaprav tukaj uspešnostni faktor nima vloge (ker nimamo podatkov o uspešnosti). Ker pač že denar je `:)` bi morali najti par; po en klub iz vsake strani; katerih dveh število članov bi bilo natanko n. Glede na to, da je število klubov vse večje, ni vrag, da ne bi našli VSAJ enega para, ki bi ustrezal.

(2.2) Hvalevredna samokritičnost tekmovalca do svoje rešitve, ki pregleda vse možne pare klubov:

> PS. Moja rešitev je pri velikih seznamih klubov zelo počasna [...] Dobro, da imamo core 2 duo procesorje, da je stvar malo hitrejša.

(2.3) Cela rešitev podnaloge (a):

```
char *Kljuc(char *Z)
{
  int x;
  char *K = (char *) strcmp(Z + 1, Z);
```

(2.3) Kako pri podnalogi (a) vsako črko niza spremeniti v zaporedno številko te črke znotraj abecede?

```
for (int a = 0; a < dolzina; a++)    // preveri prvo besedo
{
  for (int b = 0; b < 28; b++)       // številka 28 bi naj bila
                                     // število črk v ang. abecedi[15]
  {
    if (Z[a] == char(b + 63))        // b + 63 v ASCII kodi veliki ali mali A,
                                     // s katerim bi primerjal (ne vem, če je točno 63)[16]
    {
      spomin[a] = b;
    }
  }
}
```

(2.3) Dekadentna, a povsem pravilna rešitev podnaloge (b):

```
public static String Kljuc(String vnos)
{
  StringTokenizer st = new StringTokenizer(vnos, " ");
  String resitev = "";
  while (st.hasMoreTokens())
    resitev += st.nextToken();
  resitev = resitev.toLowerCase();
  return resitev;
}
```

[15] Naš tekmovalec se lahko tolaži s tem, da je vsaj v dobri družbi. Znan je primer rednega profesorja na Univerzi v Ljubljani, ki je 28 črk pripisal celo slovenski abecedi...

[16] Nekdo očitno ni prebral na prvi strani besedila te naloge opombe pod črto, iz katere bi videl, da ima mali `a` (ki ga tukaj zanima) številko 97. Lahko bi tudi napisal kar `b + 'a'`, saj so v C-ju znaki pravzaprav le poseben primer celih števil.

(2.3) Pogumna rešitev podnaloge (c):

```
char *Kljuc(char *Z)
{
  int dolzina = Z.length();
  return dolzina;
}
```

(2.3) Cela rešitev podnaloge (c):

> Izračunaš dolžino besede in sortiraš po eni izmed metod za sortiranje (bubble itd.. . .).

(2.3) Dva koristna pogojna stavka pri podnalogi (d):

```
zac = Z[x][y];
i = 1;
if (i == 1)
  { if (zac > Z[x][y])
    { . . .
    }
  }
```

(2.3) Pri podnalogi (d) je eden od tekmovalcev napisal funkcijo Kljuc, ki dobi ključ iz niza Z tako, da v njem uredi posamezne znake (z bubble sortom). Tako bi npr. iz niza 63750 nastal ključ 03567.

(2.3) Zanimivost: nekateri imajo navado pisati (v C/C++) *(Z + i) namesto Z[i], da pridejo do i-tega znaka niza Z.

(2.3) Nekaterim stavek „Napiši tak podprogram Kljuc. . . “ in deklaracija tega podprograma nista dovolj. . .

> Nikjer ni napisano, kaj funkcija dobi kot podatek in kaj mora ta vrniti. Nikjer ni napisano, ali moramo spisati tudi urejanje ali tvoriti ključ.

(2.4) Cela rešitev te naloge:

> Pride lahko do negativnega stanja, če je stanje npr. 100 in ti vneseš za vzdig 100,1. Preprečiš lahko tako, da spremeniš double znesek v int znesek,

(2.4) Cela rešitev te naloge:

> Do negativnega stanja enostavno ne more priti zaradi **if** stavka. Lahko pa pride do izgube zaradi tega, ker je Racun deklarirano kot **int** in potem, če dvignemo 50,7 od 100, kar imamo ostane 49,3, ampak, ker je Racun **int**, se to zaokroži na 49 in pri tem izgubimo 0,3.
>
> To lahko rešimo tako, da v podprogramu Dvig deklariramo (**double** Racun, **double** Znesek).

(2.4) Cela rešitev te naloge:

> Do negativnega stanja na računu pride takrat, če je dvig večji, kot je stanje na računu.

(2.4) Cela rešitev te naloge:

> Do napačnega stanja na računu lahko pride, če vnesemo negativni znesek za dvig, zato moramo dati še eno zanko, ki bo preverjala in zagotavljala, da bomo lahko vnesli le pozitivne zneske.

(2.4) Cela rešitev te naloge:

> Do negativnega stanja na računu bi prišlo, če banka zaračuna še kakšno provizijo poleg dviga in bi bil seštevek teh dveh že večji od trenutnega stanja. Rešitev tega slučaja bi bila, da bi znesku prišteli še provizijo (znesek + provizija).

(2.5) Nek tekmovalec je naredil precej neupravičeno predpostavko:

> Naloga je podobna Zakladom... zato bom delal, kot da so figurice v arrayu, podobno kot pri zakladih...

Zaradi tega je štel kroglice na posameznem delu šahovnice tako, da je šel v zanki po vseh poljih tistega dela in pri vsakem polju z gnezdeno zanko po seznamu vseh figuric, da bi tako preveril, če na tistem polju stoji kakšna figura ali ne. Štetje figur na nekem delu šahovnice je torej v splošnem $O(n^4)$, če je del velik in je figur na šahovnici veliko. Ker mora figure šteti za vsako od $O(n^2)$ možnih razdelitev šahovnice, ima njegova rešitev skupno časovno zahtevnost $O(n^6)$.

Pomembna razlika med to nalogo in lovom na zaklade je, da je bilo znano, koliko bo zakladov (in to število je bilo majhno v primerjavi s številom vseh polj na igralni površini), tukaj pa nimamo nobenega podobnega zagotovila o številu figuric.

(2.5) Pripis na koncu neke rešitve:

> PS. Moja rešitev je zopet počasna. ☹

(3.1) Častna omemba v kategoriji "no, but nice try":

```
#include <fstream>
int main() {
  std::ifstream ifs("neboticniki.in"); int w, h, c1, c2; ifs >> w >> h >> c1 >> c2;
  std::ofstream ofs("neboticniki.out"); ofs << (h * c1);
  return 0;
}
```

Mogoče je pogledal primer v opisu naloge in iskal kakšen način, da bi iz štirih števil v vhodni datoteki (3, 2, 5, 9) izračunal pravilni rezultat (10)?

Kakorkoli že, malo kasneje je oddal še eno različico gornje rešitve, pri kateri se je spremenila le vrednost, ki jo izpisuje v četrti vrstici:

```
ofs << (c2 * c1) - (c1 + c1+1 + c1+2 + c1+3 + c1+4);
```

Vsekakor je treba priznati, da tudi ta pravilno reši testni primer iz opisa naloge.

(3.1) Nek drug tekmovalec je poslal rešitev, ki se začne takole:

```
#include <stdio.h>
voiud main ()
```

Ob oddaji ga je ocenjevalni strežnik seveda opozoril, da se ne prevede, in po 18 sekundah je oddal novo različico, v kateri je gornjo tipkarsko napako odpravil. Ampak to očitno pomeni, da programa pri sebi ni poskušal prevesti, preden ga je oddal. Če ga ni prevedel, ga tudi testirati ni mogel. Mar ljudje res mislijo, da bo nekaj deset vrstic dolg program že kar v prvem poskusu deloval brez napak? (Za povrhu je bila ta njegova rešitev čisto zgrešena.)

(3.1) V zadnjih minutah pred koncem tekmovanja oddajajo nekateri vse koščke izvorne kode, kar jim jih pride pod roke, najbrž v upanju, da bodo mogoče vendarle rešili vsaj kakšen testni primer. Spodnji primer je eden od bolj optimističnih, se pa za razliko od nekaterih drugih vsaj prevede...

```
#include <stdio.h>
int main() {
  int w, h, c1, c2;
  FILE *f;
  f = fopen("neboticniki.in", "r");
  fscanf(f, "%d %d %d %d", w, h, c1, c2);
}
```

(3.1) Nek tekmovalec je kot svojo prvo oddajo, več kot tri ure po začetku tekmovanja, poslal tole:

```
#include <fstream>
#include <iostream>
#include <stdio.h>
int main()
{
  std::ifstream ifs("neboticniki.in");
  int w, h, c1, c2, rezultat = 0;
  ifs >> w
      >> h
      >> c1
      >> c2;
  int parcela[2][3];
  for (int j = 0; j < h; j++)
  {
    parcela[0][j] = c2 - 1;
  }
  rezultat = (w + w) + 2 * h;
  std::ofstream ofs("neboticniki.out"); ofs << rezultat;
  return 0;
}
```

Njegova naslednja oddaja je prišla približno četrt ure kasneje. Spremenila se je le ena vrstica:

```
rezultat = ((h * w) - 2) - (c2 * h * w);
```

Še deset minut kasneje pa je poslal naslednjo različico, ki se drži načela: ko vse drugo odpove, poskusi (1) preimenovati spremenljivke in (2) izpisovati naključne rezultate.

```
#include <fstream>
#include <iostream>
#include <stdio.h>

int main()
{
  std::ifstream ifs("neboticniki.in");
  int dolzina, sirina, min, max, rezultat = 0 ;
  ifs >> dolzina
      >> sirina
      >> min
      >> max;

  rezultat = rand();

  std::ofstream ofs("neboticniki.out"); ofs << rezultat;
  return 0;
}
```

(3.2) Isti tekmovalec je šest minut kasneje že poslal tale program za naslednjo nalogo. To, da se niti ta niti rahlo popravljena različica (20 sekund kasneje) nista prevedli, ga ni odvrnilo od tega, da ne bi že dve minuti kasneje poslal tudi svoje prve oddaje za nalogo 3.3 (ki se tudi ni prevedla).

```
#include <fstream>

int main()
{
  std::ifstream ifs("neboticniki.in");
  int stopnice, pleskarji;
  ifs >> stopnice >> pleskarji;

  std::ofstream ofs("neboticniki.out"); ofs << rezultat;
  system("PAUSE");
  return 0;
}
```

(3.2) Pri tej nalogi so nekateri tekmovalci zelo razsipni s pomnilnikom. To, da si naslednji tekmovalec alocira skoraj 1 GB pomnilnika, da bi lahko spremljal stanje celega stopnišča — to lahko še nekako razumemo. Ampak on je potem to tabelo uporabljal kot tabelo 32-bitnih **int**ov, kar med drugim pomeni, da se mu bo program sesul, če bo stopnic več kot 250 milijonov.

```
void sprazni(int *stopnice, long n) {
  long i;
  for (i = 0; i < n; i++)
    stopnice[i] = 0;
}
.
.
.
int *stopnice = calloc(sizeof(char), 1000000000);
.
.
.
sprazni(stopnice, n);
```

(3.2) Nek drug tekmovalec je pri isti nalogi oddal program, ki se na srečo ni prevedel (niti po več poskusih, med katerimi ga je še malo popravljal), kajti če bi se, bi

poskušal za vsako stopnico alocirati strukturo z dvema **int**oma in enim kazalcem, torej vsaj kakšnih 12 bytov na stopnico...

(3.2) Nek drug tekmovalec pa je pretiraval v nasprotni smeri in si alociral majceno stopnišče na skladu:

```
int stopnica[99999];
```

Žal je to zadoščalo le za enega od naših testnih primerov. Če bi si alociral večjo tabelo, bi lahko znotraj časovne omejitve rešil še dva večja testna primera.

(3.2) Prva nagrada v kategoriji "no, but nice try":

```
struct _int
{
  int barva:6;
};
⋮
_int *barve = new _int[st_stopnic];
```

Žal je struktura v C++ vedno dolga vsaj 1 byte, ne pa le 6 bitov, kot je najbrž upal. (Mimogrede, ker je barv le 26, bi šlo tudi s 5 biti, le nepredznačena števila bi bilo dobro uporabiti.) Nekateri dandanašnji prevajalniki bi, vsaj pri privzetih nastavitvah prevajanja, dodelili vsaki takšni strukturi še več prostora, npr. 4 byte. Sicer pa pri tovrstnih rešitvah glavni problem ni bila poraba pomnilnika — 1 GB bi se ga na ocenjevalnem računalniku že še našlo — pač pa poraba časa. Pri sto milijonih stopnic bi še šlo, milijarda pa je za simulacijo na celem stopnišču praviloma že preveč.

(3.3) Nagrada za najbolj neumesten poskus razvijanja zank:

```
for (i = 0; i < stopnice; i++)
  { if (marked[i] == 1)
      barva[0] = barva[0] + 1;
    if (marked[i] == 2)
      barva[1] = barva[1] + 1;
    if (marked[i] == 3)
      barva[2] = barva[2] + 1;
    ⋮
    if (marked[i] == 26)
      barva[25] = barva[25] + 1;
  }
```

(3.3) Nek tekmovalec je kot svojo prvo oddajo, dve uri po začetku tekmovanja, poslal tole:

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main() {
  string pogresana1 = "Peter";
  string pogresana2 = "Janez";
```

```
    string pogresana3 = "Tadej";
    int j = 0;
    std::ifstream ifs("osebe.in");
    while (! ifs.eof())
    {
        string ime;
        ifs >> ime;
        if (pogresana1 == ime || pogresana2 == ime || pogresana3 == ime)
        {
            j++;
        }
    }
    std::ofstream ofs("osebe.out");
    ofs << j;
    return 0;
}
```

Slabo uro kasneje je poslal novo različico:

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main() {
    int j = 0;
    string primerjava;
    std::ifstream ifs("osebe.in");
    while (! ifs.eof())
    {
        string ime;
        ifs >> ime;
        if (ime == ime)
        {
            j++;
        }
        ifs >> primerjava;
    }
    std::ofstream ofs("osebe.out");
    ofs << j;
    return 0;
}
```

(3.3) Še ena iz skupine izrazito optimističnih oddaj (pet minut pred koncem tekmovanja):

```
#include <fstream>
#include <iostream>
#include <stdlib.h>

int main()
{
    for (;;)
        {}
            std::ofstream ofs("osebe.out"); ofs << rezultat;

    return 0;
}
```

Ko je opazil, da se ne prevede, je čez pol minute oddal „popravljeno“ različico:

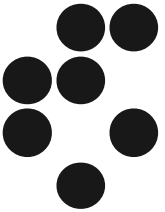
```
#include <fstream>
#include <iostream>
#include <stdlib.h>

int main()
{
  for (;;)
    {}
  int rezultat=*rezultat;
      std::ofstream ofs("osebe.out"); ofs << rezultat;

  return 0;
}
```

(Lov na zaklade) Kako izračunati število korakov, ki jih potrebujemo, da pridemo na karirasti mreži od z do k? Preprosta rešitev je max(abs(z.x − k.x), abs(z.y − k.y)), lahko pa do pravilnega rezultata pridemo tudi na počasnejši in bolj zapleten način:

```
public int Razdalja(Koordinate k, Koordinate z)
{
  // Koliko potez rabimo, da pridemo do tja:
  int moves = 0;
  int kx = k.x, ky = k.y;
  while (kx != z.x || ky != z.y)
  {
    kx −= max(min(kx − z.x, 1), −1);
    ky −= max(min(ky − z.y, 1), −1);
    moves++;
  }
  return moves;
}
```

SODELUJOČE INŠTITUCIJE

Institut Jožef Stefan



Institut je največji javni raziskovalni zavod v Sloveniji s skoraj 800 zaposlenimi, od katerih ima približno polovica doktorat znanosti. Več kot 150 naših doktorjev je habilitiranih na slovenskih univerzah in sodeluje v visokošolskem izobraževalnem procesu. V zadnjih desetih letih je na Institutu opravilo svoja magistrska in doktorska dela več kot 550 raziskovalcev. Institut sodeluje tudi s srednjimi šolami, za katere organizira delovno prakso in jih vključuje v aktivno raziskovalno delo. Glavna raziskovalna področja Instituta so fizika, kemija, molekularna biologija in biotehnologija, informacijske tehnologije, reaktorstvo in energetika ter okolje.

Poslanstvo Instituta je v ustvarjanju, širjenju in prenosu znanja na področju naravoslovnih in tehniških znanosti za blagostanje slovenske družbe in človeštva nasploh. Institut zagotavlja vrhunsko izobrazbo kadrom ter raziskave in razvoj tehnologij na najvišji mednarodni ravni.

Institut namenja veliko pozornost mednarodnemu sodelovanju. Sodeluje z mnogimi uglednimi institucijami po svetu, organizira mednarodne konference, sodeluje na mednarodnih razstavah. Poleg tega pa po najboljših močeh skrbi za mednarodno izmenjavo strokovnjakov. Mnogi raziskovalni dosežki so bili deležni mednarodnih priznanj, veliko sodelavcev IJS pa je mednarodno priznanih znanstvenikov.

Tekmovanje so podprli naslednji odseki IJS:

CT3 — Center za prenos znanja na področju informacijskih tehnologij

Center za prenos znanja na področju informacijskih tehnologij izvaja izobraževalne, promocijske in infrastrukturne dejavnosti, ki povezujejo raziskovalce in uporabnike njihovih rezultatov. Z uspešnim vključevanjem v evropske raziskovalne projekte se Center širi tudi na raziskovalne in razvojne aktivnosti, predvsem s področja upravljanja z znanjem v tradicionalnih, mrežnih ter virtualnih organizacijah. Center je partner v več EU projektih.

Center razvija in pripravlja skrbno načrtovane izobraževalne dogodke kot so seminarji, delavnice, konference in poletne šole za strokovnjake s področij inteligentne analize podatkov, rudarjenja s podatki, upravljanja z znanjem, mrežnih organizacij, ekologije, medicine, avtomatizacije proizvodnje, poslovnega odločanja in še kaj. Vsi dogodki so namenjeni prenosu osnovnih, dodatnih in vrhunskih specialističnih znanj v podjetja ter raziskovalne in izobraževalne organizacije. V ta namen smo postavili vrsto izobraževalnih portalov, ki ponujajo že za več kot 500 ur posnetih izobraževalnih seminarjev z različnih področij.

Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi virtualno učečo se skupnost in pripomoči k učinkovitejšemu povezovanju znanosti in industrije ter večji prepoznavnosti domačega znanja v slovenskem, evropskem in širšem okolju.

E1 — Avtomatika, biokibernetika in robotika

Raziskave Odseka za avtomatiko, biokibernetiko in robotiko obsegajo teme, ki obravnavajo značilnosti gibanja pri človeku ter njegovo povezavo z okoljem, s strojem ali tehnološkim procesom. Rezultate teh raziskav uporabljajo v industrijski avtomatizaciji in robotizaciji ter v različnih vejah medicine in v športu. Poleg temeljnih raziskav na teh področjih težimo k izvajanju raziskav, ki omogočajo, da se pridobljeno znanje in tehnologije čim prej prenesejo k uporabnikom.

Glavne smeri raziskav se nanašajo na integracijo mobilnosti in manipulacije pri industrijskih in servisnih robotih, na humanoidne robote, na študij fizioloških značilnosti človeka v različnih (ekstremnih) okoljih, na razvoj novih biomedicinskih naprav, metod in postopkov ter na problematiko avtomatizacije, robotizacije in informatizacije industrijske proizvodnje.

V odseku razvijamo in izdelujemo električne stimulatorje, ki jih uporabljajo po vsem svetu. V zadnjem času smo večji del raziskav usmerili v obravnavo bolnikov, ki imajo motnje dihanja v sodelovanju s Kliniko za pljučne bolezni Golnik. Namen raziskave je določiti aktivnost abdominalnih mišic med telesno vajo in vpliv visokofrekvenčne električne stimulacije na utrujanje teh mišic. Ugotovili smo potek periodične dihalne aktivnosti mišic glede na obremenitev in prag pojava. Rezultate bomo uporabili pri biopovratni vezavi in zmanjšanju aktivnosti, relaksaciji oziroma blokiranju abdominalnih mišic pri dihanju.

E2 — Odsek za sisteme in vodenje

Poslanstvo odseka, ki smo si ga postavili kot vodilno nit ob začetku delovanja, smo opredelili kot „premoščanje prepada med teorijo in prakso“.

Osrednji poudarek pri načinu našega dela je na zahtevi, da je treba izhajati iz konkretnih (industrijskih) problemov ter raziskave prilagajati temu, ne pa nasprotno. To dejstvo seveda potegne za seboj potrebo po širokem območju zelo različnih znanj, tako po tipu dela (raziskave, razvoj, inženirstvo itd.) kot tudi stroki (avtomatika, elektronika, računalništvo itd.).

Poslanstvo je izšlo iz potreb domačega okolja, saj so izkušnje pokazale, da spoznanj iz raziskav praktično ne moremo uporabiti pri reševanju konkretnih aplikativnih problemov in je s tem tudi opredelilo način našega dela.

Dejavnosti odseka obsegajo raziskave, razvoj in aplikacije na širšem področju računalniško podprtega vodenja in regulacije (predvsem) tehničnih sistemov, skupaj z ustreznimi storitvami, inženirstvom in izobraževanjem

S svojim znanjem vam lahko pomagamo na področju elektronike, merilne in regulacijske tehnike, računalniške avtomatizacije in informatizacije procesov.

Samo v času od svoje formalne ustanovitve (1986) je odsek delal večje ali manjše projekte za okoli 100 slovenskih in tujih podjetij, večinoma iz sektorja industrijske proizvodnje.

E8 — Odsek za tehnologije znanja

Poslanstvo Odseka za tehnologije znanja IJS je razvoj naprednih informacijskih tehnologij za zajemanje, shranjevanje, upravljanje in odkrivanje znanja s poudarkom na rudarjenju prodatkov oz. strojnem učenju, podpori odločanja in razvoju jezikovnih

tehnologij, katerih cilj je prispevati vrhunske znanstvene rezultate v svetovno zakladnico znanja ter pospeševati aplikacije teh tehnologij za razvoj e-znanosti in družbe znanja.

Dolgoročni cilji odseka so razvoj metod inteligentne analize podatkov, upravljanja znanja, podpore odločanja in računalniškega jezikoslovja ter njihova uporaba za reševanje praktičnih problemov na področju ekologije, medicine, zdravstvenega varstva, ekonomije in tržništva. V raziskave vključujemo tudi novejša področja informacijskih tehnologij: semantični splet in upravljanje mrežnih organizacij.

E9 — Odsek za inteligentne sisteme

Osnovni cilji Odseka za inteligentne sisteme so raziskave računalniških osnov inteligence in razvoj naprednih aplikacij s področja inteligentnih informacijskih storitev, analize podatkov, inteligentnega preiskovanja spleta, podpore odločanja, inteligentnih agentov, medicine, ekologije, jezikovnih tehnologij, inteligentne proizvodnje in ekonomije. Z več kot 20-letno tradicijo pri raziskavah in razvoju na širšem področju umetne inteligence, inteligentnih sistemov, medicinske informatike, procesiranja naravnega jezika in kognitivnih znanosti se je Odsek za inteligentne sisteme uveljavil v evropskem in svetovnem merilu. Sodelavci odseka so razvili več delujočih sistemov, ki so pomembni tako v slovenskem kot mednarodnem merilu.

Raziskovalna področja Odseka za inteligentne sisteme:

- induktivno logično programiranje
- evolucijsko računanje
- večstrategijsko učenje in principi mnogoterega znanja
- rudarjenje spletnih podatkov — sinteza znanja za modeliranje in vodenje sistemov
- sistemi za podporo odločanja
- principi inteligence in kognitivnih znanosti
- inteligentni agenti in večagentni sistemi
- umetna inteligenca v medicini
- sinteza govora
- ontologije in semantični splet
- analiza igranja iger.

*

Fakulteta za matematiko in fiziko

Fakulteta za matematiko in fiziko je članica Univerze v Ljubljani. Sestavljata jo Oddelek za matematiko in Oddelek za fiziko. Izvaja dodiplomske univerzitetne študijske programe matematike, računalništva in informatike ter fizike na različnih smereh od pedagoških do raziskovalnih.

Prav tako izvaja tudi podiplomski specialistični, magistrski in doktorski študij matematike, fizike, mehanike, meteorologije in jedrske tehnike.

Poleg rednega pedagoškega in raziskovalnega dela na fakulteti poteka še vrsta obštudijskih dejavnosti v sodelovanju z različnimi institucijami od Društva matematikov, fizikov in astronomov do Inštituta za matematiko, fiziko in mehaniko ter Instituta Jožef Stefan. Med njimi so tudi tekmovanja iz programiranja, kot sta Programerski izziv in Univerzitetni programerski maraton.

SREBRNI POKROVITELJI

Amebis
Podjetje Amebis se ukvarja z razvojem jezikovnih tehnologij, pomemben del dejavnosti pa predstavlja elektronsko založništvo slovarjev, leksikonov in enciklopedij ter podpora njihovim knjižnim verzijam. Posebej znani so slovarji ASP, saj jih uporablja velika večina Slovencev. Na področju jezikovnih modulov smo razvili in še naprej izpopolnjujemo slovenski črkovalnik, delilnik, tezaver in slovnični pregledovalnik. Vsi štirje omenjeni moduli lahko delujejo kot samostojni programi ali kot vgradni moduli nekaterih programov drugih programskih hiš (npr. Microsoft Word). Na področju strojnega prevajanja (naravnih jezikov) smo razvili prevajalni sistem Presis, ki trenutno prevaja iz slovenščine v angleščino in obratno, v razvoju pa so že novi jezikovni moduli. Število uporabnikov stalno narašča, iz verzije v verzijo pa se izboljšuje tudi kvaliteta prevajanja. Na področju govornih tehnologij se predvsem ukvarjamo s sintezo (slovenskega) govora. V sodelovanju z Institutom „Jožef Stefan“ smo razvili sintetizator Govorec (2.0), ki ga nameravamo v prihodnje izboljševati v smeri kvalitetnejšega in razumljivejšega govora. Precej energije skupaj z nekaterimi drugimi partnerji posvečamo gradnji besedilnih korpusov, ki nam predstavljajo jezikovno infrastrukturo za razvoj drugih jezikovnih baz. Kot zadnje večje področje naj omenimo še sisteme dialoga in baze znanja, ki računalnikom in ljudem omogočajo pisno ali govorno komunikacijo v naravnem (slovenskem) jeziku.

CDE
Podjetje CDE nove tehnologije d. d. sodi med vodilne v Sloveniji na področju razvoja in integracije sodobnih računalniških tehnologij s poudarkom na telekomunikacijah in telefoniji. Družba je po dejavnostih razdeljena na pet profitnih centrov:

- CTI.programska.oprema
  CDE razvija programsko opremo, kot so klicni in kontaktni centri, glasovne storitve, PC telefoni in aplikacije računalniško podprte telefonije — CTI.
- COCOS VOIP poslovno omrežje
  VOIP poslovna omrežja niso več razmišljanje o prihodnosti, temveč so del sedanjosti in predstavljajo optimalno rešitev nadgradnje obstoječih telekomunikacijskih sistemov. COCOS VOIP poslovno omrežje temelji na COCOS IP PBX komunikacijskem strežniku, ki se prek standardnih vmesnikov povezuje do obstoječih

operaterjev fiksnega omrežja, novih VOIP operaterjev, prek omrežja LAN pa do uporabniških terminalov.

- KLICNI.STUDIO
  Tržimo storitve lastnega kontaktnega centra s 23 delovnimi mesti za telefonske agente. V CDE KLICNEM.STUDIU za znane naročnike sprejemamo in opravljamo klice s področja prodaje, informiranja strank, svetovanja, ohranjanja stika in pozornosti do kupcev.
- CALL.CENTER.COLLEGE
  Imamo svoj izobraževalni center, v katerem se šola agente klicnega in kontaktnega centra, tu potekajo treningi telefonske komunikacije za operaterje in poslovne sekretarje, seminarji za vodje klicnih in kontaktnih centrov, šola se tehnično osebje za podporo delovanja klicnih in kontaktnih centrov.
- CD.DVD.produkcija
  V sklopu svojih aktivnosti sodi CDE d. d. tudi med vodilne v Sloveniji na področju produkcije CD-ROM diskov, kjer predstavlja SONY DADC Austria AG solventnega partnerja, ki zagotavlja konkurenčnost in kakovost.

Datalab

Podjetje Datalab d. d. je bilo ustanovljeno 3. 9. 1997 z namenom razvoja integralnih poslovno-informacijskih sistemov za mala in srednja podjetja.

Ukvarjamo se z razvojem poslovne programske opreme ter svetovanjem. Zavedamo se, da je uspešno in skrbno vodenje podjetja mogoče le ob podpori kakovostnega in zanesljivega informacijskega sistema.

Smo tim strokovnjakov različnih smeri z ogromno znanja in izkušenj pri reševanju vprašanj in težav pri uporabi sodobne tehnologije.

Naš slogan *Spremenite podatke v dobiček* jasno kaže na našo poslovno filozofijo. Menimo namreč, da ni cilj posedovati informacijsko tehnologijo, ampak z njo delati bolje.

Ciljni segmenti so mikro, mala in srednja podjetja ter njihovi računovodski servisi.

Datalab d. d. razvija uporabniku prijazen poslovno informacijski sistem PANTHEON™, ki omogoča enostavno zbiranje, analiziranje, planiranje in kontroliranje podatkov , transakcij in procesov. Je poslovna programska oprema z a e-stoletje, ki uporabnikom omogoča pridobivanje in ohranjanje konkurenčne prednosti. Število uporabnikov se vsak dan povečuje, trenutno imamo preko 11000 uporabnikov.

Podjetje več kot uspešno pokriva velik del slovenskega trga in le-tega vsako leto širi. Poleg tega pa je podjetje razširjeno tudi v tujino in sicer svoje podružnice ima v naslednjih državah: Bosna, Srbija, Hrvaška, Makedonija, Črna gora, Kosovo, Nemčija in Italija. Naslednji dve državi, ki ju želimo osvojiti, sta Romunija in Bolgarija.

Datalab od leta 2006 vsako šolsko leto štipendira dva nova študenta s področja računalniškega programiranja, informatike ali ekonomije oziroma organizacije.

Marg

Margovci so hecni možici, ki radi ustvarjajo, se zabavajo in spoznavajo novo. Včasih igrajo biljard, včasih fusbal, večkrat pa tudi trdo delajo. Spoznavajo in soustvarjajo nove tehnologije, trudijo se povezovati ljudi in svetove ter graditi skupnost. Verjamejo, da je prihodnost v povezovanju in da vsaka nova možnost sodelovanja pomeni korak naprej. Zato načrtujejo in razvijajo orodja, ki gradijo informacijsko družbo 21. stoletja.

Margovci so tudi šarmerji. Nimajo kravat, le kratke hlače in navihan pogled.

Margovci pa nismo le mi, Margovec si lahko tudi ti. Pokliči in obišči nas, pridi na kak naš dogodek. Morda ti bo pa v naši družbi všeč in morda bodo ravno tvoje ideje odprle nova obzorja. Za bolj resne imamo zato Margovci tudi spletno stran *www.marg.si*. Tam najdeš vse o nas, čeprav je stran, kot prava kovačeva kobila, že malce zastarela.

Pri Margu je meja med delom, zabavo in prostim časom zelo zabrisana. Ne verjameš? Presodi sam!

Quintelligence

Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurencne prednosti. Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja. IKT (informacijsko-komunikacijska tehnologija) je postavila temelje za nemoten pretok in hranjenje podatkov in informacij. S primernimi metodami je potrebno na osnovi teh informacij izpeljati ustrezne analize in odločitve. Nivo upravljanja in delovanja se tako seli iz informacijske logistike na mnogo bolj kompleksen in predvsem nedeterminističen nivo razvoja in uporabe metodologij. Tako postajata razvoj in uporaba metod za podporo obvladovanja znanja (knowledge management, KM) vedno pomembnejši segment razvoja.

Podjetje Quintelligence je in bo usmerjeno predvsem v razvoj in izvedbo metod in sistemov za pridobivanje, analizo, hranjenje in prenos znanja. S kombiniranjem delnih — problemsko usmerjenih rešitev, gradimo kompleksen in fleksibilen sistem za podporo KM, ki bo predstavljal osnovo globalnega informacijskega centra znanja.

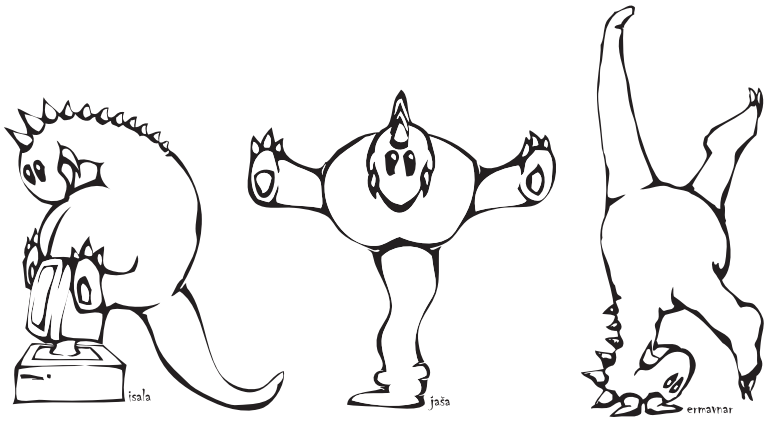
Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.

BRONASTI POKROVITELJI


rrc
RRC Računalniške storitve, d.d.


SUN
creative solutions


ZRC
ZASAVSKI RAČUNSKI CENTER


isala
jaša
ermavnar