3. tekmovanje IJS v znanju računalništva

Institut Jožef Stefan, Ljubljana, 29. marca 2008

Bilten

Bilten 3. tekmovanja IJS v znanju računalništva
Institut Jožef Stefan, 2008

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Andrej Bauer, Matija Grabnar, Miha Grčar, Tomaž Hočevar, Boris Horvat, Aleksandar Jurišić, Klemen Kenda, Peter Keše, Jurij Kodre, Mark Martinec, Mojca Miklavec, Mitja Trampuš, Miha Vuk, Klemen Žagar, Janez Brank.

Tisk: Tiskarna knjigoveznica Radovljica, d. o. o.
Naklada: 350 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

37.091.27:004(497.4)

TEKMOVANJE IJS v znanju računalništva (3 ; 2008 ; Ljubljana)
Bilten [Elektronski vir] / 3. tekmovanje IJS v znanju računalništva, Ljubljana, 29. marca 2008 ; avtorji nalog Nino Bašić ... [et al.] ; uredil Janez Brank. — Ljubljana : Institut Jožef Stefan, 2008

Način dostopa (URL): `http://rtk.ijs.si/2008/rtk2008-bilten.pdf`

ISBN 978-961-264-008-8
1. Bašić, Nino 2. Brank, Janez, 1979–
241906944

KAZALO

Struktura tekmovanja	5
Nasveti za 1. in 2. skupino	6
Naloge za 1. skupino	9
Naloge za 2. skupino	13
Navodila za 3. skupino	19
Naloge za 3. skupino	22
Nalogi za ogrevanje	28
Neuporabljene naloge iz leta 2006	30
Rešitve za 1. skupino	40
Rešitve za 2. skupino	47
Rešitve za 3. skupino	58
Rešitvi nalog za ogrevanje	85
Rešitve neuporabljenih nalog 2006	88
Rezultati	118
Nagrade	122
Šole in mentorji	123
Tekmovanje programov: Štiri v vrsto	124
Off-line naloga: Kino	127
Anketa	134
Rezultati ankete	138
Cvetke	146
Sodelujoče inštitucije	153
Pokrovitelji	156

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 vsaki skupini dobijo tekmovalci po pet nalog; pri ocenjevanju štejejo posamezne naloge kot enakovredne (v prvi in drugi skupini lahko dobi tekmovalec pri vsaki nalogi do 20 točk, v tretji pa pri vsaki nalogi do 100 točk).

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 20.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, 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. 6–8 za 1. in 2. skupino, str. 19–21 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 124–126) in tekmovanje v off-line nalogi (ki je podrobneje predstavljeno na straneh 127–132).

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:

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

```c
#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;
}
```

Še isti trije primeri v pythonu:

```
# Branje dveh števil in izpis vsote:
import sys

a, b = sys.stdin.readline().split()
a = int(a); b = int(b)
print "%d + %d = %d" % (a, b, a + b)
```

```
# Branje standardnega vhoda po vrsticah:
import sys

i = d = 0
for s in sys.stdin:
  s = s.rstrip('\n') # odrežemo znak za konec vrstice
  i += 1; d += len(s)
  print "%d. vrstica: \"%s\"" % (i, s)
print "%d vrstic, %d znakov." % (i, d)
```

```
# Branje standardnega vhoda znak po znak:
import sys

i = 0
while True:
    c = sys.stdin.read(1)
    if c == "": break # EOF
    sys.stdout.write(c)
    if c != '\n': i += 1
print "Skupaj %d znakov." % i
```

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. GrbaveBesede

Nekateri spletni wiki strežniki omogočajo skrajšan zapis sklicev na druge dokumente tako, da v besedilu prepoznajo grbave besede (angleško: CamelCase) in jih obravnavajo posebej. Grbava beseda je sestavljena iz velikih in malih črk, pri tem se mora v njej vsaj dvakrat pojaviti par velike in male črke (v tem vrstnem redu).

Primeri grbavih besed:	Primeri besed, ki *niso* grbave:
`DomNaKrimu`	`dom`
`WriteLn`	`naKrimu`
`skokNaKonec`	`Grbavebesede`
`NaCl`	`Write`
`NotredamskiZvonar`	`NaOH`

Napiši program, ki bo prebral besedilo iz vhodne datoteke (bere naj do konca, torej do EOF) in v njem poiskal vse grbave besede ter jih izpisal, vsako v svojo vrstico. Besede v vhodni datoteki so ločene med seboj s presledki ali konci vrstic; predpostaviš lahko, da števil in posebnih znakov v besedilu ni.[1]

2. Predavalnice

Na neki srednji šoli prirejajo večji dogodek. Potekal bo ves dan in bo obsegal kopico predavanj. Vsako predavanje ima znan čas začetka in konca ter zasede eno predavalnico. Nekatera predavanja potekajo hkrati, zato je seveda potrebnih več učilnic. (Časi predavanj so določeni tako, da je v čas posameznega predavanja že zajet tudi čas, ki je potreben za to, da se na koncu predavanja predavalnica izprazni in da lahko vanjo pridejo poslušalci naslednjega predavanja. Torej, če imamo podatek, da se neko predavanje začne ob istem času, ob katerem se neko drugo predavanje konča, to pomeni, da lahko tidve predavanji uporabljata isto predavalnico.) Pomagaj ravnatelju napisati program, ki bo izračunal največje število hkratnih predavanj, da bo znal priskrbeti potrebno število predavalnic.

(*a*) [15 točk] Ravnatelj pozna čas (ure in minute) začetka in konca vsakega dogodka. **Opiši postopek**, ki prebere te podatke in izpiše potrebno število predavalnic. Poskusi najti čim učinkovitejši postopek, ki bo deloval hitro tudi v primerih, ko je predavanj zelo veliko. Koliko je minut v enem dnevu?

[1] Zgornje besedilo je tako, kakršno so dobili dijaki na tekmovanju. Izkaže se, da je v tej obliki naloga za reševalce v C-ju malo težja kot za tiste, ki delajo npr. v pascalu ali C++u, ker nimajo na voljo preprostega mehanizma za delo z nizi poljubne dolžine (kot je tip string v pascalu in C++). Mišljeno je, da lahko tekmovalec predpostavi, da nobena beseda v vhodni datoteki ni daljša od npr. 100 znakov.

(*b*) [5 točk] Kaj pa, če časi začetka in konca predavanj niso podani v urah in minutah, pač pa kot neka realna števila? To pomeni, da vsako predavanje pravzaprav določa nek interval na številski premici, mi pa bi radi poiskali, kje se prekriva največ teh intervalov (oz. koliko se jih tam prekriva). **Opiši postopek**, ki bi rešil tudi to različico naloge (oz. utemelji, če jo reši že postopek, ki si ga opisal v odgovoru na podnalogo (*a*)).

3. Darila

Na novoletni zabavi so se prisotni obdarovali z darili. Znano je število prisotnih oseb, za vsak par oseb pa vemo tudi, koliko je bilo vredno darilo, ki ga je dala ena oseba drugi. **Napiši program**, ki ugotovi, kdo je imel največ dobička (torej pri kom je razlika med skupno vrednostjo prejetih in podarjenih daril največja). Če je enak največji dobiček doseglo več oseb, je vseeno, katero od njih izpišeš.

Pri tem lahko predpostaviš, da že obstajata naslednji dve funkciji, ki ju lahko pokličeš, da dobiš podatke o ljudeh in darilih:

- **function** StOseb: integer; { *v pascalu* }
  int StOseb(); /* *v C/C++* */
  def StOseb(): . . . # *v pythonu*

  Vrne število oseb na zabavi.

- **function** Darilo(a, b: integer): integer; { *v pascalu* }
  int Darilo(**int** a, **int** b); /* *v C/C++* */
  def Darilo(a, b): . . . # *v pythonu*

  Vrne vrednost darila, ki ga je oseba a podarila osebi b. Posamezne osebe so oštevilčene s celimi števili od 1 do StOseb.

4. Elektronska ključavnica

V računskem centru nekega inštituta je vhod varovan z elektronsko ključavnico in številčnico. Na številčnici so števke od 0 do 9 in tipka „Prekliči“. Geslo je neko zaporedje števk.[2]

Vrata je treba odpreti, takoj ko uporabnik vtipka pravilno geslo. Če pritisne na tipko „Prekliči“, se vse tisto, kar je natipkal pred tem, ne upošteva. Prav tako naj se po uspešnem odprtju vrat zbiranje gesla začne znova.

Na primer, če je geslo 3119 in

- uporabnik natipka 3119 ⟶ vrata odpremo
- uporabnik natipka 3118 ⟶ vrat NE odpremo
- uporabnik natipka 33119 ⟶ vrat NE odpremo
- uporabnik natipka 13⟨prekliči⟩3119 ⟶ vrata odpremo

[2]Mogoče velja posebej opozoriti, da naloga ne daje zagotovil o dolžini gesla. Na tekmovanju so mnogi tekmovalci predpostavili, da je geslo dolgo natanko štiri števke, ker je pač v nalogi podan primer gesla s štirimi števkami. Mišljeno pa je, da bi tekmovalec napisal rešitev, ki deluje za poljubno dolga gesla.

Napiši program, ki v neskončni zanki bere pritisnjene tipke (vsak pritisk na tipko je sporočen samostojno) in odpre vrata, ko je vtipkano pravilno zaporedje. Predpostaviš lahko, da že obstajata naslednja podprograma in spremenljivka:

- **var** Geslo: string; — niz števk, ki je pravo geslo;
- **procedure** Odkleni; — odklene ključavnico;
- **function** PreberiTipko: char; — počaka, da uporabnik pritisne kakšno tipko in jo vrne kot rezultat funkcije (eno od števk od `'0'` do `'9'`, ob pritisku na tipko „Prekliči“ pa vrne znak `'P'`).

Še deklaracije v C/C++:

```
const char* geslo;
void Odkleni();
char PreberiTipko();
```

In v pythonu:

```
geslo = "..."
def Odkleni(): ...
def PreberiTipko(): ...
```

5. Kdo je izdal skrivnost?

Telefonski operaterji zbirajo tako imenovane prometne podatke o telefonskih pogovorih: kdo je koga klical in kdaj. Čeprav pri tem zbiranju še ne gre za prisluškovanje pogovorom, so tudi prometni podatki strogo zasebni podatki, saj se da na njihovi podlagi marsikaj sklepati o klicočih in jih lahko operaterji posredujejo preiskovalnim organom samo na izrecno zahtevo sodišča ob utemeljenem sumu kaznivih dejanj (ko lahko sodišče odredi tudi prisluškovanje imetnikom izbranih telefonskih številk).

Pa recimo, da teh moralnih dilem nimamo. Dobili smo zaporedni seznam prometnih podatkov nekega operaterja o vseh telefonskih klicih v nekem časovnem obdobju. Elementi seznama so pari telefonskih številk, ki sta uspešno vzpostavili telefonsko zvezo, torej takole (zaradi enostavnosti bomo pri tej nalogi predpostavili, da so vse telefonske številke štirimestne, npr. interne številke nekega podjetja):

```
1705 2312
1326 1705
3789 1230
2312 2372
0137 1705
```

in tako dalje. Seznam je lahko zelo dolg, saj se samo v enem dnevu opravi nekaj milijonov telefonskih klicev. Zaradi enostavnosti v seznamu niso zapisani časi klicev, vendar je seznam urejen po časovnem zaporedju, torej prvi element seznama je prvi klic v opazovanem časovnem obdobju in tako naprej.

Vprašanje: zanima nas, ali je neka zaupna informacija, ki jo je imel lastnik telefonske številke a, lahko prek telefonskih pogovorov prišla do osebe, ki je lastnik

telefonske številke b? Mogoče se je oseba a z b-jem pogovarjala neposredno (potem je odgovor preprost), lahko pa se je a najprej pogovarjal z nekim c, potem je d poklical c-ja in nazadnje je d poklical b-ja. Informacija od a do b bi lahko prišla tudi še po kakšni drugačni poti, npr. tako, da e najprej pokliče a-ja in potem b pokliče e-ja.

(1) [15 točk] **Opiši postopek**, ki pregleda seznam vzpostavljenih zvez in za dani dve telefonski številki iz seznama, npr. a in b, ugotovi, ali je lastnik telefonske številke b lahko prišel do zaupne informacije, ki jo je imel lastnik številke a.

(2) [5 točk] **Opiši**, kaj bi bilo treba v postopku, ki si ga sestavil pri točki (1), spremeniti, da bi izpisal najkrajše zaporedje klicev, po katerem je zaupna informacija lahko prišla od a do b. (Lahko se izkaže tudi, da primernega zaporedja klicev sploh ni. Na primer, pri zgornjem primeru petih klicev ni načina, da bi prišla informacija od številke 1326 do številke 2312. Številka 1326 informacijo sicer lahko pove številki 1705, vendar po tem klicu številka 1705 komunicira le še s številko 0137, ne pa s številko 2312.) Pri tem „najkrajše zaporedje“ pomeni tisto, ki ga sestavlja najmanjše število klicev.

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. Roboti

Na kvadratni mreži velikosti $n \times n$ polj se peščica preprostih robotov premika naokrog in opravlja določene naloge. Vsak robot pri izvrševanju svojih nalog opravlja vedno enako krožno pot, ki se začne in konča v isti točki (je pa ta pot pri različnih robotih različna in tudi začetna/končna točka je pri različnih robotih lahko različna).

Pot vsakega posameznega robota je torej opisana z zaporedjem polj (x, y), pri čemer sta prvi in zadnji položaj enaka. Koordinate so cela števila od 1 do n. V tem zaporedju so podane koordinate čisto vseh polj, po katerih se je robot gibal. Dano je eno tako zaporedje za vsakega robota (vseh robotov pa je z).

Vse te preproste robote bi radi nadomestili z enim samim robotom Marvin™, ki bi opravljal naloge vseh dosedanjih robotov. Da bo imel Marvin za delo dovolj energije, bi mu radi postavili električni priključek v neko tako polje, ki ga bo lahko obiskal spotoma, ne glede na to, po poti katerega izmed prvotnih z robotov se trenutno giblje.

Opiši postopek, ki odkrije, ali obstaja vsaj eno skupno polje, po katerem so se gibali vsi roboti. Razmisli, kako bi tak postopek deloval v zelo velikih kvadratnih mrežah (torej pri velikem n).

2. rsync

Ko je Cefizelj ukradel občinsko blagajno, je skupaj z njo odnesel še trdi disk, na katerem so bili shranjeni podatki o vseh Butalcih. Saj ne, da bi jih skrbelo za varnost podatkov (Cefizelj tako ali tako ne zna brati), toda zdaj so sami ostali brez njih in bi jih radi nazaj.

Na srečo je župan poskrbel, da so vse zaupne in sila pomembne podatke shranjevali na blu-ray tlačenke, dasiravno jih (tlačenk) od takrat, ko jih je Cefizelj že pred časom premešal, niso znali več urediti po datumu. Ostal pa jim je seznam zapisov oblike ⟨številka tlačenke, ime datoteke, datum⟩. Primer takega seznama:

Tlačenka	Datoteka	Datum
1	`kozmijan_buta.txt`	1. 1. 2008
2	`gregor_brezhlacnice.txt`	7. 3. 2007
3	`kozmijan_buta.txt`	1. 1. 2006
3	`gregor_brezhlacnice.txt`	1. 1. 2008
4	`kozmijan_buta.txt`	5. 6. 2007
4	`gregor_brezhlacnice.txt`	3. 4. 2007
4	`fido_kljukec.txt`	1. 1. 2006

Seznam je trenutno urejen po številki tlačenke, župan pa bi zdaj rad za vsako datoteko poiskal njeno najnovejšo varnostno kopijo (torej tisto z najkasnejšim datumom). Posebej opozorimo na to, da tlačenke niso nujno oštevilčene na nek sistematičen in

koristen način, torej se ne moremo zanesti na to, da so na tlačenkah z višjimi številkami novejše različice datotek ali kaj podobnega.

Župan je prišel na idejo, da bi problem rešili takole:

(*i*) uredi seznam padajoče po datumu (novejši naprej);

(*ii*) uredi (že urejeni) seznam naraščajoče po imenih datotek;

(*iii*) sprehodi se čez celoten seznam datotek (po vrstnem redu) — če je datoteka po imenu enaka prejšnji, jo briši s seznama.

Pri tem pa poteka urejanje v točki (*i*) po naslednjem postopku (recimo, da so podatki o datotekah v neki tabeli T z indeksi od 0 do $n-1$):

```
1  za vsak i od 0 do n − 2:
2     za vsak j od i + 1 do n − 1:
3        če datum datoteke T[i] ni kasnejši kot datum datoteke T[j],
4           potem med seboj zamenjaj tidve datoteki v tabeli T
            (torej pride dosedanja T[i] v celico z indeksom j,
            dosedanja T[j] pa v celico z indeksom i);
```

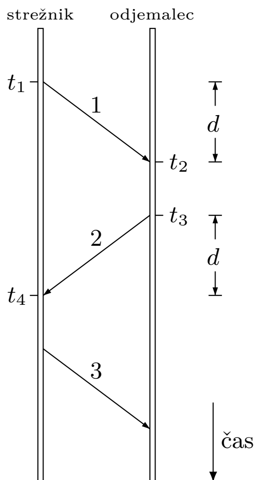
Urejanje v točki (*ii*) poteka po čisto podobnem postopku, le vrstica 3 se spremeni takole:

```
3′       če ime datoteke T[i] ni po abecedi pred imenom datoteke T[j],
```

Naloga: ali opisani postopek za iskanje najnovejših različic vseh datotek deluje pravilno, torej ali res obdrži v seznamu najnovejšo različico datoteke? Če da, to utemelji; če ne, pa pokaži primer, na katerem pride opisani postopek do napačnih rezultatov, in opiši, kako bi ga popravil, da bi deloval pravilno.

3. Usklajevanje ur

Protokol PTP (Precision Time Protocol, IEEE 1588) omogoča sinhronizacijo ur med dvema računalnikoma (eden je strežnik, ki pozna točen čas, drugi pa je odjemalec in bi si rad nastavil uro na čas, ki je čim bliže tistemu na strežniku). Pri tej nalogi si bomo ogledali malo poenostavljeno različico tega protokola. Usklajevanje ur poteka v treh korakih:



1. Najprej strežnik pogleda svoj trenutni lokalni čas (recimo, da je to t_1) in ga pošlje odjemalcu.

2. Odjemalec to sporočilo sprejme in si zapomni svoj lokalni čas sprejema (recimo, da je to t_2). Odjemalec nato pošlje strežniku odgovor na njegovo sporočilo in si zapomni svoj lokalni čas, ko je ta odgovor poslal (recimo t_3).

3. Strežnik prejme ta odgovor, pogleda svoj lokalni čas sprejema (recimo t_4) in ga pošlje odjemalcu.

Pri tem protokolu predpostavljamo, da potujejo sporočila v obe smeri (od strežnika do odjemalca in obratno) enako dolgo, na primer d časovnih enot; predpostavimo tudi, da se sporočila ne izgubljajo. Odjemalec torej lahko razmišlja takole: če je njegova ura na primer za r časovnih enot pred tisto na strežniku, velja $t_2 = t_1 + d + r$ in $t_4 = t_3 - r + d$. Ker odjemalec po zgoraj opisani izmenjavi sporočil pozna čase t_1, t_2, t_3 in t_4, lahko izračuna r (po formuli $r = (t_2 - t_4 - t_1 + t_3)/2$) in torej ve, da mora svojo uro pomakniti za r enot nazaj (če je r negativen, to seveda v resnici pomeni, da mora uro pomakniti za $-r$ enot naprej).

Napiši naslednje **podprograme**, ki naj poskrbijo za usklajevanje ure po opisanem protokolu:

- **procedure** SinhronizirajUro; — to je podprogram, ki ga bo uporabnik poklical na strežniku, ko je potrebno sinhronizirati uro na odjemalcu s tisto na strežniku. Ta podprogram naj naredi, kar je pač treba, da se bo začel postopek sinhronizacije po opisanem protokolu.

- **procedure** SprejemNaStrezniku(Sporocilo: SporociloT); — to je podprogram, ki ga bo sistem poklical na strežniku, ko le-ta prejme kakšno sporočilo od odjemalca.

- **procedure** SprejemNaOdjemalcu(Sporocilo: SporociloT); — to je podprogram, ki ga bo sistem poklical na odjemalcu, ko le-ta prejme kakšno sporočilo od strežnika.

Tip SporociloT si definiraj sam in vanj vključi vse tiste podatke, ki jih tvoji podprogrami potrebujejo za implementacijo opisanega protokola.

```
type SporociloT = record
  ...     { dopolni po lastnih željah in presoji }
end; {SporociloT}
```

Predpostaviš lahko, da se čas meri v nekih majhnih osnovnih enotah (npr. mikrosekundah) od nekega vnaprej dogovorjenega začetnega dogodka. Predpostavi tudi, da je tip integer (oz. **int** v C/C++) dovolj velik za shranjevanje celih števil, s katerimi merimo čas pri tej nalogi. Tvoji podprogrami si lahko pomagajo z naslednjimi podprogrami, za katere predpostavi, da so že na voljo:

- **function** VrniUro: integer; — vrne trenutni lokalni čas.
- **procedure** NastaviUro(t: integer); — nastavi lokalni čas računalnika na t.
- **procedure** Poslji(Sporocilo: SporociloT); — pošlje Sporocilo drugemu računalniku.

Še deklaracije v C/C++:

```
typedef struct { /* definiraj sam(a) */ } SporociloT;

/* Podprogrami, ki jih moraš napisati ti: */
void SinhronizirajUro();
void SprejemNaStrezniku(SporociloT sporocilo);
void SprejemNaOdjemalcu(SporociloT sporocilo);

/* Podprogrami, za katere lahko predpostaviš, da že obstajajo: */
int VrniUro();
void NastaviUro(int t);
void Poslji(SporociloT sporocilo);
```

In v pythonu:

```
# Podprogrami, ki jih moraš napisati ti:
def SinhronizirajUro(): ...                # naj ne vrača ničesar
def SprejemNaStrezniku(sporocilo): ...     # naj ne vrača ničesar
def SprejemNaOdjemalcu(sporocilo): ...     # naj ne vrača ničesar

# Podprogrami, za katere lahko predpostaviš, da že obstajajo:
def VrniUro(): ...                         # vrne neko vrednost tipa int
def NastaviUro(t): ...                     # vrne None
def Poslji(sporocilo): ...                 # vrne None
```

4. Društvo ljubiteljev ničel

Pred časom se je pojavil iskalnik Blackle, ki posnema videz Googla, vendar uporablja črno ozadje namesto belega, menda v upanju, da bo zaslon za prikaz črnih pik porabil manj elektrike, kot bi je za prikaz belih pik.

Po zgledu Blackla se je osnovalo Društvo ljubiteljev ničel, ki se zavzema za to, da bi bilo čim več bitov ugasnjenih tudi v pomnilniku računalnika. Blok pomnilnika si lahko predstavljamo kot tabelo n besed, vsaka beseda pa je 16-bitno število od 0 do 65535:

```
const n = ...;
type word = 0..65535;
     BlokT = array [0..n − 1] of word;       { v pascalu }

#define n ...
typedef unsigned short BlokT[n];             /* v C/C++ */
```

Če vse besede v našem bloku xor-amo z neko konstantno vrednostjo c (ki je tudi celo število od 0 do 65535), se lahko zgodi, da bo po tej operaciji število prižganih bitov v bloku manjše kot prej. (Pri tem pa bomo lahko kasneje s ponovnim xor-anjem spet rekonstruirali prvotne podatke, če jih bomo še kdaj potrebovali.) Pomagaj članom Društva ljubiteljev ničel in jim **napiši funkcijo** NajvecNicel, ki bo za dani blok podatkov ugotovila, s katero konstanto c bi bilo treba xor-ati vse besede v njem, da bi bilo po tem skupno število ugasnjenih bitov v bloku največje možno. (Če se da to največje število ugasnjenih bitov doseči pri več različnih vrednostih konstante c, je vseeno, katero od njih vrne tvoja funkcija.) Tvoja funkcija naj ustreza naslednji deklaraciji:

```
function NajvecNicel(Blok: BlokT): word;     { v pascalu }
unsigned short NajvecNicel(BlokT blok);      /* v C/C++ */
def NajvecNicel(blok): ...                   # v pythonu; blok bo seznam (list) int-ov,
                                             # funkcija NajvecNicel pa naj vrne int
```

Poskusi poskrbeti, da bo tvoja rešitev učinkovita in da bo funkcija delovala čim hitreje tudi pri velikih n.

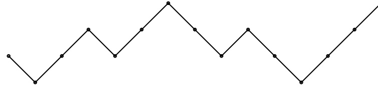
Operacija xor (v C/C++ in pythonu jo opravlja operator ^, v pascalu pa **xor**) deluje tako, da je nek bit v rezultatu prižgan natanko tedaj, kadar sta na tistem mestu v operandih bita različna (torej ko je tisti bit v enem od operandov prižgan, v enem pa ugasnjen). Primer:

$$\begin{aligned} 12345_{10} &= 0011\,0000\,0011\,1001_2 \\ 45678_{10} &= 1011\,0010\,0110\,1110_2 \\ 12345_{10} \text{ xor } 45678_{10} &= 1000\,0010\,0101\,0111_2 = 33367_{10}, \end{aligned}$$

torej je $12345 \text{ xor } 45678 = 33367$.

5. Cik cak

Predstavljajmo si cikcakasto črto, ki se v vsakem koraku premakne diagonalno za eno enoto desno in eno enoto gor ali pa za eno enoto desno in eno enoto dol. Primer:



Takšno črto lahko narišemo tudi v tekstovnem načinu, npr. takole:

```
...../\....../
../\/oo\/\../o
\/oooooooo\/oo
```

Da bo slika lepša, smo prazna polja prikazali s pikami, če ležijo nad črto, in z znaki „o“, če ležijo pod njo; črto pa smo sestavili iz znakov „/“ (za premik desno in gor) in „\“ (za premik desno in dol).

Opazimo lahko, da je potek črte enolično določen že z zaporedjem znakov „/“ in „\“. Črto na zgornjem primeru lahko torej predstavimo z nizom \//\//\\/\\///. **Napiši podprogram** NarisiCrto, ki kot parameter dobi niz znakov „/“ in „\“, ki opisuje neko tako cikcakasto črto; tvoj podprogram naj to črto nato izriše prek več vrstic s pomočjo znakov „/“, „\“, „.“ in „o“, tako kot je to prikazano na zgornjem primeru.

Tvoj podprogram naj ustreza naslednji deklaraciji:

```
procedure NarisiCrto(s: string);     { v pascalu }
void NarisiCrto(char *s);            /* v C/C++ */
def NarisiCrto(s): ...               # v pythonu
```

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++, C# ali java, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz JDK 1.6 in s prevajalnikom za C# iz Visual Studia 2008. Za delo lahko uporabiš `FP` oz. `ppc386` (FreePascal), `GCC/G++` (GNU C/C++ — command line compiler), `GCJ` (za javo 1.4), Java 2 SDK (za javo 1.6) in Visual Studio 2005.

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `RTK.EXE`, ki ga lahko uporabiš za oddajanje 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`
`rtk` *ImeNaloge*.`cs`

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Datoteka z izvorno kodo, ki jo oddajaš, ne sme biti daljša od 30 KB. 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 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. Nato 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.

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/`):

- V pascalu:

```
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. {PoskusnaNaloga}
```

- V C-ju:

```
#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;
}
```

- V C++:

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

- V javi:

```
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();
  }
}
```

- V C#:

```
using System.IO;
class Program
{
  static void Main(string[] args)
  {
    StreamReader fi = new StreamReader("poskus.in");
    string[] t = fi.ReadLine().Split(' '); fi.Close();
    int i = int.Parse(t[0]), j = int.Parse(t[1]);
    StreamWriter fo = new StreamWriter("poskus.out");
    fo.WriteLine("{0}", 10 * (i + j)); fo.Close();
  }
}
```

NALOGE ZA TRETJO SKUPINO

1. PINi (`pini.in`, `pini.out`)

Za večjo varnost pri uporabi številskih gesel (PINov) je koristno ta gesla redno spreminjati. Recimo, da si izberemo novo geslo na podlagi starega po naslednjem postopku: izračunamo vsoto števk starega gesla; nato pa zadnjo števko te vsote dodamo na konec gesla, prvo števko gesla pa pobrišemo. Tako na primer pri geslu 1979 izračunamo $1 + 9 + 7 + 9 = 26$ in dobimo novo geslo 9796.

Napiši program, ki za dan začetni PIN ugotovi naslednje stvari:

(a) Če PIN vsak dan zamenjamo po zgoraj opisanem postopku, čez koliko dni bomo izbrali PIN, ki smo ga nekoč pred tem že uporabljali?

(b) Pri koliko drugih PINih bi se ponavljanje začelo prej kot pri danem začetnem PINu?

(c) Koliko največ prijateljev bi lahko uporabljalo tak sistem, ne da bi pri tem kdo od njih kdaj prišel do PINa, ki ga je pred tem uporabil že kdo drug?

Vhodna datoteka: v prvi vrstici je celo število n, ki pove dolžino PINov pri tem testnem primeru. Veljalo bo $1 \leq n \leq 6$. V drugi vrstici je neko zaporedje n števk, ki pove začetni PIN.

Izhodna datoteka: vanjo izpiši tri cela števila, vsako v svojo vrstico; prvo število naj bo odgovor na vprašanje (a), drugo na vprašanje (b) in tretje na vprašanje (c).

Primer vhodne datoteke:

```
4
1979
```

Pripadajoča izhodna datoteka:

```
1560
640
12
```

Še en primer:

```
4
0606
```

Pripadajoča izhodna datoteka:

```
312
16
12
```

2. Berberi (`berberi.in`, `berberi.out`)

V 11. stoletju se je mogočna berberska vojska podala na osvajalne pohode na vzhod. Na začetku so napredovali hitro in brez težav prodrli prav daleč; z leti pa so se odnosi med vodilnimi oficirji začeli krhati. Nesoglasja so dosegla vrhunec leta 1125, ko sta se dva velika voditelja, Yusuf ibn Tashfin in Abu Abd Allah Muhammad ibn Tumart, sprla in odšla vsak po svoje: Yusuf na sever, Muhammad pa na jug. Kot to pritiče vojaškim vodjem, je vsakemu sledil tudi del vojske; del pa je ostal kar tam, kjer so se vsi skupaj sprli.

Zgodovina se rada ponavlja in tako se je v naslednjih letih še velikokrat zgodilo, da je ta ali ona skupina razpadla na tri dele, ta pozneje spet na tri in tako naprej. Vsak od teh razpadov je potekal zelo podobno prvemu: na mestu prepira je ostala skupinica in tam ustanovila zaselek, natanko dve drugi skupinici pa sta s tistega mesta sprti odpotovali vsaka po svoje.

Vsaka od takih skupin je pozneje ponovno razpadla ali pa se na nekem mestu ustalila in tam tudi sama ustanovila zaselek.

V vsakem zaselku so zelo ponosni na svojega vojskovodjo ob velikem razkolu leta 1125: vsak ve, ali je izšel iz skupine, ki je takrat sledila Yusufu, ali one, ki je sledila Muhammadu. Zdaj, po skoraj tisočletju, tudi zgodovinarje zanima, koliko je „Yusufovih“ in koliko „Muhammadovih“ zaselkov.

Napiši program, ki kot vhod dobi zemljevid z vrisanimi potmi vseh skupin in prešteje posebej Yusufove in posebej Muhammadove zaselke.

Vhodna datoteka. Prva vrstica vsebuje dve s presledkom ločeni celi števili, širino w in višino h zemljevida ($3 < w \leq 3000$, $3 < h \leq 3000$). Sledi h vrstic s po w znaki; vsak od znakov je pika („`.`“) ali `X`. Pika označuje prazno, nedotaknjeno pokrajino, `X` pa označuje poti berberske vojske oziroma njenih skupin.

Dve polji na zemljevidu štejeta za sosednji le, če imata skupno neko stranico. Če se torej dva znaka `X` dotikata le v vogalih, to pomeni, da se nobena skupina ni selila neposredno od enega izmed teh `X`-ov do drugega. Poti različnih skupin se niso nikoli v zgodovini dotaknile ali sekale.

Zunaj zemljevida (razen na zahodu) Berberi niso pustošili. Če torej na primer neke poti navidez „zmanjka“ na zgornjem robu zemljevida, se gotovo ne nadaljuje še dlje na sever in njen konec označuje zaselek.

Zahod je na levi strani zemljevida; v skrajnem levem stolpcu je z `X` označeno eno samo polje — tisto, kjer so Berberi čisto na začetku sploh vdrli na vzhod, na področje zemljevida. Od tam so nekaj časa potovali skupaj v ravni črti na vzhod (torej proti desni), dokler se niso v neki točki prvič razcepili — Muhammadova skupina je odšla naravnost proti jugu, Yusufova naravnost proti severu.

Izhodna datoteka. Vanjo izpiši dve števili, ločeni s presledkom: najprej število Yusufovih zaselkov in nato število Muhammadovih zaselkov.

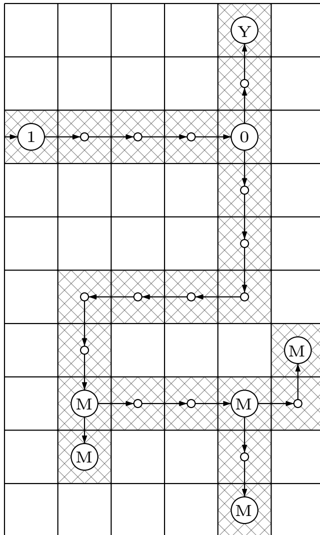
Primer vhodne datoteke:

```
6 10
....X.
....X.
XXXXX.
....X.
....X.
.XXXX.
.X...X
.XXXXX
.X..X.
....X.
```

Pripadajoča izhodna datoteka:

```
1 5
```

Razlaga zgornjega primera: naslednja slika še enkrat prikazuje zgornji primer vhoda, tokrat dopolnjen z oznakami:



- `1` — Točka, kjer je celotna vojska vdrla na vzhod.
- `0` — Veliki razkol 1125. Tu je nastal zaselek, ki ne spada ne k Muhammadu ne k Yusufu.
- `Y` — Yusufov zaselek. Njegova vojska se v tem primeru ni sprla in se je hitro ustalila.
- `M` — Muhammadovi zaselki. Dva sta nastala, ko se je vojska sprla in razcepila, trije pa, ko so se posamezne skupine ustalile.

3. Piskrc špagetov (`spageti.in`, `spageti.out`)

Koledar na steni kaže 29. marec in *chef* Iztok, dežurni kuhar na Ministrstvu za prehrano, si v grozni paniki ruva lase: bliža se zasedanje vrha Evropske unije in zvečer bo treba vse pogostiti. Povabljenih je na tone gostov, on pa ima tako malo loncev!

Da bi čim bolje izkoristil tudi tiste najmanjše piskrce, ki se prašijo v omari v kotu, je Iztok za večerjo predvidel špagete. Špageti so sila praktični — če je lonec zanje premajhen, jih prelomiš na pol, pa je. Ena sama težava je pri razpolavljanju špagetov: tako ugleden kuhar za tako prestižno večerjo špagetov ne sme nalomiti kar tako, približno, temveč mora enega po enega kar se da natanko razpoloviti z medeninastim nožičem na naoljeni laneni krpi.

Vsi špageti so vedno celoštevilskih dolžin: pri razpolavljanju špageta lihe dolžine $2k+1$ nastane en špaget dolžine k in en dolžine $k+1$, pri razpolavljanju špageta sode dolžine $2k$ pa nastaneta dva špageta dolžine k. Izjema je špaget dolžine 1, ki ga ni mogoče razpoloviti.

Za razpolavljanje bo zadolžen Iztokov pomočnik Branko, ki pa mu vihtenje nožiča še ne gre prav dobro od rok in zna v večeru razpoloviti omejeno število špagetov. Njegova naloga je, da z razpolavljanjem špagete kar čim bolj skrajša: najdaljši špaget naj ima čim manjšo dolžino, da se bodo tako lahko vsi špageti kuhali v čim manjšem piskrcu.

Napiši program, ki izračuna dolžino najdaljšega od špagetov, ki ga bo še treba spraviti v piskrc po tistem, ko z njimi opravi Branko.

Vhodna datoteka. Prva vrstica vsebuje dve celi števili, n in b, ločeni s presledkom. Pri tem je n (zanj velja $1 \leq n \leq 10^6$) število škatel špagetov, ki jih je Iztok že nakupil; b (zanj velja $1 \leq b \leq 10^9$) pa je število špagetov, ki jih je Branko zmožen razpoloviti v času, ki je še ostal do večerje.

Sledi n vrstic; i-ta med njimi vsebuje dve celi števili, m_i in d_i, ločeni s presledkom. Število m_i ($1 \leq m_i \leq 10^4$) je število špagetov v i-ti škatli, d_i pa je dolžina vsakega od špagetov iz te škatle (vsi špageti v tej škatli so enako dolgi). Veljalo bo $1 \leq d_i \leq 10^9$ — na željo evropskih birokratov namreč merimo dolžino špagetov v mikrometrih. Škatle so urejene po dolžini špagetov, tako da velja $d_1 \leq d_2 \leq d_3 \leq \ldots \leq d_{n-1} \leq d_n$.[3]

Tvoj program bomo pognali na desetih različnih vhodnih datotekah. Pri prvih štirih izmed njih bo veljalo $d_i \leq 10^6$ (za vsak i). Pri prvih dveh bo veljalo tudi $n \leq 10^4$ in $d_i \leq 10^4$ (za vsak i).

Izhodna datoteka: vanjo izpiši eno samo celo število, namreč najmanjšo možno dolžino najdaljšega špageta po tistem, ko jih Branko največ b-krat razpolovi.

Razpolavljati je dovoljeno tudi špagete, ki so že sami nastali z razpolavljanjem.

Primer vhodne datoteke:

```
5 4
12 1
4 1
2 2
1 5
2 5
```

Pripadajoča izhodna datoteka:

```
3
```

[3] Zgornje besedilo je tako, kakršno so dobili tekmovalci na samem tekmovanju. Vidimo, da omejitve teoretično dopuščajo, da imamo v vhodni datoteki skupno 10^{10} špagetov (namreč 10^6 škatel s po 10^4 špageti). Spodobilo bi se še dodati, da ni bilo mišljeno, da bi se morali tekmovalci spopadati s to možnostjo; v resnici skupno število špagetov (torej $\sum_{i=1}^{n} m_i$) pri nobenem testnem primeru ni presegalo 10^8. Ta podatek je koristen, da ne bomo v skrbeh, če bomo za štetje špagetov uporabljali 32-bitna cela števila.

4. Redki nizi (`nizi.in`, `nizi.out`)

Dan je nek niz s, dolg n znakov, ki smo ga dobili takole:

- Najprej smo vse znake niza s postavili na neko začetno vrednost, recimo c.
- Nato smo vzeli nek niz t_1 (recimo, da je dolg m_1 znakov) in nek začetni indeks z_1 (za katerega velja $1 \leq z_1 \leq n + 1 - m_1$). Z njim smo povozili del niza s z začetkom pri indeksu z_1; z drugimi besedami, i-ti znak niza t smo zapisali čez $(z_1 + i - 1)$-ti znak niza s, in to za vse i od 1 do m_1.
- Postopek iz prejšnje točke smo ponovili še za več drugih nizov $t_2, t_3, \ldots, t_k$, ki so imeli vsak svoj začetni indeks $z_2, z_3, \ldots, z_k$ in vsak neko svojo dolžino.

Primer: recimo, da imamo $n = 20$, $c =$ `x`, $k = 3$ in da po vrsti uporabimo naslednje nize: najprej $t_1 =$ `abcfg` z začetnim indeksom $z_1 = 10$; nato $t_2 =$ `fghxjx` z začetnim indeksom $z_2 = 7$; in nazadnje še $t_3 =$ `xfg` z začetnim indeksom $z_3 = 17$. Tako dobimo po vrsti:

```
xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxabcfgxxxxxx
xxxxxxfghxjxfgxxxxxx
xxxxxxfghxjxfgxxxfgx
```

Tako smo torej dobili $s =$ `xxxxxxfghxjxfgxxxfgx`.

Napiši program, ki prebere podatke o nizu s in nato za drug niz p ugotovi, kolikokrat se p pojavlja kot strnjen podniz niza s. Če bi na primer v nizu s iz gornjega primera iskali $p =$ `xfg`, bi videli, da se pojavlja trikrat.

Vhodna datoteka: v prvi vrstici sta celi števili n in k, ločeni s presledkom. V drugi vrstici je znak c. Sledi k vrstic; i-ta med njimi vsebuje najprej celo število z_i, nato presledek in nato niz t_i. Sledi še ena vrstica, ki vsebuje niz p. Veljalo bo: $1 \leq n \leq 10^9$; $1 \leq k \leq 10000$; skupna dolžina vseh nizov t_i je največ 10^6; niz p je dolg vsaj 1 in kvečjemu 100 znakov. Vsi znaki v nizih so male črke angleške abecede.

Pri petih od desetih testnih primerov bo veljalo $n \leq 10^6$.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer število pojavitev niza p kot strnjenega podniza v nizu s.

Primer vhodne datoteke:

```
20 3
x
10 abcfg
7 fghxjx
17 xfg
xfg
```

Pripadajoča izhodna datoteka:

```
3
```

5. Obračanje barv (`toggle.in`, `toggle.out`)

Chuck Norris can create a rock so heavy that even he can't lift it. And then he lifts it anyways, just to show you who the fuck Chuck Norris is.

Imamo tabelo velikosti $w \times h$, v kateri je vsaka celica pobarvana z belo ali črno barvo. Dovoljena operacija na tej tabeli je, da si izberemo neko vrstico ali stolpec in vsem celicam v njej ali njem zamenjamo barvo. Če so bile prej bele, so potem črne in obratno.

Napiši program, ki ugotovi, kolikšno je najmanjše število črnih polj, ki jih lahko dobimo s temi operacijami, in kolikšno je najmanjše število potrebnih operacij, da to dosežemo.

Vhodna datoteka: v prvi vrstici sta celi števili w (širina tabele) in h (višina tabele), ločeni s presledkom. Zanju velja $1 \leq \min\{w, h\} \leq 21$ in $\max\{w, h\} \leq 50$. Sledi h vrstic, ki opisujejo začetno stanje tabele; vsaka od teh vrstic vsebuje w znakov, pri čemer pika („`.`“) pomeni belo polje, znak „`#`“ pa črno.

Pri petih od desetih testnih primerov bo veljalo $\max\{w, h\} \leq 10$.

Izhodna datoteka: vanjo izpiši dve celi števili, ločeni z enim presledkom. Prvo število naj bo najmanjše število črnih polj, ki ga je mogoče pri dani vhodni tabeli doseči z operacijami, kakršne opisuje naloga; drugo število pa naj bo najmanjše število operacij, v katerih lahko pridemo do tega števila črnih polj.

Primer vhodne datoteke:

```
3 4
.#.
#.#
.#.
###
```

Pripadajoča izhodna datoteka:

```
1 3
```

Pri tem primeru vidimo, da lahko s tremi operacijami (obrniti moramo srednji stolpec ter drugo in četrto vrstico) dosežemo, da je v tabeli le eno črno polje (srednje polje v četrti vrstici).

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. Izpis HTMLja

Mislimo si preprost dokument v jeziku HTML — poleg besedila vsebuje še oznake oblike `<ime>` in `</ime>`, pri čemer je „`ime`“ ime enega od elementov jezika HTML. (Jezik HTML dovoli v oznakah sicer tudi še presledke in atribute, v besedilu pa komentarje, vendar predpostavimo, da v naših dokumentih teh reči ne bo.)

Napiši program, ki prebere s standardnega vhoda (ali pa datoteke) nek takšen dokument v jeziku HTML, in za tisti del dokumenta, ki leži med oznakama `<body>` in `</body>`, izpiše vse besedilo (ne pa tudi oznak). Pri izpisu naj tudi skoči v novo vrstico na vsakem takem mestu, kjer se v vhodnem dokumentu pojavi oznaka `<br>` oziroma `<br/>`.

2. Popravilo ograje

Odkar se je na Kranjskem razcvetel turizem in imajo zato veliko imenitnih obiskovalcev iz tujine, Kranjci veliko bolj skrbijo za svojo samopodobo. Obnovili so že vsa pročelja hiš, sedaj pa so se lotili še cest. Med drugim bodo zamenjali zaščitno ograjo, ki je že precej poškodovana. Ker so nekoliko ekonomični, želijo zamenjati samo poškodovane dele ograje.

Ograja je sestavljena iz deščic dolžine 1 m, ki so druga za drugo pritrjene na stebričke ob cesti. Te deščice tvorijo eno dolgo neprekinjeno ograjo, zato so jih označili s števili od 1 do n. (k-ta deščica se tako na levem koncu stika s $(k-1)$-vo deščico, na desnem pa s $(k+1)$-vo deščico. Izjemi sta le prva in zadnja deščica, ki imata samo po eno sosedo.)

Ograjo so natančno pregledali in naredili seznam oznak poškodovanih deščic. Ko so hoteli naročiti nove deščice, s katerimi bi zamenjali poškodovane, so ugotovili, da so v trgovini na voljo le deske dolžine k metrov ($k > 1$). Te sicer lahko tudi razrežejo, vendar bo od obeh kosov uporaben samo eden — desko lahko torej skrajšajo, ne morejo je pa razdeliti na dve.

Ograjo bodo menjali tako, da bodo iz ograje odstranili l ($l \leq k$) zaporednih metrskih deščic (med njimi so lahko tudi nepoškodovane, ki bodo pri ostranjevanju žal uničene) in jih nadomestili z novo desko (ki jo bodo po potrebi skrajšali na l metrov). Zanima jih najmanjše število desk, ki jih morajo kupiti, da bodo obnovili celo ograjo.

Vhodna datoteka: v prvi vrstici so tri cela števila, n, k in c, ločena s po enim presledkom. Pri tem je n dolžina ograje, k dolžina nadomestnih desk, c pa število poškodovanih odsekov. Sledi c vrstic z oznakami poškodovanih odsekov (oznake so cela števila od 1 do n in so podane v naraščajočem vrstnem redu). Veljalo bo: $1 \leq n \leq 1\,000\,000$, $1 < k \leq 1\,000\,000$ in $0 \leq c \leq n$.

Izhodna datoteka: vanjo naj tvoj program izpiše število deščic, ki jih je potrebno kupiti.

Primer vhodne datoteke:

```
10 4 6
1
3
4
5
7
9
```

Pripadajoča izhodna datoteka:

```
3
```

Ograjo s tega primera lahko popravimo tako, da kupimo tri deske; prvo desko skrajšamo na dolžino 1 in z njo zamenjamo odsek 1; drugo desko skrajšamo na dolžino 3 in z njo zamenjamo odseke 3, 4 in 5; tretjo desko pa skrajšamo na dolžino 3 in z njo zamenjamo odseke 7, 8 in 9 — odsek 8 sicer ni poškodovan, vendar s tem ni nič narobe. Le z dvema deskama dolžine 4 pa se ograje ne da obnoviti, čeprav je poškodovanih samo 6 odsekov na ograji.

NEUPORABLJENE NALOGE IZ LETA 2006

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 1. tekmovanjem IJS v znanju računalništva (leta 2006), pa jih potem na tistem tekmovanju nismo uporabili (ker se nam je nabralo več predlogov nalog, kot smo jih potrebovali za tekmovanje). Ker tudi te neuporabljene naloge niso nujno slabe, jih zdaj objavljamo v letošnjem biltenu, če bodo komu mogoče prišle prav za vajo. Poudariti pa velja, da niti besedilo teh nalog niti njihove rešitve (ki so na str. 88–117) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Palindromni stavki

Palindromni stavek je stavek, za katerega velja naslednje: če v njem pobrišemo vse nečrkovne znake (na primer presledke, ločila, števke itd.), nato pa njegove besede staknemo skupaj v en sam dolg niz, dobimo niz, ki se z desne proti levi bere enako kot z leve proti desni.

Nekaj primerov: „`PERICA REZE RACI REP.`“ in „`PE:RICA REZE---RACI/REP`“ sta palindromna stavka; „`PERICA REZE RACI KLJUN`“ pa ni palindromni stavek.

Napiši podprogram

```
function JePalindromniStavek(S: string): boolean;
```

ali, v C/C++,

```
bool JePalindromniStavek(const char *S);
```

ki vrne `true`, če je dani niz `S` palindromni stavek, sicer pa vrne `false`. Da bo naloga lažja, lahko predpostaviš, da se v stavku pojavljajo le ločila, presledki in velike črke angleške abecede.

2. Živi in mrtvi

> Wherever the living are, the dead will be there too. [...] Already in Iceland the dead outnumber us, for we're the third generation.
>
> Margaret Elphinstone, *The Sea Road*

Neka skupina ljudi je ustanovila novo naselbino in v njej med drugim tudi vodijo podatke o rojstvih in smrtih vseh prebivalcev. Znano je, koliko prebivalcev je imela naselbina ob ustanovitvi; odtlej pa je za vsako leto njenega obstoja znano, koliko ljudi se je tisto leto v naselbini rodilo in koliko jih je umrlo. (Predpostavimo, da naselbina od ustanovitve naprej ni imela nikakršnih stikov z zunanjim svetom — ni priseljevanja, odseljevanja ipd.)

Napiši program, ki za vsako izmed prvih sto let obstoja naselbine ugotovi, kaj od naslednjega trojega velja za tisto leto: (1) zagotovo je bilo v tej naselbini v vsakem trenutku tega leta več živih kot mrtvih; (2) zagotovo je bilo v tej naselbini v vsakem trenutku tega leta več mrtvih kot živih; (3) ni mogoče zagotovo sklepati

niti na (1) niti na (2), ker ne poznamo točnih časov posameznih rojstev in smrti znotraj leta.[4]

Predpostavi, da so na voljo naslednje funkcije, ki povedo podatke o naselbini:

```
function ZacetnoSteviloPrebivalcev: integer; external;
function SteviloRojstev(Leto: integer): integer; external;
function SteviloSmrti(Leto: integer): integer; external;
```

oz. v C/C++:

```
extern int ZacetnoSteviloPrebivalcev;
extern int SteviloRojstev(int leto);
extern int SteviloSmrti(int leto);
```

Leta se štejejo s celimi števili od 1 naprej.

3. Lomljenje besedila

Da lahko besedilo prikažemo na ozkem zaslončku mobilnega telefona, ga moramo razdeliti v več vrstic in to tako, da ni nobena vrstica širša od širine zaslona. Pri tem poznamo tako širino zaslona kot širino posameznih znakov besedila. Širina vrstice besedila je definirana kot vsota širin vseh znakov te vrstice.

Dogovorimo se, da lahko novo vrstico začnemo le pri presledku, torej nikoli ne razdelimo posamezne besede med dve ali več vrstic. Ko začnemo novo vrstico, pobrišemo presledke na koncu prejšnje in začetku naslednje vrstice.

Napiši program, ki prebere besedilo s standardnega vhoda (zapisano je kot ena sama dolga vrstica) in na standardni izhod izpiše isto besedilo, razbito na vrstice tako, da nobena ni širša od širine zaslona. Predpostaviš lahko, da sta ti na voljo naslednji funkciji:

```
function SirinaZaslona: integer; external;              { Vrne širino zaslona v pikslih. }
function SirinaZnaka(c: char): integer; external;       { Vrne širino znaka c v pikslih. }
```

Širino znakov in širino zaslona merimo v pikslih, torej slikovnih elementih (najmanjša točka, ki jo lahko nadziramo pri prikazu na zaslonu). Predpostaviš lahko, da ni nobena beseda v vhodnem besedilu širša od širine zaslona in da nima nobena beseda več kot 100 znakov.

Še deklaraciji v C/C++:

```
extern int SirinaZaslona();
extern int SirinaZnaka(char c);
```

[4]Bolj formalno, pa tudi malo manj morbidno, lahko nalogo formuliramo takole. Recimo, da je bilo na začetku nekega leta živih z ljudi, mrtvih pa m, in da se je v tistem letu r ljudi, umrlo pa jih je s. Oglejmo si vsa zaporedja oblike $\rho = (a_1, \ldots, a_{r+s})$, pri čemer je r členov zaporedja enakih 1, s členov pa je enakih -1. Za vsako tako zaporedje ρ definirajmo še dve zaporedji z^ρ in m^ρ, definirani s formulami $z_0^\rho = z$, $m_0^\rho = m$, $z_{t+1}^\rho = z_t^\rho + a_t$ in $m_{t+1}^\rho = m_t^\rho - \min\{a_t, 0\}$. Zaporedje ρ je *veljavno*, če za vsak t od 0 do $r+s-1$ velja $z_t^\rho > 0$, na koncu zaporedja pa velja $z_{r+s}^\rho \geq 0$. O možnosti (1) govorimo v primeru, ko za vsako veljavno zaporedje ρ in za vsak t od 0 do $r+s$ velja $z_t^\rho > m_t^\rho$, o možnosti (2) pa v primeru, ko za vsako veljavno ρ in vsak t velja $z_t^\rho < m_t^\rho$. Če ni izpolnjen noben od teh dveh pogojev, rečemo, da velja možnost (3). Izziv naloge je v tem, da je veljavnih zaporedij ρ veliko (tja do $\binom{r+s}{r}$) in bi radi našli postopek, ki ne bi zares pregledoval vseh možnih ρ.

4. Taborniki (I)

Tabornikom se je sesul trdi disk na njihovem strežniku `Obrocek`, kjer so gostovali tudi vsi taborniški obveščevalni spiski (*mailing lists*). Ko so pregledovali škodo, so ugotovili, da so nekateri obveščevalni spiski izginili. Pri pregledu varnostnih kopij, so z grozo ugotovili, da jim konfiguracijske datoteke posameznih obveščevalnih spiskov manjkajo. V svoje veliko olajšanje pa so videli, da je seznam arhivov ostal nedotaknjen.

Sistemski administrator je tako v dve tekstovni datoteki spravil dva seznama.

1. Seznam obveščevalnih spiskov iz arhiva (`seznam-arhiv.txt`), v katerem je bilo ime vsakega obveščevalnega spiska izpisano v svoji vrstici.

2. Seznam trenutno delujočih obveščevalnih spiskov (`seznam-delujoci.txt`), ki je narejen enako kot zgornji.

Napiši program, ki na podlagi podatkov iz obeh datotek izpiše seznam manjkajočih obveščevalnih spiskov. Predpostaviš lahko, da so imena obveščevalnih spiskov v obeh datotekah že urejena po abecedi in da nobeno ime ni prazen niz niti ni daljše od 100 znakov. Če ne znaš drugače, lahko predpostaviš tudi, da v nobeni datoteki ni več kot tisoč imen.

5. Nerimska števila

Pri tej nalogi bomo predstavljali števila z nizi velikih črk angleške abecede. Za posamezno črko imamo predpisano vrednost (celo število, večje od 0) in to tako, da ima črka `A` vrednost 1, vsaka naslednja črka abecede pa ima vrednost, ki je večkratnik vrednosti prejšnje črke.

Vrednost nekega niza črk definirajmo preprosto kot vsoto vrednosti posameznih črk v njem.

Niz znakov je *palindrom*, če se bere z desne proti levi enako kot z leve proti desni.

Napiši podprogram, ki za dano celo število n (večje od 0) izpiše najkrajši tak palindromni niz, ki ima vrednost n. Če je možnih več enako kratkih nizov, je vseeno, katerega izpišeš.

Tvoj podprogram naj bo oblike

```
procedure NajkrajsiPalindrom(n: integer);
```

Predpostaviš lahko, da je na voljo funkcija

```
function Vrednost(c: char): integer; external;
```

ki ti daje podatke o vrednosti posameznih črk (od `'A'` do `'Z'`).

Še deklaraciji v C/C++:

```
extern int Vrednost(char c);
void NajkrajsiPalindrom(int n);
```

Primer: če imamo vrednosti črk `A` = 1, `B` = 5, `C` = `D` = ... = `Z` = 10, je niz `AACCCBCCCAA` primer najkrajšega palindromnega niza pri številu $n = 69$.

6. Taborniki (II)

Tabornikom se je sesul trdi disk na njihovem strežniku `Obrocek`, kjer so gostovali tudi vsi taborniški obveščevalni spiski (*mailing lists*). Ugotovili so, da so izginili podatki o članih vseh izgubljenih obveščevalnih spiskov, ostal pa jim je dnevnik vpisov/izpisov z obveščevalnih spiskov. Pomagajte administratorju, da bo na podlagi dnevniških zapisov obnovil stanje članov na posameznem obveščevalnem spisku.

Dnevniška datoteka je sestavljena iz več vrstic, v vsaki vrstici pa je po en zapis naslednje oblike:

```
datum-in-ura|ime-obveščevalne-liste|akcija|e-mail-naslov
```

Pri tem je „`akcija`“ lahko „`subscribe`“ (tak zapis navaja ime novega člana obveščevalnega spiska) ali „`unsubscribe`“ (tak zapis pa pove, da je dani uporabnik izstopil iz tega obveščevalnega spiska). Zapisi so urejeni naraščajoče po času.

Napiši program, ki od uporabnika zahteva naziv obveščevalnega spiska, potem pa na podlagi dnevnika izpiše trenutne člane.

7. Nezanesljivi golobi

A in B si izmenjujeta sporočila s pomočjo golobov: A pripravi sporočilo, ga zapiše na listek ter ga nato ovije golobu okrog noge. Golob poleti proti B-ju, ki prevzame sporočilo.

Ker pa so golobi nezanesljivi (ptičja gripa, golobice, ...), se rado zgodi, da sporočila niso dostavljena.

Predlagaj postopek, po katerem naj se A in B ravnata, da bo dostava sporočil zanesljivejša, torej da bo B izvedel, kar mu A namerava sporočiti.

Pozor: če je A isto sporočilo poslal večkrat (iz kateregakoli razloga), mora biti B zmožen to zaznati, da lahko podvojeno dostavljena sporočila ignorira.

8. Začetnice

Dan je nek niz s dolžine $n > 0$ črk. **Opiši postopek**, ki prebere neko besedilo in poišče skupine n zaporednih besed, za katere velja, da če vzameš prvo črko vsake od teh besed in jih stakneš skupaj, dobiš ravno niz s. Predpostaviš lahko, da se v nizu s in v besedilu pojavljajo le male črke angleške abecede, v besedilu pa še presledki.

9. Ločevanje slik

Danih je n sivinskih slik; vse so enako velike: vsaka je razdeljena na $\mathsf{Sirina} \times \mathsf{Visina}$ pikslov, barva (oz. svetlost) posameznega piksla pa je opisana s celim številom od 0 do 255.

Radi bi našli dva taka položaja pikslov na sliki — torej para (x_1, y_1) in (x_2, y_2) — za katera velja, da se nobeni dve sliki ne ujemata v barvi obeh teh dveh pikslov. Z drugimi besedami, katerikoli dve izmed naših n slik pogledamo, se morata razlikovati ali po barvi piksla (x_1, y_1) ali po barvi piksla (x_2, y_2) ali pa celo po obeh.

Opiši postopek, ki za dano skupino slik poišče dva taka para koordinat ali pa ugotovi, da primernega para sploh ni. Če je možnih parov več, je vseeno, katerega najde.

Lahko si pomagaš z naslednjimi deklaracijami:

```
const Visina = ...; Sirina = ...; n = ...;
type byte = 0..255;
type SlikaT = array [1..Visina, 1..Sirina] of byte;
var Slike: array [1..n] of SlikaT;
```

10. Prepoznavanje ulomka

Vsako realno število $0 \leq x < 1$ lahko zapišemo v neskončnem desetiškem zapisu, kot na primer $\pi = 3{,}14159265358979323846264338\ldots$ Nekatera števila imajo dva zapisa, denimo $1/2 = 0{,}5000\ldots$ in $1/2 = 0{,}4999\ldots$ V tem primeru vedno izberemo zapis z neskončnim zaporedjem ničel.

Če zapis na nekem mestu odrežemo, dobimo končni približek (zadnje števke ne zaokrožamo navzgor). Seveda obstaja več števil, ki se začnejo z danim končnim približkom. **Napiši program**, ki poišče tak ulomek s čim manjšim imenovalcem, da se njegov desetiški zapis začne tako kot dani končni približek. Na primer, če imamo približek 0,14, so nekateri možni ulomki

$$
\begin{aligned}
130/911 &= 0{,}1427003293084\ldots \\
14/100 &= 0{,}1400000000000\ldots \\
14/99 &= 0{,}1414141414141\ldots \\
13/90 &= 0{,}1444444444444\ldots \\
7/50 &= 0{,}1400000000000\ldots \\
1/7 &= 0{,}1428571428571\ldots
\end{aligned}
$$

Najmanjši imenovalec ima v tem primeru ulomek 1/7 (od tistih z imenovalcem, manjšim od 7, se v desetiškem zapisu nobeden ne začne na 0,14).

Vhodna datoteka: vsebuje eno vrstico, v kateri je zapisan končni približek desetiškega zapisa nekega števila $0 \leq x < 1$. Celi del je torej vedno enak 0, sledi mu decimalna pika (ne vejica!) in nato še zaporedje števk. Desno od decimalne pike je vsaj ena števka in nikoli več kot šest števk.

Izhodna datoteka: vanjo napiši števec in imenovalec ulomka s čim manjšim možnim imenovalcem, tako da se desetiški zapis ulomka ujema s tistim iz vhodne datoteke v vseh podanih števkah. Števec in imenovalec naj bosta ločena z enim presledkom (ne znakom `/`).

Primer vhodne datoteke:

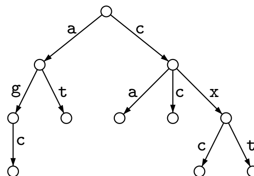
```
0.1655
```

Možna pripadajoča izhodna datoteka:

```
22 133
```

11. Presek dreves

Mislimo si drevo, ki ima na povezavah male črke angleške abecede:



Dogovorimo se še, da ne gresta iz nobenega vozlišča dve povezavi z isto črko in da so povezave, ki izhajajo iz posameznega vozlišča, urejene po abecedi.

Če začnemo v korenu drevesa in se potem spuščamo navzdol po drevesu, lahko v mislih stikamo črke na prehojenih povezavah in tako dobimo nek niz. Če lahko nek niz dobimo na ta način, bomo rekli, da ta niz *pripada* temu drevesu. Drevesu na zgornji sliki na primer pripadajo nizi `a`, `ag`, `agc`, `at`, `c`, `ca`, `cc`, `cx`, `cxc`, `cxt` in prazen niz.

Napiši program, ki prebere opise nekaj dreves in izpiše vse nize, ki pripadajo vsem tem drevesom.

V vhodni datoteki je vsako drevo predstavljeno z nekim zaporedjem znakov, ki lahko vsebuje črke in oklepaje. Drevo, ki ima eno samo vozlišče, je predstavljeno s praznim nizom. Večja drevesa so predstavljena takole: naj bo T neko tako drevo in recimo, da ima njegov koren k poddreves, i-to poddrevo označimo z T_i, do njega pa vodi povezava, na kateri je črka a_i (za $i = 1, 2, \ldots, k$). Naj bo s_i niz, ki predstavlja drevo T_i. Potem je drevo T predstavljeno z nizom `(`$a_1 s_2 a_2 s_2 \ldots a_k s_k$`)`.

Primer: drevo na gornji sliki bi bilo v vhodni datoteki predstavljeno z nizom `(a(g(c)t)c(acx(ct)))`.

V vhodni datoteki prva vrstica vsebuje celo število n ($1 \leq n \leq 100$), ki pove število dreves. V vsaki od naslednjih n vrstic je opis enega drevesa. Nobeno od teh dreves nima več kot 1000 vozlišč.

V izhodno datoteko izpiši vse take nize, ki pripadajo vsem drevesom iz vhodne datoteke. Vsakega izpiši v svojo vrstico. Če je takih nizov več, je vseeno, v kakšnem vrstnem redu jih izpišeš.

12. Kratice

Kratica je beseda, sestavljena iz prvih črk nekaj drugih besed. Včasih je kakšna od teh besed tudi sama zase kratica; v tem primeru pravimo prvotni besedi „kratica drugega reda“. Včasih je celo kakšna od besed, ki sestavljajo neko kratico, že sama kratica drugega reda; tedaj je prvotna beseda „kratica tretjega reda“. Primer:

LASER = Light Amplification by Stimulated Emission of Radiation	(kratica 1. reda)
LIGO = LASER Interferometer Gravitational Wave Observatory	(kratica 2. reda)
LSC = LIGO Software Collaboration	(kratica 3. reda)

To lahko definiramo zelo elegantno. Naj bo $A(w)$ množica besed, iz katerih je bila pridobljena kratica w. Definirajmo:

- Vsaka beseda je *kratica 0. reda.*
- w je *kratica* $(n+1)$*-vega reda* natanko tedaj, ko $A(w)$ vsebuje kakšno kratico n-tega reda.
- w je *rekurzivna kratica*, če je kratica n-tega reda za vsak $n \geq 0$.

Dana je neka množica besed W, za vsako $w \in W$ pa tudi njena $A(w)$. **Opiši postopek**, ki za vsako $w \in W$ ugotovi, ali je rekurzivna ali ni; če ni, naj ugotovi tudi njen najvišji red. (Primer rekurzivne kratice je GNU, ki je kratica za „GNU's not Unix“.)

Različica naloge lahko sprašuje po popolnih kraticah namesto po navadnih. Definirane so takole:

- Vsaka beseda je *popolna kratica 0. reda.*
- w je *popolna kratica* $(n+1)$*-vega reda* natanko tedaj, ko je $A(w)$ neprazna in so vse besede iz $A(w)$ popolne kratice n-tega reda.
- w je *popolna rekurzivna kratica*, če je popolna kratica n-tega reda za vsak $n \geq 0$.

13. Kopanje lukenj

Run, rabbit, run!
Dig that hole, forget the sun!

Nek popotnik je izgubil potni list, zato ne more več prečkati nobene meje med dvema državama. Za ladjo ali letalo nima denarja, da bi lahko prečkal kakšno morje ali ocean (pa tudi plavati ne zna). Lahko pa se še vedno prosto giblje znotraj posamezne države, če gre lahko ves čas po kopnem. Poleg tega ima tudi lopato, zato se lahko poskusi iz ene države v drugo premakniti tako, da začne kopati luknjo navpično navzdol, dokler ne pride ven na nasprotni strani Zemlje. Na ta način lahko na primer prideš iz Argentine v Čile tako, da najprej skoplješ luknjo do Kitajske, se potem malo premakneš po Kitajskem in potem od tam skoplješ luknjo do Čila.

Pomagaj našemu popotniku ugotoviti, kakšno je najmanjše število lukenj, ki jih bo moral skopati, da bo prišel iz določene države v določeno drugo državo (oz. ugotovi, da je to sploh nemogoče).

Recimo, da površje Zemlje razdelimo na $2w \times 2h$ zaplat in sicer tako, da zaplata (x, y) za $-w \leq x < w$ in $-h \leq y < h$ leži med zemljepisnima širinama $90(\frac{y}{h} - 1)$ stopinj in $90(\frac{y+1}{h} - 1)$ stopinj (negativne širine so tiste na južni polobli, pozitivne pa na severni) in med zemljepisnima dolžinama $180(\frac{x}{w} - 1)$ stopinj in $180(\frac{x+1}{w} - 1)$ stopinj (negativne dolžine so tiste zahodno od Greenwicha, pozitivne vzhodno).

Predpostavili bomo, da je vsaka zaplata bodisi v celoti zalita z morjem (in ne pripada nobeni državi) ali pa v celoti pripada eni sami državi. Iz ene zaplate lahko pridemo po površju do druge, če obe pripadata isti državi in imata skupno stranico. S kopanjem luknje pa lahko iz zaplate (x, y) pridemo do zaplate $(x - w + 2w\lfloor x/w \rfloor, -y - 1)$; seveda pa je tak premik dopusten le, če sta obe zaplati kopni.

14. Biblijski kod

Ljubitelji teorij zarote verjamejo, da se v knjigah, kot je Biblija, skrivajo tajna sporočila, do katerih se lahko dokopljemo, če na dovolj zvit način izbiramo črke iz besedila.[5] Recimo, da smo besedilo (same črke, brez ločil in presledkov) zapisali v tabelo, široko $w \leq 100$ stolpcev in visoko $H \leq 10000$ vrstic. Če si ogledamo $h \leq 30$ zaporednih vrstic te tabele, dobimo neko okno velikosti $w \times h$. Za besedo $s = s_1 s_2 \ldots s_n$ pravimo, da se *pojavlja* v tem oknu, če obstaja nek začetni položaj (x_0, y_0) in nek premik $(\triangle x, \triangle y)$, pri katerih tvori zaporedje črk na položajih $(x_0 + k\triangle x, y_0 + k\triangle y)$, $k = 0, 1, \ldots, n-1$ ravno niz s. Dan je nek slovar zanimivih besed; *ocena okna* je število, ki ga dobimo, če v oknu poiščemo vse pojavitve vseh

[5] Glej npr. članek “Bible code” v Wikipediji.

zanimivih besed in preštejemo, koliko črk okna pripada vsaj eni od teh pojavitev. **Opiši postopek**, ki za dani slovar, dano tabelo besedila in dani h ugotovi, kakšna je najvišja ocena kakšnega okna znotraj tega stolpca.

15. Izpeljava

Pri tej nalogi bomo delali z nizi znakov, vsi znaki pa so velike in male črke angleške abecede (razlika med velikimi in malimi črkami je pomembna). Dobili bomo tudi množico pravil, s katerimi lahko eno veliko črko zamenjamo z nekim nizom nič ali več črk (ki so lahko velike in/ali male). Primer:

$$\begin{array}{lcl} \texttt{A} & \to & \texttt{CeC} \\ \texttt{B} & \to & \texttt{eC} \\ \texttt{C} & \to & \texttt{ddd} \end{array}$$

Z uporabo teh treh pravil lahko iz niza `CBC` na primer dobimo niz `CeCC`, iz tega pa med drugim `CeCddd` in nato `dddeCddd` in `dddedddddd`. Do niza `dddedddddd` pa lahko pridemo tudi, če začnemo npr. z `AC`; iz tega dobimo `CeCC` in potem nadaljujemo enako kot prej.

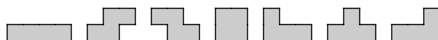
Da bo naloga lažja, se dogovorimo, da veljajo za pravila naslednje omejitve: na levi strani pravila je vedno natanko ena črka (in to velika), na desni pa nič ali več črk; za vsako veliko črko abecede obstaja največ eno pravilo, ki ima to črko na levi strani; na desni strani pravila se lahko pojavljajo poljubne male črke, od velikih črk pa samo take, ki so v abecedi kasnejše od črke na levi strani pravila (pravilo `B` $\to$ `aC` je na primer veljavno, pravilo `C` $\to$ `B` pa ne).

Opiši postopek, ki za dano množico pravil in za dani niz malih črk (recimo mu w) poišče najkrajši tak niz (recimo mu x; vsebuje lahko velike in/ali male črke), ki se ga da z uporabo pravil iz dane množice predelati v niz w.

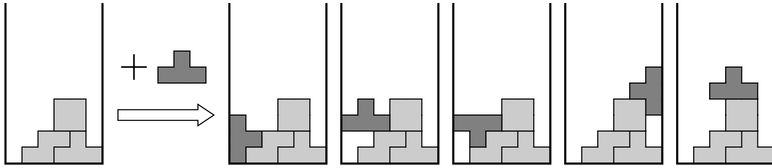
Težja različica: reši nalogo tudi za primer, ko se lahko pojavlja več pravil z isto črko na levi strani in se lahko na desni strani pravila pojavljajo poljubne velike in/ali male črke (tako da je npr. tudi `C` $\to$ `BC` možno pravilo).

16. Tetris

Imamo igralno površino širine w ($4 \le w \le 6$), v katero lahko mečemo like iz igre Tetris. Možne oblike likov so naslednje:



Dano je neko zaporedje največ 10 takšnih likov. Like moramo po vrsti metati na igralno površino; pri vsakem liku si lahko izberemo, s katere x-koordinate ga bomo vrgli; preden ga odvržemo, lahko lik tudi še vrtimo; kasneje, med padanjem, pa ga (za razliko od običajne igre Tetris) ne moremo vrteti ali premikati levo in desno. Lik vedno pade tako globoko, dokler se ne dotakne tal ali pa kakšnega že obstoječega lika. Spodnja slika kaže primer nekaj (ne pa vseh) različnih možnosti, kako lahko dodamo nov lik (pobarvan je s temnejšo barvo) na neko obstoječe stanje igralne površine:



Tako se nam liki počasi kopičijo na dnu igralne površine. Za razliko od običajnega Tetrisa se tu polne vrstice ne pobrišejo. **Opiši postopek**, ki dane like zloži na igralno površino tako, da je višina celotne dobljene „zgradbe“ čim manjša.

17. Letalski poleti

Imamo n letališč in med njimi m letalskih poletov. Vsak let je direkten (brez prestopanja) in zanj vemo, s katerega letališča poleti in na katerem pristane, poznamo pa tudi čas vzleta in pristanka.

(a) Ob nekem znanem začetnem času T_0 se nahajamo na nekem začetnem letališču s; **opiši postopek**, ki ugotovi, kako v čim krajšem času priti na neko končno letališče z. Pri tem lahko uporabimo poljubno zaporedje letov, omejitev je le ta, da mora naslednji let poleteti z istega letališča, na katerem je prejšnji let pristal, in da mora med pristankom prejšnjega in vzletom naslednjega leta miniti vsaj p minut (= čas presedanja).

(b) Recimo, da je za vsak let znana tudi njegova cena. Podan je tudi najkasnejši čas (recimo mu T_1), do katerega bi radi prišli na letališče z. **Opiši postopek**, ki poišče najcenejše zaporedje letov, ki nas do tega časa pripelje na letališče z, ali pa ugotovi, da takega zaporedja letov sploh ni. (Ceno zaporedja letov definirajmo kar kot vsoto cen posameznih letov.)

18. Osem

Imamo igro s 3×3 polji, na 8 od njih so ploščice (oštevilčene od 1 do 8), eno je prazno. V prazno polje lahko na vsakem koraku premaknemo eno od sosednjih štirih ploščic. Cilj je s čim manj koraki priti do stanja, v katerem so ploščice urejene naraščajoče (po vrsticah in v vsaki vrstici od leve proti desni), prazno polje pa je v spodnjem desnem kotu. Opiši postopek, ki za dano začetno stanje igre poišče najkrajše zaporedje potez, s katerimi pridemo v ciljno stanje, ali pa ugotovi, da se ciljnega stanja sploh ne da doseči.

19. Črna škatlica

Imamo črno škatlico, ki ima vhod, izhod in en bit pomnilnika. V vsakem koraku prebere en bit vhoda in nato izpiše en bit na izhod ter mogoče tudi spremeni vsebino svojega pomnilnika. Z drugimi besedami, škatlica deluje kot takšna funkcija:

```
const Prehod: array [0..1, 0..1] of 0..1 = (...);
      Izhod: array [0..1, 0..1] of 0..1 = (...);
var q: 0..1;    { notranje stanje (1 bit pomnilnika) }
function CrnaSkatlica(Vhod: 0..1): 0..1;
var b: 0..1;
```

```
begin
   b := Izhod[q, Vhod];
   q := Prehod[q, Vhod];
   CrnaSkatlica := b;
end; {CrnaSkatlica}
```

Težava je, da ne poznamo niti začetne vrednosti spremenljivke q niti vsebine tabel Izhod in Prehod. Našo črno škatlico lahko opazujemo le tako, da kličemo funkcijo CrnaSkatlica in ji kot parameter Vhod pošiljamo bite po svoji izbiri, nato pa opazujemo, kakšne vrednosti nam vrača.

(a) **Opiši postopek**, ki črno škatlico opazuje tako dolgo in na tak način, da lahko od nekega trenutka naprej simulira njeno delovanje. Tvoj postopek naj bo torej zmožen pri vsaki črni škatlici (ne glede na to, kakšni sta njeni tabeli Izhod in Prehod ter začetna vrednost q) prej ali slej priti v položaj, ko ve o trenutnem stanju črne škatlice in njenih tabelah Izhod in Prehod dovolj, da lahko zagotovi, da bi znal od tistega trenutka naprej za poljubno zaporedje vhodnih bitov (ne le takšnih, ki si jih izbere sam) vnaprej napovedati, kaj bo pri tem zaporedju vhodov vračala črna škatlica na izhodu.

To nalogo lahko še malo posplošimo — recimo, da ima vhod n možnih vrednosti, izhod m možnih vrednosti, notranje stanje (spremenljivka q) pa k možnih vrednosti. Pri zgoraj opisani črni škatlici imamo $n = m = k = 2$.

(b) Težja različica: reši nalogo še za črno škatlico z dvema bitoma notranjega stanja, torej $n = m = 2$ in $k = 4$.

(c) Lažja različica: reši nalogo še za črno škatlico brez vhoda, torej $n = 1$, pri čemer pa sta m in k zdaj lahko tudi precej večja od 2.

REŠITVE NALOG ZA PRVO SKUPINO

1. GrbaveBesede

Vsako vrstico vhodne datoteke pregledujmo znak za znakom. Če je prejšnji znak (ki smo si ga zapomnili v spremenljivki cp) presledek, trenutni pa ne, se je začela nova beseda in si zapomnimo njen začetni položaj v vrstici (v ta namen uporabljamo spremenljivko z), števec grb v besedi (n) pa postavimo na 0. Če je prejšnji velika črka, trenutni znak pa mala, povečamo števec grb (n) v trenutni besedi za 1. Ko pridemo do konca besede (trenutni znak je presledek, prejšnji pa ne), poglejmo, če ima vsaj dve grbi, in jo v tem primeru izpišemo (vemo, da v trenutni vrstici pokriva znake na indeksih od z do trenutnega položaja). Pazimo tudi na besede, ki se ne končajo s presledkom, pač pa s koncem vrstice; na koncu vrstice torej poglejmo, če zadnji znak ni bil presledek; če ni bil in če ima zadnja beseda vsaj dve grbi, izpišimo tudi njo.

```
program GrbaveBesede;
var Vrsta: string; z, n, j: integer; cp, cc: char;
begin
  while not Eof do begin
    ReadLn(Vrsta);
    cp := ' '; n := 0; z := 1;
    for j := 1 to Length(Vrsta) do begin
      cc := Vrsta[j];
      if (cc = ' ') and (cp <> ' ') then
        begin if n >= 2 then WriteLn(Copy(Vrsta, z, j - z)) end
      else if (cc <> ' ') and (cp = ' ') then
        begin z := j; n := 0 end
      else if (cc >= 'a') and (cc <= 'z') and (cp >= 'A') and (cp <= 'Z') then
        n := n + 1;
      cp := cc;
    end; {for}
    if (cp <> ' ') and (n >= 2) then WriteLn(Copy(Vrsta, z, j - z + 1));
  end; {while}
end. {GrbaveBesede}
```

Še v C-ju:

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

int main()
{
  char cp = ' ', cc = ' ', beseda[101];
  int n = 0, d = 0, c;
  do
  {
    /* Zapomnimo si prejšnji znak in preberimo novega. */
    cp = cc; c = getchar();
    /* Konec vrstice ali datoteke obravnavajmo enako kot presledek. */
    if (c == EOF || isspace(c)) cc = ' ';
    else cc = (char) c;
    /* Ali smo na koncu besede? */
    if (cc == ' ' && cp != ' ') {
      if (n >= 2) { beseda[d] = 0; printf("%s\n", beseda); }
```

```
      n = 0; d = 0; }
    /* Ali smo pri grbi? */
    else if (isupper(cp) && islower(cc)) n++;
    /* Znake trenutne besede hranimo v tabeli beseda. */
    if (cc != ' ') beseda[d++] = cc;
  }
  while (c != EOF);
  return 0;
}
```

V perlu pa lahko nalogo rešimo v eni vrstici:

```
perl -ne '/[A-Z][a-z].*[A-Z][a-z]/ && print "$_\n" for split'
```

2. Predavalnice

(a) Sledimo namigu iz besedila naloge in upoštevajmo, da je v enem dnevu le $24 \cdot 60 = 1440$ minut. Zato si lahko privoščimo tabelo (recimo ji a) s 1440 celicami, vsaka od njih pa nam bo za eno od minut v dnevu povedala, koliko predavanj poteka tisto minuto. Čase začetka in konca predavanj pretvorimo iz ur in minut v minute po polnoči, torej npr. iz časa $hh : mm$ nastane čas $60 \cdot hh + mm$; tako so zdaj časi čisto navadna cela števila od 0 do 1440. Naj bo z_i čas začetka i-tega predavanja, k_i pa čas konca tega predavanja. Nadaljujmo po naslednjem postopku:

postavi vse celice tabele a na 0;
za vsako predavanje i:
  povečaj celice $a[z_i], a[z_i + 1], \ldots, a[k_i - 1]$ za 1;

Zdaj je za vsako minuto dneva v tabeli a podatek o tem, koliko predavanj poteka tisto minuto. Poiskati moramo torej največjo vrednost v tej tabeli; ta nam bo povedala, kolikšno je največje število hkratnih predavanj, to pa je tudi najmanjše število predavalnic, s katerimi je še mogoče izpeljati celoten dogodek.

(b) Že pri reševanju podnaloge (a) smo videli, da je koristno, če lahko za razna časovna obdobja ugotovimo, koliko predavanj hkrati je takrat v teku. Opazimo lahko, da ko se premikamo naprej po času, se število predavanj v teku spremeni le v tistih trenutkih, ko se kakšno predavanje začne ali konča. Če je bilo na primer tik pred časom t v teku pet predavanj in se nato ob času t dve predavanji končata, začnejo pa se tri nova, bo tik za časom t v teku šest predavanj. Pametno je torej čase začetka in konca predavanj pregledovati v naraščajočem vrstnem redu; tako bomo zlahka sproti popravljali število predavanj, ki so trenutno v teku.

naj bo L prazen seznam;
za vsako predavanje i:
  dodaj na L par $(z_i, 1)$ in $(k_i, -1)$;
uredi seznam L naraščajoče (po prvem elementu parov,
  če ima več parov enak prvi element, pa po drugem);
$n := 0$; $m := 0$;
za vsak zapis (t, c) v seznamu L (po naraščajočem t):
  $n := n + c$; (* *število predavanj v teku* *)
  če je $n > m$, potem $m := n$; (* *število potrebnih predavalnic* *)
izpiši m;

V n torej vzdržujemo število predavanj, ki trenutno potekajo, v m pa največjo doslej doseženo vrednost n — to je na koncu tudi rezultat, po katerem sprašuje naloga. Pri urejanju časov moramo paziti na to, da če se ob nekem trenutku nekatera predavanja začnejo, druga pa končajo, moramo konce obdelati pred začetki, saj bi sicer lahko pomotoma dobili vtis, da poteka hkrati več predavanj, kot jih v resnici.

3. Darila

Nalogo lahko rešimo z dvema gnezdenima zankama. Zunanja zanka gre po vseh osebah, pri vsaki osebi pa moramo izračunati skupno vrednost prejetih in podarjenih daril ter razliko med njima. Tidve vsoti izračunamo v notranji zanki. Največjo doslej doseženo razliko si zapomnimo (spremenljivka NajRazlika), pa tudi številko osebe, pri kateri smo jo dosegli (NajKdo).

```
program Darila;
var i, j, Razlika, NajRazlika, NajKdo: integer;
begin
  NajKdo := −1; NajRazlika := 0;
  for i := 1 to StOseb do begin
    Razlika := 0;
    for j := 1 to StOseb do
      Razlika := Razlika + Darilo(j, i) − Darilo(i, j);
    if (NajKdo < 0) or (Razlika > NajRazlika)
    then begin NajKdo := i; NajRazlika := Razlika end;
  end; {for i}
  WriteLn('Največ dobička, ', NajRazlika, ', ima oseba ', NajKdo, '.');
end. {Darila}
```

Še rešitev v C-ju:

```
#include <stdio.h>

int main()
{
  int i, j, razlika, najRazlika = 0, najKdo = −1;
  for (i = 1; i <= StOseb(); i++)
  {
    for (razlika = 0, j = 1; j <= StOseb(); j++)
      razlika += Darilo(j, i) − Darilo(i, j);
    if (najKdo < 0 || razlika > najRazlika)
      najKdo = i, najRazlika = razlika;
  }
  printf("Največ dobička, %d, ima oseba %d.\n", najRazlika, najKdo);
  return 0;
}
```

Mogoče je, da je maksimalni dobiček poleg človeka NajKdo dosegel še kdo drug. Naloga pravi, da je v tem primeru vseeno, koga od teh ljudi izpišemo; naša rešitev izpiše med dobitniki maksimalnega dobička tistega z najmanjšo številko. Če bi hoteli izpisati vse, bi potrebovali še en prehod čez vse osebe; takrat bi za vsakega človeka pogledali, če je njegov dobiček enak NajRazlika, in ga po potrebi izpisali. (Dosežene dobičke pri vsaki osebi pa bi si lahko shranili v kakšno tabelo, da jih ne bi bilo treba računati dvakrat.)

4. Elektronska ključavnica

V spremenljivki Natipkano bomo šteli, koliko začetnih števk gesla je uporabnik doslej natipkal. Ko uporabnik pritisne eno od števk, preverimo, če je to ravno naslednja števka gesla; če se ne ujemata, to pomeni, da se je uporabnik zatipkal, kar si zapomnimo tako, da Natipkano postavimo na -1 in odtlej ignoriramo vse števke, ki jih natipka, dokler naslednjič ne pritisne tipke „Prekliči“. Če pa je pravilno pritisnil naslednjo števko gesla, povečamo števec Natipkano za 1; če bi pri tem že dosegel dolžino gesla, vemo, da je zdaj natipkano celo geslo in lahko ključavnico odklenemo, spremenljivko Natipkano pa spet postavimo na 0. Na 0 jo seveda postavimo tudi ob pritisku na tipko „Prekliči“.

```
program ElektronskaKljucavnica;

   var Geslo: string; external;
   procedure Odkleni; external;
   function PreberiTipko: char; external;

var Natipkano: integer; c: char;
begin
   Natipkano := 0;
   while true do begin
      c := PreberiTipko;
      if c = 'P' then Natipkano := 0
      else if Natipkano >= 0 then
         { Preverimo, če je naslednja števka prava. }
         if Geslo[Natipkano + 1] <> c then
            Natipkano := -1 { Uporabnik se je zatipkal. }
         else if Natipkano < Length(Geslo) - 1 then
            Natipkano := Natipkano + 1
         else { Uporabnik je napisal že celotno geslo; odklenimo vrata. }
            begin Odkleni; Natipkano := 0 end
   end; {while}
end. {ElektronskaKljucavnica}
```

Še rešitev v C-ju:

```
#include <stdio.h>

   extern const char *geslo;
   extern void Odkleni();
   extern char PreberiTipko();

int main()
{
   int natipkano = 0; char c;
   for ( ; ; )
   {
      c = PreberiTipko();
      if (c == 'P') natipkano = 0;
      else if (natipkano >= 0) { /* Je naslednja števka prava? */
         if (c != geslo[natipkano])
            natipkano = -1; /* Uporabnik se je zatipkal. */
         else if (! geslo[++natipkano])
            { Odkleni(); natipkano = 0; } }
   }
}
```

5. Kdo je izdal skrivnost?

(a) Ker so telefonske številke pri tej nalogi le štirimestne, je možnih samo 10 000 telefonskih številk. V neki tabeli (v spodnjem podprogramu je to Obvescen) si bomo za vsako številko zapomnili, ali obstaja primerno zaporedje klicev, s katerim bi lahko lastnik te številke dobil informacijo od lastnika številke a. Telefonske pogovore pregledujmo po naraščajočem času; pri pogovoru med u in v upoštevamo, da če je pred tem pogovorom eden od njiju že imel a-jevo informacijo (kar vidimo v tabeli Obvescen), jo bo imel po pogovoru tudi drugi (in tudi to vpišemo v tabelo Obvescen). Na koncu tega postopka imamo v Obvescen[b] podatek o tem, ali je a-jeva informacija prišla tudi do b-ja ali ne.

Potencialna izboljšava je še ta, da čim med pregledovanjem seznama pogovorov opazimo, da je informacija dejansko lahko prišla do b-ja, prenehamo z delom, saj smo že dobili odgovor, ki nas zanima.

```
function AliObstajaZaporedjeKlicev(a, b: integer): boolean;
const n = 10000;
var Obvescen: array [0..n − 1] of boolean; u, v: integer;
begin
  for u := 0 to n − 1 do Obvescen[u] := false;
  Obvescen[a] := true;
  while not Eof do begin
    ReadLn(u, v);
    if Obvescen[u] then Obvescen[v] := true;
    if Obvescen[v] then Obvescen[u] := true;
    if Obvescen[b] then break;
  end; {while}
  AliObstajaZaporedjeKlicev := Obvescen[b];
end; {AliObstajaZaporedjeKlicev}
```

Ali, v C-ju:

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

#define n 10000

bool AliObstajaZaporedjeKlicev(int a, int b)
{
  bool obvescen[n]; int u, v;
  for (u = 0; u < n; u++) obvescen[u] = false;
  obvescen[a] = true;
  while (scanf("%d %d", &u, &v) == 2)
  {
    if (obvescen[u]) obvescen[v] = true;
    if (obvescen[v]) obvescen[u] = true;
    if (obvescen[b]) break;
  }
  return obvescen[b];
}
```

(b) Če hočemo poiskati najkrajše zaporedje klicev, s katerim lahko informacija pride od a do b, ni dovolj, če si za vsako številko zapomnimo, ali obstaja kakšno zaporedje klicev od a do b, pač pa si moramo zapomniti tudi dolžino najkrajšega takega zaporedja (tabela StKlicev) in pa podatek o tem, od koga je lastnik te številke prejel

a-jevo informacijo (tabela Predhodnik). Ker imamo le $n = 10000$ oseb, bo najkrajše zaporedje klicev od a do b (če sploh obstaja) dolgo največ 9999 klicev (po več kot toliko klicih bi se začeli ljudje v zaporedju že ponavljati, kar pomeni, da bi nekaj vmesnih klicev lahko opustili in dobili še krajše zaporedje). Zato lahko na začetku vse elemente tabele StKlicev postavimo na n in to vrednost uporabljamo kot znak, da zaporedja klicev do neke ciljne številke sploh še nismo našli.

Če pri pregledovanju spiska pogovorov opazimo klic med u in v in npr. vemo, da obstaja zaporedje treh klicev od a do u, potem zdaj vemo tudi, da obstaja zaporedje štirih klicev od a do v in da je v-jev predhodnik v tem zaporedju lastnik številke u. To možnost si zapomnimo, če doslej do v-ja še nismo poznali nobenega zaporedja z manj kot štirimi klici.

Postopka ne smemo prekiniti takoj, ko najdemo neko zaporedje klicev do b (kot smo storili pri podnalogi (a)), ker ni nujno, da je to zaporedje že najkrajše — mogoče bomo kasneje našli še kakšno krajše.

Na koncu uporabimo tabelo Predhodnik, da rekonstruiramo zaporedje telefonskih številk, prek katerih je informacija prišla od a do b. To zaporedje pravzaprav dobimo v obrnjenem vrstnem redu, od b proti a; če ga hočemo izpisati v takem vrstnem redu, v kakršnem se je informacija širila od a do b, si ga lahko najprej shranimo v neko tabelo (v spodnjem podprogramu je to Zaporedje).

```
procedure NajkrajseZaporedjeKlicev(a, b: integer);
const n = 10000;
var StKlicev, Predhodnik, Zaporedje: array [0..n − 1] of integer; u, v: integer;
begin
  for u := 0 to n − 1 do StKlicev[u] := n;
  StKlicev[a] := 0; Predhodnik[a] := a;
  while not Eof do begin
    ReadLn(u, v);
    if StKlicev[v] > StKlicev[u] + 1 then
      begin StKlicev[v] := StKlicev[u] + 1; Predhodnik[v] := u end;
    if StKlicev[u] > StKlicev[v] + 1 then
      begin StKlicev[u] := StKlicev[v] + 1; Predhodnik[u] := v end;
  end; {while}
  if StKlicev[b] = n then WriteLn(b, ' ni dobil informacije od ', a, '.')
  else begin
    u := b;
    repeat
      Zaporedje[StKlicev[u]] := u;
      u := Predhodnik[u];
    until u = a;
    Zaporedje[0] := a;
    WriteLn('Zaporedje ', StKlicev[b], ' pogovorov od ', a, ' do ', b, ':');
    for u := 1 to StKlicev[b] do
      WriteLn('- oseba ', Zaporedje[u − 1], ' pove osebi ', Zaporedje[u]);
  end; {if}
end; {NajkrajseZaporedjeKlicev}
```

Še rešitev v C-ju:

```
void NajkrajseZaporedjeKlicev(int a, int b)
{
  int stKlicev[n], predhodnik[n], zaporedje[n], u, v;
  for (u = 0; u < n; u++) stKlicev[u] = n;
```

```
   stKlicev[a] = 0; predhodnik[a] = a;
   while (scanf("%d %d", &u, &v) == 2)
   {
      if (stKlicev[v] > stKlicev[u] + 1)
         stKlicev[v] = stKlicev[u] + 1, predhodnik[v] = u;
      if (stKlicev[u] > stKlicev[v] + 1)
         stKlicev[u] = stKlicev[v] + 1, predhodnik[u] = v;
   }
   if (stKlicev[b] == n) printf("%d ni dobil informacije od %d.\n", b, a);
   else {
      u = b;
      do { zaporedje[stKlicev[u]] = u; u = predhodnik[u]; }
      while (u != a);
      zaporedje[0] = a;
      for (u = 1; u <= stKlicev[b]; u++)
         printf("- oseba %d pove osebi %d\n", zaporedje[u - 1], zaporedje[u]);
   }
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Roboti

Da se bomo lažje pogovarjali, naj bo k število robotov, ki jih hočemo zamenjati z enim samim (Marvinom).

Rešitev s tabelo $n \times n$. Preprosta rešitev, ki se bo obnesla, če n ni prevelik, je tale: pripravimo si tabelo $n \times n$ celic (recimo ji s) in v njej za vsako točko našega kvadratnega polja hranimo podatek o tem, koliko robotov jo je obiskalo. Na koncu pregledamo celo tabelo in preverimo, če je kakšno točko obiskalo vseh k robotov.

Paziti moramo na možnost, da nek robot obišče isto točko po večkrat; v tem primeru moramo njegov obisk še vedno šteti samo enkrat. Zato je koristno imeti še eno tabelo (recimo ji z), v kateri si zapomnimo, kateri je zadnji robot, ki je to celico že obiskal.

za vsako točko $(x, y) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$:
  naj bo $s[x, y] := 0$ in $z[x, y] := 0$;
za vsakega robota i od 1 do k:
  za vsako točko (x, y) na poti i-tega robota:
    če $z[x, y] \neq i$:
      povečaj $s[x, y]$ za 1 in postavi $z[x, y]$ na i;
za vsako točko $(x, y) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$:
  če je $s[x, y] = k$, izpiši (x, y) in se ustavi;

Slabost te rešitve je, da sta pri velikem n tudi tabeli veliki, verjetno veliko večji od seznamov točk, ki so jih roboti dejansko obiskali. V takih primerih je naša rešitev potratna s časom in pomnilnikom.

Rešitev s presekom seznamov. Druga možnost je, da najprej poiščemo seznam točk, ki so skupne poti prvega in drugega robota; potem med temi točkami poiščemo tiste, ki so prisotne tudi na poti tretjega robota; med temi potem poiščemo tiste, ki so prisotne tudi na poti četrtega robota; itd.

naj bo S seznam točk, ki jih je obiskal prvi robot;
za vsakega robota i od 2 do k:
  naj bo P nek prazen seznam;
  za vsako točko t na seznamu S:
    če je t prisotna tudi na seznamu točk, ki jih je obiskal robot i,
      potem dodaj t v seznam P;
  $S := P$;
izpiši seznam S;

Kako bi v tem postopku preverili, ali je neko točko t obiskal tudi robot i? Preprosta rešitev je, da pregledamo celoten seznam točk, ki jih je obiskal robot i, in vsako od njih primerjamo s t. Če so seznami v povprečju dolgi po d točk, nam bo celoten algoritem vzel v najslabšem primeru $O(kd^2)$ časa. Potencialno koristna izboljšava bi bila, da bi robote pregledovali po naraščajoči dolžini poti, tako da bi bil seznam S ves čas, tudi na začetku, čim krajši.

Sezname točk za vsakega robota bi lahko na začetku obdelave tudi uredili (npr. po x-koordinati, točke z enakim x pa po y-koordinati; po urejanju lahko tudi pobrišemo morebitne večkratne kopije ene in iste točke) in potem točko t v posameznem

seznamu iskali z bisekcijo; časovna zahtevnost (če odmislimo urejanje) bi bila tako le še $O(kd \log d)$.

Zlivanje. Toda če sezname uredimo, lahko njihove preseke računamo še hitreje, z zlivanjem:

```
naj bo S urejen seznam točk, ki jih je obiskal prvi robot;
za vsakega robota i od 2 do k:
  naj bo S' urejen seznam točk, ki jih je obiskal robot i;
  naj bo P nek prazen seznam;
  m := dolžina seznama S; m' := dolžina seznama S';
  j := 1; j' := 1;
  dokler je j ≤ m in j' ≤ m':
    naj bo t := S[j] in t' := S'[j'];
    (* Na tem mestu velja: v P smo dodali že vse tiste točke u, ki so
       prisotne tako v S kot v S' in za katere velja u < t in u < t'. *)
    če je t = t', dodaj t na konec seznama P;
    če je t ≤ t', povečaj j za 1;
    če je t' ≤ t, povečaj j' za 1;
  S := P;
izpiši seznam S;
```

Tu se torej z dvema števcema, j in j', hkrati pomikamo po obeh seznamih (S in S'), katerih presek računamo. Če vidimo v obeh seznamih isto točko ($t = t'$), jo dodamo v izhodni seznam P in se premaknemo naprej po obeh seznamih; sicer pa se premaknemo naprej le po tistem seznamu, pri katerem smo videli manjšo točko (če je npr. $t < t'$, bodo točke, ki pridejo v seznamu S' za točko t', tudi večje od t, torej točke t v seznamu S' sploh ni).

Vidimo, da nam zlivanje dveh seznamov dolžine $O(d)$ vzame le $O(d)$ časa, tako da bi celoten postopek (če odmislimo urejanje) trajal le $O(kd)$ časa.

Kako lahko uredimo točke nekega seznama? Preprosti postopki, kot sta urejanje z izbiranjem in urejanje z vstavljanjem, bi vzeli $O(d^2)$ časa, tako da bi urejanje vseh k seznamov trajalo $O(kd^2)$ časa; potem je že vseeno, če bi ostali kar pri rešitvi s pregledovanjem neurejenih seznamov. Bolje je, če si lahko privoščimo kakšnega od učinkovitejših postopkov urejanja, npr. urejanje s kopico ali quicksort. V tem primeru nam bo vsako urejanje vzelo $O(d \log d)$ časa, celoten postopek (z zlivanjem vred) pa potem $O(kd \log d)$.

Lahko pa izkoristimo dejstvo, da so naše točke opisane s celoštevilskimi koordinatami od 1 do n. Recimo, da n ni zares gromozanski in si lahko privoščimo $O(n+d)$ pomožnega pomnilnika. Potem lahko uporabimo radix sort:

```
naj bo seznam točk, ki jih urejamo, predstavljen s tabelo S[1..d];
naj bo T[1..d] neka pomožna tabela točk;
(* Najprej v tabeli p preštejmo, koliko točk ima posamezno y-koordinato. *)
za vsak y od 1 do n: naj bo p[y] := 0;
za vsak i od 1 do d: povečaj p[S[i].y] za 1;
(* V tabeli r izračunajmo, kje se bodo v urejenem seznamu
   začele točke s posamezno y-koordinato. *)
naj bo r[1] := 1;
```

```
za vsak y od 2 do n: naj bo r[y] := r[y − 1] + p[y];
(* Zdaj točke skopirajmo iz S v urejeni seznam T. *)
za vsak i od 1 do d:
  y := S[i].y;    (* y-koordinata točke S[i] *)
  skopiraj S[i] v T[r[y]] in povečaj r[y] za 1;
```

Na koncu tega postopka vsebuje T iste točke kot S, le da so urejene po y-koordinati. Opazimo lahko, da je urejanje pri tem postopku stabilno: to pomeni, da če ima več točk isto y-koordinato, bo njihov medsebojni vrstni red v izhodni tabeli T enak, kot je bil v vhodni tabeli S. Ta lastnost nam bo prišla prav v naslednjem koraku: tabelo T, ki smo jo dobili z urejanjem po y, zdaj po enakem postopku uredimo po x-koordinatah; rezultate lahko potem zapišemo v tabelo S. V tabeli S bodo zdaj točke urejene po x, če pa ima več točk enak x, bo njihov medsebojni vrstni red enak, kot je bil v T — tam pa so bile urejene po y. To je torej točno tak vrstni red, kot smo si ga zaželeli, ko smo prvič govorili o urejanju točk. Vzel pa nam je $O(n + d)$ časa in tudi $O(n + d)$ pomožnega pomnilnika.

Rešitev z radix sortom in zlivanjem seznamov nam bo torej vsega skupaj vzela $O(k(n + d))$ časa.

Rešitev z razpršeno tabelo. Na začetku smo razmišljali o rešitvi, ki v tabeli velikosti $n \times n$ šteje, koliko seznamov vsebuje določeno točko; videli smo, da pri tej rešitvi lahko porabimo neugodno veliko časa in pomnilnika za ukvarjanje s celicami, ki jih ne vsebuje sploh nobena pot. Recimo, da z neko funkcijo h pripišemo vsaki točki nek indeks v razponu od 0 do $m - 1$; na primer $h(x, y) = (ny + x) \bmod m$. Namesto tabele velikosti $n \times n$ imejmo zdaj le tabelo (recimo ji U) velikosti m in točko (x, y) hranimo v celici $U[h(x, y)]$. Ker se lahko zgodi, da pade več točk (x, y) v isti indeks $h(x, y)$, mora vsaka celica tabele U v resnici hraniti seznam vseh točk, ki so se preslikale vanjo. Z nekaj sreče se bodo točke lepo razpršile po tabeli U in bodo ti seznami kratki, četudi je m majhen (ker bomo v tabeli hranili največ d točk hkrati, bomo vzeli $m = O(d)$). Ob vsaki točki bomo hranili tudi podatek o tem, koliko robotov jo je obiskalo. Ker nas zanimajo le točke, ki so jih obiskali vsi roboti, bomo v razpršeno tabelo U dodali le točke, ki jih je obiskal prvi robot (pametno je za prvega razglasiti tistega, ki je obiskal najmanj točk); pri vseh naslednjih robotih pa bomo le povečevali števce obiska pri tisth točkah, ki so v tabeli U in ki so jih obiskali tudi ti kasnejši roboti. Na koncu pregledamo tabelo U, da vidimo, če je kakšno točko obiskalo vseh k robotov.

```
za vsak c od 0 do m − 1: naj bo U[c] prazen seznam;
za vsako točko t, ki jo je obiskal prvi robot:
  dodaj (t, 1) v seznam U[h(t)];
za vsakega robota i od 2 do k:
  za vsako točko t, ki jo je obiskal i-ti robot:
    preglej seznam U[h(t)]:
      če se na njem pojavlja par (t, r) za nek r,
        ga spremeni v (t, r + 1);
za vsak c od 0 do m − 1:
  za vsak par (t, r) na seznamu U[c]:
    če je r = k, izpiši točko t;
```

Če predpostavimo, da nam funkcija h lepo razprši točke po tabeli U in da je tabela U velika približno d celic, bodo seznami v njej dolgi povprečno po $O(1)$ elementov, zato bo časovna zahtevnost celega postopka le še $O(kd)$, poraba pomožnega pomnilnika pa $O(d)$.

2. rsync

Postopek ne deluje pravilno. Tule je primer seznama, pri katerem vrne napačen rezultat:

Tlačenka	Datoteka	Datum
1	`a.txt`	1. 1. 2008
2	`a.txt`	1. 1. 2007

Poglejmo, kaj se zgodi, ko na tem seznamu izvedemo postopek iz besedila naloge. Najprej pride (*i*) urejanje po datumu (najnovejši naprej), ki pri tem seznamu ne spremeni ničesar; pri (*ii*) urejanju po imenu datotek pa je, ko gledamo $i = 0$ in $j = 1$, pogoj v vrstici $3'$ izpolnjen: ime datoteke $T[i]$ je `a.txt`, ime datoteke $T[j]$ je `a.txt` in vsekakor drži, da `a.txt` po abecedi ni pred `a.txt`. Ker je torej ta pogoj izpolnjen, bomo v vrstici 4 tadva elementa tabele zamenjali. Tako pride različica z dne 1. 1. 2007 na prvo mesto, novejša različica pa na drugo. Korak (*iii*) potem drugo pobriše in tako nam ostane v izhodnem seznamu starejša različica datoteke `a.txt`.

Težava našega algoritma za urejanje je torej v tem, da se medsebojni vrstni red zapisov z enako vrednostjo ključa za urejanje (v našem primeru je bilo to ime datoteke) lahko pri urejanju spremeni — če je bil pred urejanjem nek zapis pred drugim (ne nujno neposredno pred drugim), oba pa sta imela enako vrednost ključa za urejanje, se lahko zgodi, da bo po urejanju drugi zapis pred prvim. Tej lastnosti rečemo na kratko, da urejanje ni *stabilno*.

Če bi uporabili stabilno urejanje, bi deloval naš postopek pravilno: v točki (*i*) uredi zapise po datumu (novejši naprej), torej za vsako datoteko pride zapis za najnovejšo različico te datoteke pred zapise za vse zgodnejše različice te datoteke. Pri urejanju zapisov po imenu datoteke v točki (*ii*) pridejo zapisi za isto datoteko skupaj; če bi bilo to urejanje stabilno, bi bil zdaj prvi zapis med njimi prav tisti, ki je bil že prej (pred urejanjem v točki (*ii*)) prvi izmed zapisov za to datoteko, to pa je, kot smo videli, ravno tisti za najzgodnejšo različico te datoteke.

Prva težava, ki pri našem urejanju povzroča nestabilnost, so zamenjave enakih elementov. Če imata $T[i]$ in $T[j]$ enako ime, bo pogoj v vrstici $3'$ izpolnjen in program ju bo zamenjal; mi pa bi želeli, da bi medsebojni vrstni red enakih elementov ostal enak. Torej bi bilo pametno sedanji pogoj „če je $T[i] \geq T[j]$, potem ju zamenjaj“ spremeniti v „če je $T[i] > T[j]$, potem ju zamenjaj“. Pri urejanju po abecedi torej korak $3'$ postane takšen:

$3''$ če je ime datoteke $T[j]$ po abecedi pred imenom datoteke $T[i]$,

Žal pa to še ni dovolj. Oglejmo si naslednji primer:

Tlačenka	Datoteka	Datum
1	`b.txt`	1. 1. 2008
2	`b.txt`	1. 1. 2007
3	`a.txt`	1. 1. 2006

Naš postopek po zaslugi popravka $3''$ sicer ne bi zamenjal prvega in drugega zapisa (kar je dobro), pač pa bi nato zamenjal prvi in tretji zapis, tako da bi novejša različica datoteke `b.txt` pristala na koncu seznama, za starejšo različico iz leta 2007. Težave s stabilnostjo nam povzročajo zamenjave na daljavo: ko zamenjamo dva elementa $T[i]$ in $T[j]$ (ker je $T[j] < T[i]$ in $j > i$), smo sicer (po popravku $3''$) lahko prepričani, da med njima (v celicah $T[i+1], \ldots, T[j-1]$) ni nobenega elementa, ki bi bil (po imenu datoteke) enak $T[j]$ (sicer bi že tistega zamenjali s $T[i]$), lahko pa je tam kak element, ki je enak $T[i]$. Prav to se zgodi v gornjem primeru, ko je med prvim in tretjim zapisom še en zapis z enakim imenom kot prvi (`b.txt`). Zamenjava $T[i]$ in $T[j]$ v takem primeru povzroči, da pride $T[i]$ za elemente $T[i+1], \ldots, T[j-1]$ in če je imel kateri isto ime datoteke kot $T[i]$, se njun medsebojni vrstni red ob zamenjavi spremeni in pogoj za stabilnost urejanja je prekršen.

Za stabilnost lahko poskrbimo tako, da elementa $T[j]$ ne zamenjamo s $T[i]$, pač pa ga vrinemo pred $T[i]$, tako da se medsebojni vrstni red elementov $T[i], T[i+1], \ldots, T[j-1]$ nič ne spremeni.

```
1     za vsak i od 0 do n − 2:
2        za vsak j od i + 1 do n − 1:
3″          če je ime datoteke T[j] po abecedi pred imenom datoteke T[i]:
4″             za k := j, j − 1, j − 2, . . . , i + 1:
5″                zamenjaj T[k] in T[k − 1];
```

Tako smo iz urejanja z izbiranjem (*selection sort*) dobili urejanje z vstavljanjem (*insertion sort*), ki je stabilno.[6]

Še ena možna rešitev naloge pa je ta, da točki (*i*) in (*ii*) združimo v eno in torej hkrati urejamo po imenu in po datumu, takole:

```
1     za vsak i od 0 do n − 2:
2        za vsak j od i + 1 do n − 1:
3           če je ime datoteke T[j] po abecedi pred imenom datoteke T[i]
            ali pa imata datoteki enako ime in je T[j] novejša od T[i],
4              potem zamenjaj T[i] in T[j];
```

Nestabilnost urejanja nas tu nič ne moti, ker urejamo samo enkrat.

3. Usklajevanje ur

V sporočilu bomo prenašali čase t_1, t_2, t_3 in t_4 in še zaporedno številko, ki nam pove, katero izmed treh sporočil v našem protokolu je to. ZapSt = 1 pomeni sporočilo, v katerem strežnik odjemalcu sporoča t_1; ZapSt = 2 je odjemalčev odgovor, v katerem strežniku pošlje t_2 in t_3; ZapSt = 3 pa je sporočilo, s katerim strežnik odjemalcu sporoči t_4.

```
type SporociloT = record ZapSt, t1, t2, t3, t4: integer end;
```

[6] Mimogrede, tale zadnji postopek urejanja se mogoče zdi strahotno neučinkovit — ker ima tri gnezdene zanke, smo v skušnjavi, da bi si mislili, da ima časovno zahtevnost $O(n^3)$. Toda v resnici lahko razmišljamo takole: pri vsaki zamenjavi, ki jo izvede gornji algoritem, je pred zamenjavo veljalo $T[k-1] > T[k]$, po zamenjavi pa ne več. Število parov (u, v) za katere velja $u < v$ in $T[u] > T[v]$, se z vsako zamenjavo zmanjša za 1. Ker je bilo takih parov na začetku največ $n(n-1)/2$, vemo, da se vrstica $5''$ izvede največ $O(n^2)$-krat.

Ali, v C-ju:

```
typedef struct { int zapSt, t1, t2, t3, t4; } SporociloT;
```

Sinhronizacijo sprožimo tako, da na strežniku pogledamo na uro in dobljeni čas pošljemo odjemalcu kot t_1 v sporočilu z zaporedno številko 1:

```
procedure SinhronizirajUro;
var Sporocilo: SporociloT;
begin
  Sporocilo.ZapSt := 1;
  Sporocilo.t1 := VrniUro;
  PosljiSporocilo(Sporocilo);
end; {SinhronizirajUro}
```

V C-ju:

```
void SinhronizirajUro()
{
  SporociloT sporocilo;
  sporocilo.zapSt = 1;
  sporocilo.t1 = VrniUro();
  sporocilo.t2 = −1;
  sporocilo.t3 = −1;
  sporocilo.t4 = −1;
  PosljiSporocilo(sporocilo);
}
```

Podprogram SprejemNaStrezniku mora znati odreagirati na sporočilo številka 2. Vanj dopišemo še čas t_4 in ga pošljemo nazaj odjemalcu kot sporočilo 3.

```
procedure SprejemNaStrezniku(Sporocilo: SporociloT);
begin
  if Sporocilo.ZapSt <> 2 then exit; { napaka }
  Sporocilo.ZapSt := 3;
  Sporocilo.t4 := VrniUro;
  PosljiSporocilo(Sporocilo);
end; {SprejemNaStrezniku}
```

V C-ju:

```
void SprejemNaStrezniku(SporociloT sporocilo)
{
  if (sporocilo.zapSt != 2) return; /* napaka */
  sporocilo.zapSt = 3;
  sporocilo.t4 = VrniUro();
  PosljiSporocilo(sporocilo);
}
```

Podprogram SprejemNaOdjemalcu pa mora znati odreagirati na sporočila z zaporedno številko 1 in 3. V prvem primeru dopišemo v sporočilo časa t_2 in t_3 (ker vmes ne delamo ničesar drugega, bosta oba časa bolj ali manj enaka) in pošljemo strežniku sporočilo 2. Ko pa od strežnika dobimo sporočilo 3, so v njem vsi podatki, ki jih potrebujemo, zato lahko izračunamo r in popravimo odjemalčevo uro.

```
procedure SprejemNaOdjemalcu(Sporocilo: SporociloT);
var r: integer;
begin
  if Sporocilo.ZapSt = 1 then begin
    Sporocilo.ZapSt := 2;
    Sporocilo.t2 := VrniUro;
    Sporocilo.t3 := VrniUro;
    PosljiSporocilo(Sporocilo);
  end else if Sporocilo.ZapSt = 3 then with Sporocilo do begin
    r := (t2 − t4 − t1 + t3) div 2;
    NastaviUro(VrniUro − r);
  end; {if, with}
  { Če je Sporocilo.ZapSt kaj drugega kot 1 ali 3, je nekje prišlo do napake. }
end; {SprejemNaOdjemalcu}
```

V C-ju:

```
void SprejemNaOdjemalcu(SporociloT sporocilo)
{
  int r;
  if (sporocilo.zapSt == 1)
  {
    sporocilo.zapSt = 2;
    sporocilo.t2 = VrniUro();
    sporocilo.t3 = VrniUro();
    PosljiSporocilo(sporocilo);
  }
  else if (sporocilo.zapSt == 3)
  {
    r = (sporocilo.t2 − sporocilo.t4 − sporocilo.t1 + sporocilo.t3) / 2;
    NastaviUro(VrniUro() − r);
  }
  /* Če je sporocilo.zapSt kaj drugega kot 1 ali 3,
     je nekje prišlo do napake. */
}
```

Za konec si oglejmo še čudovito dekadentno rešitev, ki jo je v svojem odgovoru na tekmovanju predlagal Rok Kralj, četrtouvrščeni v drugi skupini. Mnogi programski jeziki nam omogočajo, da zahtevamo izvajanje koščka izvorne kode, ki jo podamo kot niz (npr. v neki spremenljivki). Nalogo bi lahko torej rešili tako, da bi bilo sporočilo vedno kar nek niz z izvorno kodo, podprograma SprejemNaStrezniku in SprejemNaOdjemalcu pa bi zgolj izvedla ta košček izvorne kode iz pravkar dobljenega sporočila. Večina dela se zdaj prenese v podprogram SinhronizirajUro, ki mora sestaviti takšen košček izvorne kode, ki bo, ko ga bo odjemalec prejel kot prvo sporočilo, poskrbel še za prenos drugega in tretjega sporočila in nato za sinhronizacijo ure. Primerna programska jezika za takšen pristop k reševanju naloge sta na primer PHP in python. Oglejmo si primer takšne rešitve v pythonu, kjer za izvajanje izvorne kode, podane v nizu, skrbi vgrajeni podprogram exec:

```
def SprejemNaStrezniku(sporocilo): exec(sporocilo)
def SprejemNaOdjemalcu(sporocilo): exec(sporocilo)

def SinhronizirajUro():
  PosljiSporocilo("PosljiSporocilo(\"PosljiSporocilo(\\\""
    "t1 = %d; t2 = %%d; t3 = %%d; t4 = %%%%d; "
```

```
    "r = (t2 - t4 - t1 + t3) / 2; NastaviUro(VrniUro() - r)\\\" "
    "%%%% VrniUro())\" %% (VrniUro(), VrniUro()))" % VrniUro())
```

Prvo sporočilo, ki ga pošlje strežnik odjemalcu, je torej izvorna koda takšne oblike:

```
PosljiSporocilo("PosljiSporocilo(\"t1 = ⋆; t2 = %d; t3 = %d; t4 = %%d;
                 r = (t2 - t4 - t1 + t3) / 2; NastaviUro(VrniUro() - r)\"
                 %% VrniUro())" % (VrniUro(), VrniUro()))
```

Tu smo jo zaradi preglednosti razbili v tri vrstice; v resnici se pošlje kot ena sama dolga vrstica. Namesto zvezdice ⋆ se v sporočilu v resnici pošlje število, ki podaja trenutni čas t_1, kot ga je strežniku ob sestavljanju povedala funkcija VrniUro. Ko odjemalec sporočilo prejme, izvede ta stavek in zato kot svoje sporočilo strežniku pošlje takšen niz:

```
PosljiSporocilo("t1 = ⋆; t2 = ⋆; t3 = ⋆; t4 = %d;
                 r = (t2 - t4 - t1 + t3) / 2; NastaviUro(VrniUro() - r)"
                 % VrniUro())
```

Druga in tretja zvezdica tukaj označujeta mesto, kjer se v nizu znajde številska predstavitev časov t_2 in t_3, ki ju je odjemalec dobil od funkcije VrniUro, ko je to sporočilo sestavljal. Ko strežnik prejme to sporočilo, ga izvede in ob tem pošlje odjemalcu niz takšne oblike:

```
t1 = ⋆; t2 = ⋆; t3 = ⋆; t4 = ⋆; r = (t2 − t4 − t1 + t3) / 2; NastaviUro(VrniUro() − r)
```

Tu so zdaj prisotni že vsi časi, ki jih odjemalec potrebuje. Ko se ti stavki izvedejo na odjemalcu, izračunajo popravek r in spremenijo računalnikovo uro.

Velika slabost te rešitve je seveda ta, da predstavlja potencialno varnostno luknjo v naših dveh računalnikih — vsak od njiju slepo zaupa, da koščki izvorne kode, ki jih prejema od svojega partnerja, ne bodo povzročali škode. Zlonamerneži bi lahko poslali svoje sporočilo na ista vrata, na katerih računalnika pričakujeta sporočila našega sinhronizacijskega protokola, in na ta način poskrbeli, da bi se na njiju izvedla poljubna škodljiva koda po njihovi izbiri.

Slabost te rešitve je po naših izkušnjah tudi ta, da je s sestavljanjem začetnega sporočila več dela kot pa s pisanjem tradicionalne staromodne rešitve, pri kateri se pošiljajo po mreži le časi in je logika protokola dejansko zapisana v podprogramih SprejemNaStrezniku in SprejemNaOdjemalcu.

4. Društvo ljubiteljev ničel

Operacija xor ima to lepo lastnost, da je posamezni bit rezultata odvisen le od vrednosti istoležnega bita v obeh operandih, ne pa od vrednosti drugih bitov v operandih. Zato lahko najboljšo vrednost c sestavljamo bit za bitom. Če je na primer v nekem c-ju bit 13 prižgan, se bo po xoranju vseh celic našega bloka s c vrednost bita 13 v vseh celica obrnila: tiste celice, ki so imele prej bit 13 prižgan, imajo zdaj ugasnjenega in obratno. Če je bilo prej na bitu 13 v našem bloku več enic kot ničel, je zdaj več ničel kot enic (saj so se vse bivše enice spremenile v ničle in obratno). Po drugi strani, če bi bil bit 13 v c-ju ugasnjen, bi po xoranju vseh celic bloka z vrednostjo c ostal bit 13 v vseh celicah nespremenjen. Iz tega vidimo, da je pametno bit 13 v c-ju prižgati, če je na bitu 13 v bloku več enic kot ničel, saj bomo

tako zmanjšali število ničel po xoranju; če pa je na bitu 13 v našem bloku več ničel kot enic, je pametno pustiti bit 13 v c-ju ugasnjen, saj bi si število enic po xoranju zgolj povečali, če bi ta bit prižgali.

Enak razmislek lahko seveda ponovimo na vsakem bitu, ne le na bitu 13. Spodnji podprogram se zapelje po celem bloku in v tabeli StNicel sproti povečuje števce, ki povedo, koliko celic bloka ima na posameznem bitu ničlo. Na koncu prižgemo v c tiste bite, pri katerih smo opazili več enic kot ničel.

```
function NajvecNicel(Blok: BlokT): word;
const b = 16; m = (1 shl b) − 1;
var c: word; i, j: integer;
    StNicel: array [0..b − 1] of integer;
begin
  { Za vsak bit preštejmo, v koliko celicah bloka je prižgan. }
  for j := 0 to b − 1 do StNicel[j] := 0;
  for i := 0 to n − 1 do begin
    c := Blok[i];
    for j := 0 to b − 1 do
      if not Odd(c shr j) then StNicel[j] := StNicel[j] + 1;
  end; {for i}
  { V c prižgimo tiste bite, kjer je enic več kot ničel. }
  c := 0;
  for j := 0 to b − 1 do
    if StNicel[j] < n − StNicel[j] then
      c := c or (1 shl j);
  NajvecNicel := c;
end; {NajvecNicel}
```

In v C-ju:

```
#define b 16
#define m ((1 << b) − 1)

unsigned short NajvecNicel(const BlokT blok)
{
  unsigned short c; int i, j, stNicel[b];
  for (j = 0; j < b; j++) stNicel[j] = 0;
  /* Za vsak bit poglejmo, koliko ničel je na tem mestu v našem bloku. */
  for (i = 0; i < n; i++)
    for (j = 0, c = blok[i]; j < b; j++)
      if (! (c & (1 << j))) stNicel[j]++;
  /* V c prižgimo tiste bite, kjer je enic več kot ničel. */
  for (j = 0, c = 0; j < b; j++)
    if (stNicel[j] < n − stNicel[j]) c |= 1 << j;
  return c;
}
```

Časovna zahtevnost tega postopka je $O(nb)$, če imamo blok n besed in ima vsaka beseda b bitov (pri naši nalogi je $b = 16$). Če je n velik v primerjavi z 2^b, pa lahko to rešitev še izboljšamo: za vsako celico je le 2^b možnih vrednosti, torej si lahko v neki tabeli (f v spodnjem podprogramu) za vsako od njih zapomnimo, kolikokrat se pojavlja v našem bloku. Potem nam ne bo treba šteti ničel in enic v vsaki celici bloka posebej, ampak le po enkrat za vsako možno vrednost celice. Časovna zahtevnost je le še $O(n + b2^b)$.

```
function NajvecNicel2(Blok: BlokT): word;
const b = 16; m = (1 shl b) − 1;
var c: word; i, j: integer;
    f: array [0..m] of integer;
    StNicel: array [0..b − 1] of integer;
begin
  { Za vsako možno vrednost preštejmo, kolikokrat se pojavlja. }
  for c := 0 to m do f[c] := 0;
  for i := 0 to n − 1 do f[Blok[i]] := f[Blok[i]] + 1;
  { Preštejmo, koliko je v bloku ničel na vsakem bitu. }
  for j := 0 to b − 1 do StNicel[j] := 0;
  for c := 0 to m do if f[c] > 0 then
    for j := 0 to b − 1 do
      if not Odd(c shr j) then StNicel[j] := StNicel[j] + f[c];
  { V c prižgimo tiste bite, kjer je enic več kot ničel. }
  c := 0;
  for j := 0 to b − 1 do
    if StNicel[j] < n − StNicel[j] then
      c := c or (1 shl j);
  NajvecNicel2 := c;
end; {NajvecNicel2}
```

Še v C-ju:

```
unsigned short NajvecNicel2(const BlokT blok)
{
  unsigned short c; int i, j, f[m + 1], stNicel[b];
  /* Za vsako možno vrednost preštejmo, kolikokrat se pojavlja. */
  for (i = 0; i <= m; i++) f[i] = 0;
  for (i = 0; i < n; i++) f[blok[i]]++;
  /* Preštejmo, koliko je v bloku ničel na vsakem bitu. */
  for (j = 0; j < b; j++) stNicel[j] = 0;
  for (i = 0; i <= m; i++) if (f[i] > 0)
    for (j = 0; j < b; j++) if (! (i & (1 << j))) stNicel[j] += f[i];
  /* V c prižgimo tiste bite, kjer je enic več kot ničel. */
  for (j = 0, c = 0; j < b; j++)
    if (stNicel[j] < n − stNicel[j]) c |= 1 << j;
  return c;
}
```

5. Cik cak

Recimo, da se črta začne na višini $y = 0$. Ob vsakem znaku „/“ se dvigne za 1, ob vsakem „\“ pa se spusti za 1. Za začetek se sprehodimo po nizu s, opazujmo, kako se spreminja višina, in si zapomnimo najvišjo in najnižjo doseženo višino (MinY in MaxY v spodnjem podprogramu). Ker ima najvišja dosežena točka na črti y-koordinato MaxY, bo prva vrstica črte predstavljala pas $\mathsf{MaxY} - 1 \le y \le \mathsf{MaxY}$, naslednja vrstica pas $\mathsf{MaxY} - 2 \le y \le \mathsf{MaxY} - 1$, itd., zadnja vrstica pa pas $\mathsf{MinY} \le y \le \mathsf{MinY} + 1$. Za vsako od teh vrstic torej zdaj naredimo še en prehod skozi niz s, opazujmo višino črte in se pri vsakem znaku odločimo, ali je črta tam nad trenutno vrstico (in moramo izpisati o), pod njo (in moramo izpisati piko) ali pa ravno na njej (in moramo izpisati trenutni znak).

```
procedure NarisiCrto(s: string);
var MinY, MaxY, y, u, i, n: integer;
```

```
begin
  n := Length(s);
  { Poiščimo najvišjo in najnižjo doseženo višino. }
  y := 0; MinY := y; MaxY := y;
  for i := 1 to n do begin
    if s[i] = '/' then y := y + 1 else y := y − 1;
    if y > MaxY then MaxY := y;
    if y < MinY then MinY := y;
  end; {for i}
  { Narišimo črto. }
  for u := MaxY − 1 downto MinY do begin
    y := 0;
    for i := 1 to n do begin
      { Če je trenutni znak '\', pokriva črta tu višine med y − 1 in y,
        če je trenutni znak '/', pa pokriva višine med y in y + 1.
        Naš u pa je višina spodnjega roba vrstice, ki jo trenutno izpisujemo. }
      if s[i] = '\' then y := y − 1;
      if u = y then Write(s[i])
      else if u < y then Write('o')
      else Write('.');
      if s[i] = '/' then y := y + 1;
    end; {for i}
    WriteLn;
  end; {for u}
end; {NarisiCrto}
```

Še rešitev v C-ju:

```
#include <stdio.h>
void NarisiCrto(const char *s)
{
  int minY = 0, maxY = 0, y = 0, u, i;
  for (i = 0; s[i]; i++)
  {
    if (s[i] == '/') y++; else y−−;
    if (y > maxY) maxY = y;
    if (y < minY) minY = y;
  }
  for (u = maxY − 1; u >= minY; u−−)
  {
    for (y = 0, i = 0; s[i]; i++)
    {
      if (s[i] == '\\') y−−;
      putchar(u == y ? s[i] : u < y ? 'o' : '.');
      if (s[i] == '/') y++;
    }
    putchar('\n');
  }
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. PINi

Pri tej nalogi so PINi zaporedja n števk, torej jih lahko predstavimo s celimi števili od 0 do $10^n - 1$. Naj bo f funkcija, ki nam za dani PIN izračuna novega po postopku iz naloge; na primer, pri $n = 4$ je $f(1979) = 9796$. V spodnji rešitvi to funkcijo računa podprogram Nasl. Do posameznih števk PINa a lahko pridemo tako, da upoštevamo, da je $a \bmod 10$ zadnja števka števila a, število $a \operatorname{div} 10$ pa je število a brez zadnje števke. V zanki lahko izluščimo posamezne števke a-ja in jih seštevamo. Na koncu moramo vzeti a brez prve števke (to je $a \operatorname{div} 10^{n-1}$) in mu na desni pritakniti novo števko (torej ga pomnožimo z 10 in mu novo števko prištejemo).

Če zdaj začnemo z nekim začetnim PINom a in opazujemo, kam nas pripelje funkcija f, dobimo po vrsti $a, f(a), f(f(a)), f(f(f(a))), \ldots$ Označimo $a_0 = a$ in $a_{k+1} = f(a_k)$. Ker obstaja le 10^n različnih PINov, se začnejo PINi v tem zaporedju prej ali slej ponavljati. Naj bo j najmanjši tak indeks, pri katerem je $a_j = a_i$ za nek $i < j$. Ali je mogoče, da je $i > 0$? To bi pomenilo, da je isti PIN ($a_i = a_j$) nastal tako iz a_{i-1} kot iz a_{j-1}; ta dva PINa pa sta različna. Toda ker sta a_i in a_j enaka, se ujemata v prvih $n - 1$ števkah, torej se morata a_{i-1} in a_{j-1} ujemati v zadnjih $n - 1$ števkah; če pa bi se pri tem razlikovala v prvi števki, bi imela različno vsoto števk, vendar bi se vsoti razlikovali za največ 9; torej bi bil ostanek po deljenju teh vsot z 10 različen, torej bi se morala a_i in a_j razlikovati v zadnji števki. Ker pa se ne, smo prišli v protislovje. Možnost $i > 0$ torej odpade, kar pomeni, da bomo prvo ponavljanje v našem zaporedju opazili takrat, ko bo nek a_j enak a_0, torej se bo ponovil kar naš začetni PIN a. Takrat torej vemo, da se tako a kot vsi drugi PINi v tem zaporedju začnejo ponavljati po j dnevih.

Ker je možnih PINov obvladljivo mnogo (10^n, pri čemer je $n \leq 6$, torej največ milijon PINov), si lahko v neki tabeli (v spodnjem programu se imenuje ZeVidel) za vsak PIN zapomnimo, ali smo ga v našem zaporedju že videli ali ne. Sproti štejemo korake in ko se nam ponovi začetni PIN, nam število korakov pove ravno odgovor za podnalogo (a). Enak postopek lahko potem ponovimo še za druge začetne PINe, dokler ne pregledamo vseh možnih PINov. Ko pri nekem začetnem PINu u opazimo, da se nam je le-ta ponovil po recimo d korakih, lahko povečamo števec ciklov (to bo odgovor na podnalogo (c)), obenem pa tudi vemo, da za vseh d PINov na trenutnem ciklu velja, da se začnejo ponavljati po d korakih. Tako lahko tudi sproti štejemo, pri koliko PINih pride do ponavljanja prej kot pri začetnem PINu z iz vhodne datoteke (to pa bo na koncu odgovor za podnalogo (b)).

```
program PINi;
const MaxN = 6; MaxM = 1000000;
var n, b, m, mm: integer;

  function Nasl(Pin: integer): integer;
  var i, Vsota, Novi: integer;
  begin
    Novi := (Pin mod mm) * b; Vsota := 0;
    for i := 1 to n do begin
      Vsota := Vsota + Pin mod b;
      Pin := Pin div b;
    end; {for i}
```

```
      Nasl := Novi + Vsota mod b;
   end; {Nasl}

var z, u, v, StCiklov, d, DolzinaZ, StKrajsih: integer; F: text;
     ZeVidel: packed array [0..MaxM − 1] of boolean;
begin {PINi}
   { Preberimo vhodne podatke. }
   Assign(F, 'pini.in');
   Reset(F); ReadLn(F, n); b := 10;
   m := 1; for u := 1 to n do m := m * b;    { m = 10^n }
   mm := m div b;                            { mm = 10^(n − 1) }
   ReadLn(F, z); Close(F);

   for u := 0 to m − 1 do ZeVidel[u] := false;
   { Poglejmo, po koliko korakih se začne ponavljati PIN z. }
   v := z; d := 0;
   while not ZeVidel[v] do begin ZeVidel[v] := true; d := d + 1; v := Nasl(v) end;
   StCiklov := 1; DolzinaZ := d; StKrajsih := 0;
   { Preglejmo zdaj še ostale PINe. }
   for u := 0 to m − 1 do if not ZeVidel[u] then begin
      { Računajmo naslednike u-ja, dokler ne pridemo spet do u. }
      d := 0; v := u;
      while not ZeVidel[v] do begin ZeVidel[v] := true; d := d + 1; v := Nasl(v) end;
      StCiklov := StCiklov + 1;
      { Na tem ciklu je d PINov, ki se začnejo ponavljati po d korakih.
        Je to manj kot pri PINu z, ki se začne ponavljati po DolzinaZ korakih? }
      if d < DolzinaZ then StKrajsih := StKrajsih + d;
   end; {for u}

   Assign(F, 'pini.out'); Rewrite(F);
   WriteLn(F, DolzinaZ); WriteLn(F, StKrajsih); WriteLn(F, StCiklov);
   Close(F);
end. {PINi}
```

Še rešitev v C-ju:

```
#include <stdio.h>
#include <stdbool.h>
#define MaxN 6
#define MaxM 1000000

int n, b, m, mm;
bool zeVidel[MaxM];

int Nasl(int pin)
{
   int i, vsota = 0, novi = (pin % mm) * b;
   for (i = 0; i < n; i++) { vsota += pin % b; pin /= b; }
   return novi + vsota % b;
}

int main()
{
   int z, u, v, d, dolzinaZ, stKrajsih, stCiklov;

   /* Preberimo vhodne podatke. */
   FILE *f = fopen("pini.in", "rt");
   fscanf(f, "%d", &n); b = 10;
```

```
  m = 1; for (u = 0; u < n; u++) m *= b; /* m = 10^n */
  mm = m / b;                            /* mm = 10^(n − 1) */
  fscanf(f, "%d", &z); fclose(f);

  for (u = 0; u < m; u++) zeVidel[u] = false;
  /* Poglejmo, po koliko korakih se začne ponavljati PIN z. */
  for (v = z, d = 0; ! zeVidel[v]; v = Nasl(v), d++) zeVidel[v] = true;
  dolzinaZ = d; stCiklov = 1; stKrajsih = 0;
  /* Preglejmo zdaj še ostale PINe. */
  for (u = 0; u < m; u++) if (! zeVidel[u])
  {
    /* Računajmo naslednike u-ja, dokler ne pridemo spet do u. */
    for (v = u, d = 0; ! zeVidel[v]; v = Nasl(v), d++) zeVidel[v] = true;
    /* Na tem ciklu je d PINov, ki se začnejo ponavljati po d korakih.
       Je to manj kot pri PINu z, ki se začne ponavljati po DolzinaZ korakih? */
    stCiklov++;
    if (d < dolzinaZ) stKrajsih += d;
  }

  /* Izpišimo rezultate. */
  f = fopen("pini.out", "wt");
  fprintf(f, "%d\n%d\n%d\n", dolzinaZ, stKrajsih, stCiklov);
  fclose(f); return 0;
}
```

Boljša rešitev s pomočjo teorije števil. Besedilo naše naloge zagotavlja, da bo pri vseh testnih primerih veljalo $n \leq 6$ in za take majhne n je zgoraj opisana rešitev povsem primerna. V nadaljevanju pa si oglejmo še, kako lahko rešitev s pomočjo nekaj ugotovitev iz teorije števil močno izboljšamo, tako da bo zmogla obdelati tudi malo večje n.

V zaporedju PINov $z, f(z), f(f(z)), \ldots$ je vsak naslednji PIN dobljen tako, da prejšnjemu pobrišemo eno števko na začetku in dodamo eno števko na koncu. Namesto zaporedja PINov lahko torej opazujemo kar zaporedje števk: na primer, pri $z = 1979$ imamo PINe $\underline{1979}, 979\underline{6}, 796\underline{1}, 961\underline{3}, \ldots$, kar lahko krajše predstavimo tako, da zapišemo le podčrtane števke: $1979613992\ldots$. Označimo števke v tem zaporedju z $a_1, a_2, \ldots$; zaradi pravila, s katerim računamo nove PINe, vidimo, da je vsaka števka (razen prvih n števk, ki smo jih dobili iz začetnega PINa) pridobljena iz prejšnjih n števk po formuli $a_k = (a_{k-1} + a_{k-2} + \ldots + a_{k-n}) \bmod 10$. Zaradi te formule je celotno zaporedje a_k-jev enolično določeno že s tem, ko smo si izbrali $a_1, \ldots, a_n$, torej števke začetnega PINa. To, da se začnejo PINi prej ali slej ponavljati, pa se na zaporedju a odrazi v tem, da se prej ali slej enkrat pojavi skupina n zaporednih števk, ki so enake prvim n števkam: naj bo torej t najmanjši tak indeks ($t > 0$), za katerega je $a_{t+1} = a_1, \ldots, a_{t+n} = a_n$. To je znak, da bi po t korakih iz začetnega PINa z spet prišli nazaj do istega PINa in odtlej se ponavljata tako zaporedje PINov kot zaporedje števk.

Definirajmo zdaj še dve zaporedji b in c s členi $b_k := a_k \bmod 2$ in $c_k := a_k \bmod 5$. Potem velja $b_k = ((a_{k-1} + \ldots + a_{k-n}) \bmod 10) \bmod 2 = (a_{k-1} + \ldots a_{k-n}) \bmod 2 = ((a_{k-1} \bmod 2) + \ldots + (a_{k-n} \bmod 2)) \bmod 2 = (b_{k-1} + \ldots b_{k-n}) \bmod 2$. Podobno bi se lahko prepričali tudi, da je $c_k = (c_{k-1} + \ldots + c_{k-n}) \bmod 5$. Zaporedji števk b_k in c_k lahko torej računamo po enakem postopku kot a_k, le da gledamo ostanke po deljenju z 2 oz. 5 namesto z 10. Za njiju velja podobno kot za zaporedje a: možnih

skupin n zaporednih števk je le končno mnogo (2^n ali 5^n), zato se v zaporedjih b in c prej ali slej pojavi ista n-terica števk kot na začetku in odtlej se zaporedji periodično ponavljata. Recimo, da se v b začetna n-terica prvič ponovi po u korakih, v c pa po v korakih, torej da je $b_{u+1} = b_1, \ldots, b_{u+n} = b_n$ in $c_{v+1} = c_1, \ldots, c_{v+n} = c_n$.

Ni se težko prepričati, da je vsaka od števk $0, \ldots, 9$ enolično določena s svojima ostankoma po deljenju z 2 in 5. Obstaja celo eksplicitna formula, s katero iz obeh ostankov izračunamo nazaj prvotno števko: $d = [5 \cdot (d \bmod 2) + 6 \cdot (d \bmod 5)] \bmod 10$. To je poseben primer ugotovitve, ki je v matematiki znana kot „kitajski izrek o ostankih“.[7] Če ta razmislek uporabimo pri naših zaporedjih, vidimo, da pri vsakem i velja $a_i = (5b_i + 6c_i) \bmod 10$.

Naj bo τ najmanjši skupni večkratnik števil u in v. Ker se b ponavlja s periodo u, je $b_{\tau+i} = b_i$; podobno se c ponavlja s periodo v, zato je $c_{\tau+i} = c_i$. Potemtakem pa je $a_{\tau+i} = (5b_{\tau+i} + 6c_{\tau+i}) \bmod 10 = (5b_i + 6c_i) \bmod 10 = a_i$. Torej so števke $a_{\tau+1}, \ldots, a_{\tau+k}$ enake prvim k števkam $a_1, \ldots, a_k$. Ker smo prej rekli, da se a ponavlja s periodo t, sledi, da je τ večkratnik t-ja.

Po drugi strani, ker se a ponavlja s periodo t, je $a_{t+1} = a_1, \ldots, a_{t+k} = a_k$. Torej je tudi $b_{t+1} = b_1, \ldots, b_{t+n} = b_n$; če pišemo $t = r \cdot u + o$ za nek $0 \leq o < u$, je $b_{t+j} = b_{r \cdot u + o + j} = b_{o+j}$, torej iz $b_{t+1} = b_1, \ldots, b_{t+n} = b_n$ sledi $b_{o+1} = b_1, \ldots, b_{o+n} = b_n$. Torej je nemogoče, da bi bil $o > 0$, ker bi to pomenilo, da se začetna n-terica števk zaporedja b ponovi že po o korakih, ne pa šele po u korakih (mi pa smo za u vzeli najmanjši indeks, pri katerem se začetna n-terica ponovi). Torej je $o = 0$, kar pomeni, da je t večkratnik števila u. Podoben razmislek bi lahko opravili pri zaporedju c in videli, da je t tudi večkratnik števila v. Torej je t skupni večkratnik števil u in v, kar pomeni, da je t tudi večkratnik τ-ja.

Tako torej vidimo, da t in τ delita drug drugega, kar pomeni, da sta enaka. Perioda zaporedja a je torej najmanjši skupni večkratnik period zaporedij b in c.

S temi ugotovitvami lahko pridemo do cenejšega postopka za reševanje naše naloge. Oglejmo si za začetek podvprašanje (a). Naša prvotna rešitev si je v tabeli velikosti 10^n označevala, kateri PINi (torej: katere skupine n zaporednih števk v zaporedju a) so se že pojavili, in računala nove in nove člene zaporedja, dokler ni prišla nazaj do prvotnega PINa. Iz števk $a_1, \ldots, a_n$ (ki smo jih dobili iz začetnega PINa) lahko izračunamo $b_1, \ldots, b_n$ ter $c_1, \ldots, c_n$ in nato po enakem postopku kot prej za a poiščemo periodo zaporedij b in c, najmanjši skupni večkratnik teh dveh period pa je perioda zaporedja a — to pa je ravno tisto, po čemer sprašuje podnaloga (a). Pri delu z zaporedjem b lahko uporabimo tabelo 2^n elementov, pri zaporedju c pa tabelo 5^n elementov, torej prihranimo ogromno pomnilnika v primerjavi s prvotno rešitvijo.

Do odgovorov za podnalogi (b) in (c) ni težko priti, če uspemo prešteti, koliko zaporedij s kakšno periodo obstaja. To lahko spet naredimo posebej za dvojiška zaporedja in posebej za petiška zaporedja po enakem postopku, kot ga je naša pr-

[7] Gl. npr. Wikipedijo *s. v.* Chinese remainder theorem. V splošnem ta izrek pravi, da če imamo neka paroma tuja si števila $m_1, \ldots, m_r$, je potem vsako celo število od 0 do $M - 1$ (za $M = \prod_{i=1}^{r} m_i$) enolično določeno s svojimi ostanki po deljenju z $m_1, \ldots, m_r$. Z drugimi besedami, vsakemu naboru ostankov $o_1, \ldots, o_r$ (pri čemer je $0 \leq o_i < m_i$ za vse $i = 1, \ldots, r$) pripada natanko tako eno število $x \in \{0, \ldots, M - 1\}$, za katero je $o_i = x \bmod m_i$ za vse i od 1 do r. Pri naši nalogi delamo v desetiškem sestavu, torej nas zanima $M = 10$ in smo zato vzeli $m_1 = 2$ in $m_2 = 5$.

votna rešitev uporabila za desetiška zaporedja. Pri tem spet porabimo $O(2^n + 5^n)$ pomnilnika in $O(n \cdot (2^n + 5^n))$ časa. Pri $n = 4$ na primer za dvojiška zaporedja ugotovimo, da imamo en cikel dolžine 1 in tri cikle dolžine 5 (torej 15 zaporedij s periodo 5); za petiška zaporedja pa ugotovimo, da imamo en cikel dolžine 1 in dva cikla dolžine 312 (torej 624 zaporedij s periodo 312). Potem lahko razmišljamo takole. Vsako dvojiško zaporedje b s periodo u je enolično določeno s svojimi prvimi n števkami $b_1, \ldots, b_n$; podobno je tudi vsako petiško zaporedje c s periodo v enolično določeno s svojimi prvimi n števkami $c_1, \ldots, c_n$. Iz njiju lahko zdaj po zgoraj omenjeni formuli $a_i = (5b_i + 6c_i) \bmod 10$ izračunamo desetiško zaporedje, čigar perioda je (kot smo videli zgoraj) ravno najmanjši skupni večkratnik števil u in v. Če bi namesto b-ja ali c-ja vzeli kakšni drugi dve zaporedji, recimo b' in c', in iz njiju spet izračunali desetiško zaporedje a' po formuli $a'_i = (5b'_i + 6c'_i) \bmod 10$), bi bilo to zaporedje zagotovo drugačno od a: formula, s katero računamo a_i iz b_i in c_i (ali pa a'_i iz b'_i in c'_i) nam namreč zagotavlja, da je $b_i = a_i \bmod 2$ in $c_i = a_i \bmod 5$ (in podobno $b'_i = a'_i \bmod 2$ in $c'_i = a'_i \bmod 5$); če bi torej bilo $a_i = a'_i$, bi sledilo $b_i = (a_i \bmod 2) = (a'_i \bmod 2) = b'_i$ in podobno $c_i = c'_i$, torej bi prišli v protislovje s predpostavko, da smo za b' in c' vzeli drugi dve zaporedji kot za b in c.

Iz tega torej vidimo, da če imamo npr. r_u različnih dvojiških zaporedij s periodo u in s_v različnih petiških zaporedij s periodo v, potem imamo tudi $r_u \cdot s_v$ različnih desetiških zaporedij s periodo $\text{lcm}\{u, v\}$ (torej najmanjši skupni večkratnik u in v).[8] Pri $n = 4$ imamo npr. eno dvojiško zaporedje s periodo 1 in petnajst dvojiških zaporedij s periodo 5; pri petiških zaporedjih pa eno s periodo 1 ter 624 zaporedij s periodo 312. Iz tega lahko sklepamo, da imamo pri desetiških zaporedjih (za $n = 4$) eno zaporedje s periodo 1; petnajst s periodo 5; 624 s periodo 312; in 9360 ($= 15 \cdot 624$) zaporedij s periodo $5 \cdot 312 = 1560$. (Za preizkus lahko preverimo, če smo res prešteli vseh 10^n zaporedij: $1 + 15 + 624 + 9360$ je res enako 10000.)

Z opisano izboljšavo lahko nalogo dovolj učinkovito rešimo tudi za malo večje n. Pri $n = 15$ potrebujemo na primer za petiška zaporedja tabelo 5^{15} bitov, kar je približno 3,5 GB; tabela velikosti 10^{15} bitov, kakršno bi tu zahtevala naša prvotna rešitev, pa bi bila že neobvladljivo velika. Podobno dobrodošla je tu tudi izboljšava v porabi časa.

Zapišimo to rešitev še v obliki programa — tokrat v pythonu, da ne bo predolg:[9]

```
def Naslednji(pin, n, b):
    novi = pin % (b ** (n - 1))  # odrežemo prvo števko
    vsota = 0
    while pin > 0: vsota += pin % b; pin //= b
    return novi * b + (vsota % b)

def PreglejCikel(pin, n, b, zeVidel):
    dolzina = 0
    while not zeVidel[pin]:
        zeVidel[pin] = True
        pin = Naslednji(pin, n, b)
```

[8] To sicer še niso nujno vsa desetiška zaporedja s periodo $\text{lcm}\{u, v\}$, ker lahko včasih isto vrednost $\text{lcm}\{u, v\}$ dosežemo z različnimi pari (u, v).

[9] Treba pa je priznati, da v pythonu najbrž ne bi mogli iti do $n = 15$, ker bi v tabeli zeVidel vsak element zasedel precej več kot 1 bit pomnilnika. Pri našem poskusu s to pythonovsko različico rešitve je zasedel vsak element po 32 bitov. Če nas zanimajo večji n, bi bilo koristno za tabelo zeVidel uporabiti tip bytes, ki ga podpira oz. bo podpiral python od verzije 2.6 naprej.

```
        dolzina += 1
    return dolzina

def PrestejCikle(zacetni, n, b):
    zeVidel = [False] * (b ** n)
    # Najprej poglejmo, na kako dolgem ciklu je začetni PIN.
    dolzina = PreglejCikel(zacetni, n, b, zeVidel)
    stPinovPoDolzini = {dolzina: dolzina}
    # Preglejmo še vse ostale cikle.
    for pin in xrange(b ** n):
        if zeVidel[pin]: continue
        d = PreglejCikel(pin, n, b, zeVidel)
        stPinovPoDolzini[d] = stPinovPoDolzini.get(d, 0) + d
    return dolzina, stPinovPoDolzini

# Evklidov algoritem za največji skupni delitelj.
def gcd(u, v):
    while v > 0: (u, v) = (v, u % v)
    return u
# Najmanjši skupni večkratnik.
def lcm(u, v): return (u // gcd(u, v)) * v

# Preberimo vhodne podatke.
f = file("pini.in", "rt")
n = int(f.readline())
pin = f.readline().strip()
f.close()

# Izračunajmo začetne števke pripadajočega dvojiškega in petiškega zaporedja.
pin2 = sum((int(pin[i]) % 2) * (2 ** (n - 1 - i)) for i in xrange(n))
pin5 = sum((int(pin[i]) % 5) * (5 ** (n - 1 - i)) for i in xrange(n))

# Preštejmo, koliko ciklov kakšne dolžine je pri dvojiških in petiških zaporedjih.
dolzina2, st2 = PrestejCikle(pin2, n, 2)
dolzina5, st5 = PrestejCikle(pin5, n, 5)

# Kako dolg je cikel našega začetnega (desetiškega) PINa?
dolzina = lcm(dolzina2, dolzina5)

# Izračunajmo zdaj, koliko je ciklov pri desetiških PINih
# in koliko PINov je na krajših ciklih od našega zaeetnega.
stNaKrajsih = 0; stCiklov = 0
for (d2, koliko2) in st2.iteritems():
    for (d5, koliko5) in st5.iteritems():
        d = lcm(d2, d5)
        koliko = koliko2 * koliko5
        # Našli smo še „koliko“ desetiških PINov, ki se začnejo
        # ponavljati po d korakih. Torej na vsakih d takih pinov pride en cikel.
        stCiklov += koliko // d
        if d < dolzina: stNaKrajsih += koliko
# Izpišimo rezultate.
f = file("pini.out", "wt")
f.write("%d\n%d\n%d\n" % (dolzina, stNaKrajsih, stCiklov))
f.close()
```

Kot zanimivost si oglejmo skupno število ciklov (torej odgovor pri podnalogi (c)) za prvih nekaj n-jev: za n od 1 do 17 dobimo 10, 6, 20, 12, 40, 850, 816, 206, 280, 66 458, 267 140, 80 580, 2 980, 375 032, 10 340, 19 492 080 in 36 580.

Rešitev z manjšo porabo pomnilnika. Videli smo, da pravkar opisana rešitev porabi pravzaprav le še 5^n bitov pomnilnika, če delamo z n-mestnimi PINi. To je že velika izboljšava v primerjavi z 10^n pri prvotni rešitvi, vendar pa tudi 5^n hitro postane preveč. Na primer, videli smo, da je pri $n = 15$ tabela 5^n bitov velika že dobre 3,5 GB, pri $n = 16$ pa že skoraj 18 GB. Prvo mogoče že še gre, 18 GB pomnilnika pa dandanes še ni preveč pogosto. Oglejmo si zdaj še rešitev, ki porabi zelo malo pomnilnika, žal pa je cena za to precej večja poraba časa.

V naši gornji rešitvi potrebuje veliko tabelo zeVidel predvsem podprogram PrestejCikle, da z njeno pomočjo takoj vidi, če je nek PIN že pregledal (v okviru kakšnega od doslej pregledanih ciklov) in mu ga torej ni treba pregledovati še enkrat. To tabelo uporablja sicer tudi podprogram PreglejCikel, vendar zanj ni tako kritična — to, kdaj je prišel naokrog celega cikla, lahko preverja tudi tako, da si pač zapomni PIN, pri katerem je začel, in po vsakem koraku preverja, če je novi PIN enak tistemu prvotnemu. Podprogramu PrestejCikle pa lahko pomagamo takole: ker po novem ne bo več imel tabele, ki bi mu povedala, kateri PINi so bili že obiskani, bo moral začeti pregledovati cikel pri čisto vsakem PINu. Vendar pa, ker pregledujemo PINe po naraščajočem vrstnem redu, vemo, da vsak cikel obiščemo že takrat, ko je bila naša spremenljivka pin enaka najmanjšemu PINu tega cikla. Če torej pri sprehodu po nekem ciklu pridemo do kakšnega PINa, ki je manjši od tistega, pri katerem smo začeli, vemo, da smo ta cikel že videli in ga lahko takoj nehamo pregledovati.

V našem gornjem pythonovskem programu lahko zdaj pobrišemo podprogram PreglejCikel, nato pa podprogram PrestejCikle zamenjamo z naslednjim:

```
def PrestejCikle(zacetni, n, b):
  # Najprej poglejmo, na kako dolgem ciklu je začetni PIN.
  dolzina = 1; u = Naslednji(zacetni, n, b)
  while u != zacetni: u = Naslednji(u, n, b); dolzina += 1
  # Preglejmo zdaj sistematično vse cikle.
  stPinovPoDolzini = {}
  for pin in xrange(b ** n):
    # Preglejmo cikel, na katerem leži ta PIN.
    u = Naslednji(pin, n, b); d = 1
    while u > pin: u = Naslednji(u, n, b); d += 1
    if u < pin: continue # Ta cikel smo nekoč prej že pregledali.
    stPinovPoDolzini[d] = stPinovPoDolzini.get(d, 0) + d
  return dolzina, stPinovPoDolzini
```

Poraba časa bo zdaj seveda večja kot prej, ker se lahko zgodi, da mora notranja zanka **while** narediti kar nekaj iteracij, preden opazimo, da smo ta cikel nekoč že videli. Vendar pa v povprečju ponavadi ni treba prav veliko iteracij, preden to opazimo; pri $n = 6$ za petiška zaporedja ($b = 5$) na primer porabimo povprečno po 5,4 iteracije; pri $n = 10$ pa po 10,9 iteracije. Če to pomeni, da bomo porabili desetkrat več časa kot pri prejšnji rešitvi, je stvar mogoče vendarle sprejemljiva, če bomo zaradi tega na koncu vsaj prišli do rezultata, pri prejšnji rešitvi pa ne bi mogli, ker nimamo 5^n bitov pomnilnika. Omeniti velja tudi, da je sedanji postopek zelo primeren za paralelno izvajanje — vsak procesor (ali pa vsak računalnik) lahko na primer pregleda nek interval možnih vrednosti spremenljivke pin, pri tem so lahko čisto ločeni (ne potrebujejo skupnega pomnilnika in ne komunicirajo med seboj), na koncu pa le združimo (torej seštejemo) dobljene tabele stPinovPoDolzini z vseh procesorjev.

2. Berberi

Zemljevid začnimo preiskovati v tistem polju na zahodnem robu, pri katerem so Berberi sploh vstopili na zemljevid. Od tam sledimo njihovim selitvam po poljih, označenih z `X`. Kjer se njihova pot razveji, obiščimo najprej eno nadaljevanje poti in kasneje še drugo. Pri tem torej vidimo, da je koristno, če si pri vsakem polju zapomnimo, od kod smo v to polje prišli (da se bomo lahko kasneje vrnili po isti poti). Ko se v neko točko razvejitve vrnemo, bi radi zdaj nadaljevali po drugi poti kot prvič, kar pomeni, da si je koristno zapomniti tudi, katera polja smo že obiskali. Poleg tega moramo v neki spremenljivki hraniti podatek o tem, čigavo pot trenutno spremljamo: pot združene berberske vojske pred prvo razvejitvijo, pot Yusufove vojske (in njenih naslednic) ali pot Muhammadove vojske (in njenih naslednic). Na začetku vemo, da spremljamo pot združene vojske; pri pri prvi razvejitvi nadaljujmo npr. najprej po Muhammadovi poti, koordinate te začetne razvejitve pa si zapomnimo. Ko se bomo kasneje vrnili vanjo in nadaljevali po drugem možnem nadaljevanju, bomo torej vedeli, da se zdaj gibljemo po Yusufovi poti.

Pri vsakem razvejišču (razen prvem) in pri vsakem polju, ki nima nobenega prehojenega soseda (razen tistega, iz katerega smo pravkar prišli), povečamo števec zaselkov za tisto vojsko, katere pot trenutno spremljamo.

Ko stojimo v nekem polju in ugotavljamo, katera sosednja polja so Berberi še prehodili, moramo paziti na primere, ko leži trenutno polje na robu zemljevida in torej nekaterih sosedov mogoče sploh nima. Spodnji program si pomaga s tem, da zemljevid na vseh robovih poveča še za eno polje, torej ima tabelo velikosti $(w+2) \times (h+2)$ namesto $w \times h$; ta dodatna polja na robovih pa vsa obravnava kot prazna (kot da bi bile v vhodni datoteki tam pike).

```
program Berberi;
const MaxW = 3000; MaxH = 3000;
type CelicaT = (Gor, Levo, Dol, Desno, Prazna, Pregledana, Neobiskana);
     KdoT = (Nic, Yusuf, Muhammad);
const DX: array [CelicaT] of integer= (0, −1, 0, 1, 0, 0, 0);
      DY: array [CelicaT] of integer= ( −1, 0, 1, 0, 0, 0, 0);
var A: packed array [0..MaxH + 1, 0..MaxW + 1] of CelicaT;
    w, h, x, y, xx, yy, yKoren, xRazcep, Stopnja: integer; c: char; F: text;
    StMest: array [KdoT] of integer; d, dNasl: CelicaT; Poddrevo: KdoT; Prvic: boolean;
begin
  { Preberimo vhodne podatke. }
  Assign(F, 'berberi.in'); Reset(F); ReadLn(F, w, h);
  for x := 1 to w do begin A[0, x] := Prazna; A[h + 1, x] := Prazna end;
  for y := 1 to h do begin A[y, 0] := Prazna; A[y, w + 1] := Prazna end;
  yKoren := −1; xRazcep := −1;
  for y := 1 to h do begin
    for x := 1 to w do begin
      Read(F, c); if c = '.' then A[y, x] := Prazna else A[y, x] := Neobiskana;
      if x = 1 then if c = 'X' then yKoren := y;
    end; {for x}
    ReadLn(F);
  end; {for y}
  Close(F);
  { Preglejmo drevo. }
  StMest[Yusuf] := 0; StMest[Muhammad] := 0; StMest[Nic] := 0; Poddrevo := Nic;
```

```
  x := 1; y := yKoren; A[y, x] := Desno;
  while A[y, x] <> Prazna do begin
    { Določimo stopnjo trenutne točke in poiščimo kakšnega
      še neobiskanega otroka. }
    Stopnja := 0; dNasl := Prazna; Prvic := true;
    for d := Gor to Desno do begin
      xx := x + DX[d]; yy := y + DY[d];
      if not (A[yy, xx] in[Pregledana, Neobiskana]) then continue;
      Stopnja := Stopnja + 1;
      if A[yy, xx] = Neobiskana then dNasl := d else Prvic := false;
    end; {for d}
    if Stopnja = 0 then { Smo v listu. }
      StMest[Poddrevo] := StMest[Poddrevo] + 1
    else if Stopnja > 1 then begin { Smo v razvejišču. }
      if Poddrevo = Nic then xRazcep := x { Mesto prvega spora med Y in M. }
      else if Prvic then StMest[Poddrevo] := StMest[Poddrevo] + 1;
    end; {if}
    if dNasl <> Prazna then begin { Spustimo se naprej po drevesu. }
      if (x = xRazcep) and (y = yKoren) then { Zapomnimo si, v katero poddrevo }
        if dNasl = Dol then Poddrevo := Muhammad                    { gremo zdaj. }
        else Poddrevo := Yusuf;
      x := x + DX[dNasl]; y := y + DY[dNasl];
      A[y, x] := dNasl;
    end else begin { Vrnimo se nazaj v očeta trenutne točke. }
      d := A[y, x]; A[y, x] := Pregledana;
      if (x = xRazcep) and (y = yKoren) then Poddrevo := Nic;
      x := x − DX[d]; y := y − DY[d];
    end; {if}
  end; {while}
  { Izpišimo rezultat. }
  Assign(F, 'berberi.out'); Rewrite(F);
  WriteLn(F, StMest[Yusuf], ' ', StMest[Muhammad]); Close(F);
end. {Berberi}
```

Še rešitev v C-ju:

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

#define MaxW 3000
#define MaxH 3000

typedef enum { Gor, Levo, Dol, Desno, Prazna, Pregledana, Neobiskana } CelicaT;
typedef enum { Nic, Yusuf, Muhammad } KdoT;
const int DX[4] = { 0, −1, 0, 1 };
const int DY[4] = { −1, 0, 1, 0 };
CelicaT a[MaxH + 2][MaxW + 2];

int main()
{
  FILE *f; char c; KdoT poddrevo; bool prvic; CelicaT d, dNasl;
  int w, h, x, y, xx, yy, yKoren, xRazcep, stopnja, stMest[3];

  /* Preberimo vhodne podatke. */
  f = fopen("berberi.in", "rt");
  fscanf(f, "%d %d\n", &w, &h);
  for (x = 1; x <= w; x++) { a[0][x] = Prazna; a[h + 1][x] = Prazna; }
  for (y = 1; y <= h; y++) { a[y][0] = Prazna; a[y][w + 1] = Prazna; }
  yKoren = −1; xRazcep = −1;
```

```
  for (y = 1; y <= h; y++) {
    for (x = 1; x <= w; x++) {
      c = (char) fgetc(f); a[y][x] = (c == '.') ? Prazna : Neobiskana;
      if (x == 1 && c == 'X') yKoren = y; }
    c = (char) fgetc(f); }
  fclose(f);

  /* Preglejmo drevo. */
  stMest[Yusuf] = 0; stMest[Muhammad] = 0; poddrevo = Nic;
  x = 1; y = yKoren; a[y][x] = Desno;
  while (a[y][x] != Prazna)
  {
    /* Določimo stopnjo trenutne točke in poiščimo kakšnega
       še neobiskanega otroka. */
    stopnja = 0; dNasl = Prazna; prvic = true;
    for (d = Gor; d <= Desno; d++)
    {
      xx = x + DX[d]; yy = y + DY[d];
      if (a[yy][xx] != Pregledana && a[yy][xx] != Neobiskana) continue;
      stopnja++;
      if (a[yy][xx] == Neobiskana) dNasl = d;
      else prvic = false;
    }
    if (stopnja == 0) /* Smo v listu. */
      stMest[poddrevo]++;
    else if (stopnja > 1) { /* Smo v razvejišču. */
      if (poddrevo == Nic) xRazcep = x; /* Mesto prvega spora med Y in M. */
      else if (prvic) stMest[poddrevo]++; }
    if (dNasl != Prazna) { /* Spustimo se naprej po drevesu. */
      if (x == xRazcep && y == yKoren) /* Zapomnimo si, v katero poddrevo */
        poddrevo = (dNasl == Dol) ? Muhammad : Yusuf;      /* gremo zdaj. */
      x += DX[dNasl]; y += DY[dNasl];
      a[y][x] = dNasl; }
    else { /* Vrnimo se nazaj v očeta trenutne točke. */
      d = a[y][x]; a[y][x] = Pregledana;
      if (x == xRazcep && y == yKoren) poddrevo = Nic;
      x -= DX[d]; y -= DY[d]; }
  }

  /* Izpišimo rezultat. */
  f = fopen("berberi.out", "wt");
  fprintf(f, "%d %d\n", stMest[Yusuf], stMest[Muhammad]); fclose(f); return 0;
}
```

3. Piskrc špagetov

Radi bi, da bi bil na koncu tudi najdaljši špaget čim krajši; torej je pametno pri vsakem lomljenju prelomiti najdaljši špaget. Lepo je, da so v vhodnih podatkih škatle s špageti že urejene po dolžini, tako da ni težko vedeti, kje so najdaljši špageti, pri katerih moramo začeti z lomljenjem. Z lomljenjem pa nam nastajajo krajši špageti; če imamo škatlo m špagetov dolžine d, nam ob lomljenju nastane $2m$ špagetov dolžine $d/2$, če je d sod; če pa je d lih, nam ob lomljenju nastane m špagetov dolžine $(d+1)/2$ in m špagetov dolžine $(d-1)/2$. Nalogo lahko torej načeloma rešimo s takšnim postopkom:

naj bo S seznam škatel špagetov, urejen padajoče po dolžini špagetov;

dokler je $b > 0$ (torej dokler lahko še kaj prelomimo):
  če je v prvi škatli (tisti z najdaljšimi špageti) več kot b špagetov
    ali pa so ti špageti dolžine 1, se ustavimo;
  vzemi (in izbriši) prvo škatlo iz seznama;
  zmanjšaj b za število špagetov v tej škatli;
  prelomi špagete v tej škatli in iz njih naredi eno ali dve novi škatli
    (odvisno od tega, ali je njihova dolžina soda ali liha), ki ju dodaj
    na pravo mesto v S (tako da ostane urejen padajoče po dolžini);
    (če je v S že neka škatla s špageti primerne dolžine, pa nove
    špagete zgolj dodaj vanjo, namesto da vrivaš novo škatlo v seznam);
izpiši dolžino špagetov v prvi škatli seznama S;

Neugodno pri tem postopku je, da je vrivanje novih škatel v seznam S načeloma lahko časovno precej potratna operacija, če se je lotimo preveč naivno — npr. če bomo hranili seznam v tabeli in bo treba vsakič, ko bomo vanj vriniti novo škatlo, premakniti vse nadaljnje škatle za eno celico naprej po tabeli. V seznamu $O(n)$ škatel bi vsako tako vrivanje trajalo $O(n)$ časa. Izmed naših desetih testnih primerov bi takšna rešitev verjetno dovolj hitro rešila le prva dva (pri katerih je $n \leq 10^4$; drugod gre lahko n do 10^6).

Lažje je, če hranimo seznam kot verigo, v kateri so elementi povezani med seboj s kazalci; potem moramo paziti le še na to, da bomo dovolj hitro našli mesto, kamor je v seznamu treba vriniti novi element. Če bomo vsakič prečesali ves seznam od začetka naprej, da bi ugotovili, kam vriniti novi element (ki predstavlja skupino pravkar prelomljenih špagetov), nam bo to spet vzelo preveč časa. Na srečo si lahko pomagamo z dejstvom, da škatle ves čas obravnavamo po padajoči dolžini, zato tudi dolžine špagetov po prelamljanju tvorijo padajoče zaporedje. Naslednje vrivanje bo torej prišlo na istem mestu kot prejšnje ali pa nekje kasneje. Tako bomo s kazalcem, ki označuje mesto vrivanja škatel z razpolovljenimi špageti, hodili ves čas le naprej po seznamu, tako da se ta kazalec pri celotnem postopku premakne največ tolikokrat, kolikor je elementov, ki smo jih kdajkoli dodali v seznam. V povprečju nam torej vsako lomljenje vzame $O(1)$ časa; če smo začeli z n škatlami dolžine največ d, bomo lahko vsako škatlo razpolovili največ $O(\log_2 d)$-krat, torej bo časovna zahtevnost celotnega postopka le še $O(n \log d)$.

Še ena različica te rešitve pa je, da prelomljenih špagetov ne vrivamo v prvotni seznam S, pač pa v nek nov seznam S'. Kot smo opazili že zgoraj, tvorijo dolžine prelomljenih špagetov padajoče zaporedje, zato bomo v seznam S' dodajali ves čas le na koncu. Ko pa se odločamo, katero škatlo špagetov bi lomili v naslednjem koraku, pogledamo prvo iz seznama S in prvo iz seznama S' ter vzamemo tisto, v kateri so daljši špageti.

To različico rešitve smo uporabili tudi v spodnjem programu. Izkoristimo lahko tudi opažanje, da S' ne bo nikoli vseboval več kot $2n+1$ škatel hkrati, zato ga lahko hranimo kar v krožni tabeli (*ring buffer*) z $2n + 1$ elementi. To zgornjo mejo nam pokaže naslednji preprosti razmislek: recimo, da začnemo s škatlo dolžine 23; ta nam razpade v škatli 12 in 11; ko prelomimo ti dve, nam iz 12 nastane škatla 6, iz 11 pa 6 in 5; ko prelomimo 6 in 5, nam iz 6 nastane 3, iz 5 pa 3 in 2; škatle nam torej ves čas nastopajo v skupinah po največ dve in obe škatli v skupini se po dolžini razlikujeta za največ 1: $\{23\}, \{12, 11\}, \{6, 5\}, \{3, 2\}$ itd. Preden se lotimo lomljenja

naslednje skupine, smo morali prelomiti že vse špagete iz prejšnje skupine, zato je v seznamu S' prisotna v vsakem trenutku le ena od teh skupin in mogoče še ena škatla iz prejšnje skupine. Ker smo začeli z n škatlami seznama S, nam zdaj ta razmislek pove, da imamo v S' največ n takih parov škatel in za povrhu še eno škatlo iz para, v katerem smo doslej prelomili šele eno škatlo, tisto drugo pa bomo v naslednjem lomljenju. Tako imamo v S' največ $2n + 1$ škatel.[10]

```
program Spageti;
const MaxN = 1000000;
var
  n, b, i, j, d, d2, m, m2: integer;
  mi, di: array [0..MaxN − 1] of integer; lz2: boolean;
  mi2, di2: array [0..2 * MaxN] of integer; i2, j2, n2, qMod: integer;
  F: text;
begin
  { Preberimo vhodne podatke. }
  Assign(F, 'spageti.in'); Reset(F); ReadLn(F, n, b);
  j := 0;
  for i := 0 to n − 1 do begin
    ReadLn(F, mi[j], di[j]);
    if j > 0 then if di[j] = di[j − 1] then { več škatel enake dolžine združimo }
      begin mi[j − 1] := mi[j − 1] + mi[j]; continue end;
    j := j + 1;
  end; {for i}
  n := j; qMod := 2 * n + 1; { največja možna dolžina seznama S' }
  { Lomimo špagete. }
  i2 := 1; j2 := 0; n2 := 0;
  while true do begin
    { V katerem seznamu je najdaljša škatla, S ali S'? }
    if n2 <= 0 then lz2 := false
    else if j <= 0 then lz2 := true
    else lz2 := di2[i2] >= di[j − 1];
    if not lz2 then begin { Lomimo najdaljšo škatlo iz S. }
      j := j − 1; m := mi[j]; d := di[j]
    end else begin { Lomimo najdaljšo škatlo iz S'. }
      m := mi2[i2]; d := di2[i2]; i2 := (i2 + 1) mod qMod; n2 := n2 − 1;
```

[10] Malo bolj formalen dokaz lahko sestavimo s pomočjo indukcije. Trdimo, da za seznam S' skoraj ves čas velja naslednje: množico vseh dolžin škatel v S' lahko dobimo tako, da vzamemo največ k različnih dolžin $d_1, \ldots, d_k$, za nekatere izmed teh dolžin (lahko tudi nobene) dodamo še dolžino $d_i + 1$; pri tem je k število škatel iz seznama S, ki smo jih že prelomili.

Prepričajmo se, da to res drži. Na začetku je S' prazen in trditev zanj drži. Recimo zdaj, da drži pred nekim lomljenjem; radi bi se prepričali, da drži tudi po njem. (1) Če smo prelomili neko škatlo dolžine d iz seznama S, se k poveča za 1, v S' pa prideta na konec največ dve novi škatli dolžine $\lfloor d/2 \rfloor$ in $\lfloor d/2 \rfloor + 1$ (slednja le, če je d lih), torej trditev še vedno drži. (2) Druga možnost pa je, da prelomimo neko škatlo iz S'. (2.1) Če je ta škatla del para $\{2r, 2r + 1\}$, pomeni, da zdaj lomimo $2r + 1$ in na konec seznama dodamo $r + 1$ in r; seznam je zdaj začasno za en element daljši kot prej; toda naslednje lomljenje bo prelomilo škatlo $2r$ (iz S'; če pa je tudi v S taka škatla, bomo v isti sapi prelomili še njo) in iz nje ne bo naredilo nobene nove škatle, ker se bo dalo nove kose špagetov kar dodati v obstoječo škatlo dolžine r. Trditev zdaj spet drži. (2.2) Če pa smo prelomili škatlo iz para $\{2r - 1, 2r\}$, smo torej najprej prelomili $2r$ in dodali na konec seznama škatlo r; naslednje lomljenje pa iz $2r - 1$ naredi še škatlo $r - 1$ in doda nekaj špagetov v škatlo r; trditev zdaj spet drži. (2.3) Če je dolžina pravkar prelomljene škatle ena od tistih, ki ne nastopajo v paru, iz nje zdaj bodisi nastane ena škatla pol krajših špagetov ali pa par škatel z dolžinama oblike $\lfloor d/2 \rfloor$ in $\lfloor d/2 \rfloor + 1$, ampak trditev v vsakem primeru spet drži.

Tako torej vidimo, da če naša trditev v nekem trenutku drži, bo po največ dveh nadaljnjih prelomih spet držala, vmes pa bo dolžina seznama S' nikoli ne bo več kot za eno škatlo daljša kot zdaj.

```
      { Če so enako dolgi špageti tudi v S, bomo v isti sapi lomili še njih. }
      if j > 0 then if di[j − 1] = d then begin j := j − 1; m := m + mi[j] end;
    end; {if}
    if (d = 1) or (m > b) then break; { ne moremo prelomiti te skupine }
    b := b − m;
    { Prelomili smo m špagetov dolžine d. Nastane skupina m2 špagetov dolžine d2. }
    d2 := (d + 1) div 2; if Odd(d) then m2 := m else m2 := 2 * m;
    { Mogoče je na koncu S' že škatla za to dolžino. }
    if n2 > 0 then if di2[j2] = d2 then begin mi2[j2] := mi2[j2] + m2; d2 := −1 end;
    if d2 > 0 then begin { Če pa ni, dodajmo zdaj novo škatlo. }
      j2 := (j2 + 1) mod qMod;
      mi2[j2] := m2; di2[j2] := d2; n2 := n2 + 1;
    end; {if}
    { Če je bil d lih, nastane še ena skupina malo krajših špagetov dolžine d div 2. }
    if Odd(d) then begin { Ta bo zagotovo potrebovala novo škatlo. }
      d2 := d div 2; j2 := (j2 + 1) mod qMod;
      mi2[j2] := m2; di2[j2] := d2; n2 := n2 + 1;
    end; {if}
  end; {while}
  { Izpišimo rezultat. }
  Assign(F, 'spageti.out'); Rewrite(F); WriteLn(F, d); Close(F);
end. {Spageti}
```

Še rešitev v C-ju:

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

#define MaxN 1000000
int mi[MaxN], di[MaxN], mi2[2 * MaxN + 1], di2[2 * MaxN + 1];

int main()
{
  int n, b, i, j, d, d2, m, m2, i2, j2, n2, qMod; bool iz2;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("spageti.in", "rt");
  fscanf(f, "%d %d", &n, &b);
  for (i = 0, j = 0; i < n; i++)
  {
    fscanf(f, "%d %d", &mi[j], &di[j]);
    if (j > 0 && di[j] == di[j − 1]) /* več škatel enake dolžine združimo */
      { mi[j − 1] += mi[j]; continue; }
    j++;
  }
  n = j; qMod = 2 * n + 1; /* največja možna dolžina seznama S' */

  /* Lomimo špagete. */
  i2 = 1; j2 = 0; n2 = 0;
  for ( ; ; )
  {
    /* V katerem seznamu je najdaljša škatla, S ali S'? */
    if (n2 <= 0) iz2 = false;
    else if (j <= 0) iz2 = true;
    else iz2 = (di2[i2] >= di[j − 1]);
    if (! iz2) /* Lomimo najdaljšo škatlo iz S. */
      { j−−; m = mi[j]; d = di[j]; }
    else { /* Lomimo najdaljšo škatlo iz S'. */
      m = mi2[i2]; d = di2[i2]; i2 = (i2 + 1) % qMod; n2−−;
```

```
      /* Če so enako dolgi špageti tudi v S, bomo v isti sapi lomili še njih. */
      if (j > 0 && di[j − 1] == d) m += mi[−−j]; }
    if (d == 1 || m > b) break; /* ne moremo prelomiti te skupine */
    b −= m;

    /* Prelomili smo m špagetov dolžine d. Nastane skupina m2 špagetov dolžine d2. */
    d2 = (d + 1) / 2; if (d & 1) m2 = m; else m2 = 2 * m;
    /* Mogoče je na koncu S' že škatla za to dolžino. */
    if (n2 > 0 && di2[j2] == d2) mi2[j2] += m2;
    else { /* Če pa ni, dodajmo zdaj novo škatlo. */
      j2 = (j2 + 1) % qMod; mi2[j2] = m2; di2[j2] = d2; n2++; }
    /* Če je bil d lih, nastane še ena skupina malo krajših špagetov dolžine d / 2. */
    if (d & 1) { /* Ta bo zagotovo potrebovala novo škatlo. */
      d2 = d / 2; j2 = (j2 + 1) % qMod;
      mi2[j2] = m2; di2[j2] = d2; n2++; }
  }
  /* Izpišimo rezultat. */
  f = fopen("spageti.out", "wt"); fprintf(f, "%d\n", d); fclose(f); return 0;
}
```

Rešitev s tabelo špagetov po dolžini. Nalogo lahko rešimo tudi tako, da namesto seznama škatel vzdržujemo tabelo, v kateri je za vsako možno dolžino špageta navedeno, koliko špagetov te dolžine imamo. Potem se ni težko zapeljati po tej tabeli od daljših špagetov proti krajšim in jih lomiti; ko neko skupino enako dolgih špagetov prelomimo, moramo ustrezno povečati enega ali dva elementa tabele, da upoštevamo na novo nastale polovičke pravkar razpolovljenih špagetov. Če so bili v vhodni datoteki špageti dolgi največ s, bo ta rešitev porabila $O(d)$ pomnilnika za hranjenje tabele in tudi $O(d)$ časa za pregled te tabele. Pri naših testnih primerih bi bila torej ta rešitev primerna za prve štiri (od desetih), saj pri njih naloga zagotavlja, da bo $d \leq 10^6$. Pri ostalih testnih primerih pa so bile dolžine najdaljših špagetov čez 950 milijonov, torej bi ta rešitev porabila preveč pomnilnika in preveč časa.

```
program Spageti;
const MaxN = 1000000; MaxD = 1000000;
type TabelaT = packed array [0..MaxD] of integer;
var T: ↑TabelaT; F: text;
    n, b, i, m, d, dNaj: integer;
    mi, di: array [1..MaxN] of integer;
begin
  { Preberimo vhodne podatke. }
  Assign(F, 'spageti.in'); Reset(F); ReadLn(F, n, b);
  dNaj := 0;
  for i := 1 to n do begin
    ReadLn(F, mi[i], di[i]);
    if di[i] > dNaj then dNaj := di[i];
  end; {for i}
  dNaj := dNaj + 1;

  { Pripravimo tabelo s številom špagetov za vsako dolžino. }
  GetMem(T, sizeof(integer) * dNaj);
  for i := 1 to dNaj − 1 do T↑[i] := 0;
  for i := 1 to n do T↑[di[i]] := T↑[di[i]] + mi[i];
```

```
  { Lomimo špagete. }
  d := dNaj − 1;
  while d > 2 do begin
    m := T↑[d];
    if m > 0 then begin
      { Imamo m špagetov dolžine d; prelomimo jih. }
      if m > b then break; { Preveč jih je. }
      b := b − m;
      T↑[d div 2] := T↑[d div 2] + m;
      T↑[(d + 1) div 2] := T↑[(d + 1) div 2] + m;
    end; {if}
    d := d − 1;
  end; {while}
  FreeMem(T, sizeof(integer) * dNaj);

  { Izpišimo rezultat. }
  Assign(F, 'spageti.out'); Rewrite(F); WriteLn(F, d); Close(F);
end. {Spageti}
```

Oglejmo si to rešitev še v C-ju:

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

#define MaxN 1000000
int mi[MaxN], di[MaxN];

int main()
{
  int n, b, i, j, d, dNaj, m, *T;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("spageti.in", "rt");
  fscanf(f, "%d %d", &n, &b);
  for (i = 0, dNaj = 0; i < n; i++)
  {
    fscanf(f, "%d %d", &mi[i], &di[i]);
    if (di[i] > dNaj) dNaj = di[i];
  }

  /* Pripravimo tabelo s številom špagetov za vsako dolžino. */
  T = (int *) malloc((dNaj + 1) * sizeof(int));
  for (d = 0; d <= dNaj; d++) T[d] = 0;
  for (i = 0; i < n; i++) T[di[i]] += mi[i];

  /* Lomimo špagete. */
  for (d = dNaj; d > 1; d−−)
  {
    m = T[d]; if (m <= 0) continue;
    if (m > b) break;
    b −= m;
    T[d / 2] += m;
    T[(d + 1) / 2] += m;
  }
  free(T);

  /* Izpišimo rezultat. */
  f = fopen("spageti.out", "wt");
  fprintf(f, "%d\n", d); fclose(f); return 0;
}
```

Dokaz pravilnosti. Postopek, ki smo ga uporabili za lomljenje špagetov v naši rešitvi, sodi v družino požrešnih (*greedy*) algoritmov. Za konec si oglejmo malo bolj formalen dokaz za to, da nas opisani pristop (ki v vsakem koraku prelomi najdaljšega izmed vseh preostalih špagetov) res pripelje do najboljše rešitve. Kot je pri požrešnih algoritmih v navadi, se lahko takšnega dokaza tudi v našem primeru lotimo tako, da predpostavimo, da bi obstajala neka rešitev, še boljša od naše požrešne; nato se bomo prepričali, da to pripelje v protislovje oz. da bi lahko tisto optimalno rešitev korak za korakom predelali tako, da bi na koncu postala enaka naši požrešni, pri tem pa se med predelavo ne bi nikoli poslabšala.

Vsako zaporedje lomljenj špagetov lahko predstavimo tako, da ob vsakem lomljenju zapišemo dolžino špageta, ki smo ga pravkar prelomili. Naj bo $a = (a_1, ..., a_k)$ zaporedje dolžin, ki ga dobimo, če vsakič prelomimo najdaljši špaget. Recimo zdaj, da ta rešitev ni najboljša; torej obstaja neko drugo zaporedje lomljenj, recimo $c = (c_1, \ldots, c_r)$, pri katerem je na koncu najdaljši preostali špaget krajši kot pri našem zaporedju a. Ker sta zaporedji različni, se mogoče v prvih nekaj členih ujemata, prej ali slej pa se mora pojaviti neko neujemanje. To se lahko zgodi na naslednje načine:

(1) Mogoče se a konča prej kot c. Toda a se konča šele v trenutku, ko ni mogoče prelomiti nobenega špageta več, bodisi ker so ostali le še špageti dolžine 1 ali pa ker smo prelomili že b špagetov. V vsakem primeru torej, če se je c doslej ujemal z a, tudi pri c ni mogoče nadaljevati z lomljenjem; torej ta možnost odpade.

(2) Mogoče se c konča prej kot a. Toda ker sta se do sem obe zaporedji ujemali, imamo pri c zdaj enak kup špagetov kot do tega mesta pri a in ker a nadaljuje z lomljenjem, bo imel na koncu kvečjemu še boljšo rešitev kot c, ne pa slabše. Torej tudi ta možnost odpade.

(3) Ostane torej le še možnost, da se obe zaporedji nadaljujeta, vendar istoležni elementi niso več enaki; naj bo torej i prvi indeks, pri katerem je $a_i \neq c_i$. Če je možnih več enako dobrih optimalnih rešitev c, izberimo med njimi tisto, ki ima največjo vrednost i (torej pri kateri do neujemanja z a pride čim kasneje). Ker a vedno prelomi najdaljši špaget, gotovo velja $a_i > c_i$. Zdaj ločimo dve možnosti.

(3.1) Mogoče bo najdaljši špaget po koncu zaporedja a dolg a_i. To pomeni, da je po i-tem lomljenju še toliko špagetov dolžine a_i, da zaporedje a v preostalih $k - i$ lomljenjih ni uspelo prelomiti vseh teh špagetov. Ker se a ustavi šele, ko z lomljenjem ne more nadaljevati (ker je izvedel že b lomljenj), in ker je c doslej lomil enako kot a, lahko sklepamo, da tudi c ne bo mogel prelomiti vseh špagetov dolžine a_i (ker tudi c ne sme izvesti več kot b lomljenj), torej na koncu njegova rešitev ne bo nič boljša od a-jeve; prišli smo v protislovje.

(3.2) Mogoče pa bo najdaljši špaget po koncu zaporedja a krajši od a_i. To pomeni, da a prej ali slej prelomi vse preostale špagete dolžine a_i. Če hoče c dobiti na koncu boljšo rešitev kot a, mora tudi on sčasoma prelomiti vse preostale špagete dolžine a_i, vključno s tistim, ki ga je a prelomil v i-tem lomljenju, ko je c prelomil krajši špaget c_i. Recimo da c špaget a_i prelomi nekje kasneje, ob času j (torej je $c_j = a_i$); c je torej oblike $(a_1, \ldots, a_{i-1}, c_i, c_{i+1}, \ldots, c_{j-1}, a_i, c_{j+1}, \ldots, c_r)$. Rešitev c se nič ne poslabša, če špaget a_i prelomimo že ob času i, ne šele ob času j, torej če c predelamo v $(a_1, \ldots, a_{i-1}, a_i, c_i, c_{i+1}, \ldots, c_{j-1}, c_{j+1}, \ldots, c_r)$. Tako predelana rešitev je še vedno optimalna, poleg tega pa se z a ujema v prvih i členih, ne le v prvih $i-1$

členih; mi pa smo prej rekli, da smo c izbrali tako, da se ujema z a v maksimalno veliko členih; torej smo spet prišli v protislovje.

Tako torej vidimo, da bi nas obstoj boljše rešitve od naše požrešne v vsakem primeru pripeljal v protislovje.

4. Redki nizi

Niz s je pri tej nalogi lahko zelo dolg — do 10^9 znakov — ampak večino tega niza sestavljajo le znaki c. Kaj drugega se lahko v s-ju pojavlja le tam, kjer smo čezenj napisali nize t_i, ampak ti nizi so vsi skupaj dolgi največ 10^6 znakov. Tako je torej s sestavljen iz dolgih skupin samih c-jev, ki jih tu in tam prekinja kakšen kratek podniz, ki je nastal iz nizov t_i.

Niz iz primera v besedilu naloge lahko na primer v mislih razdelimo takole:

$$s = \texttt{xxxxxx}\ \underline{\texttt{fghxjxfg}}\ \texttt{xx}\ \underline{\texttt{xfg}}\ \texttt{x}.$$

Podčrtani so tisti deli, ki so nastali iz t_i-jev. Spomnimo se, da je bil „privzeti“ znak c v tem primeru $c = \texttt{x}$.

Lepo pri takšni predstavitvi niza s je, da tistih delov, ki jih sestavljajo sami c-ji, ni treba predstavljati eksplicitno, pač pa je dovolj že, če poznamo njihovo dolžino. Pa tudi iskanje podniza p v takem delu s-ja je zelo preprosto: če je p sestavljen iz samih c-jev, se v nizu $ccc\ldots c$ pojavlja na vsakem možnem mestu, sicer pa v nobenem. Malo resneje se bomo morali z iskanjem p-ja ukvarjati le v tistih delih s-ja, ki so nastali iz nizov t_i, in na meji med temi deli in tistimi deli, ki so sestavljeni iz samih c-jev.

Najprej pa si oglejmo, kako sploh priti do želene predstavitve niza s. Recimo, da je niz t_i dolg m_i znakov in ga vpišemo v s z začetkom na indeksu z_i. To pomeni, da t_i pokrije v nizu s znake od z_i do $z_i + m_i - 1$. Ti intervali se pri različnih t_i-jih lahko prekrivajo, mi pa bi jih radi zdaj združili. (Tako je na primer v primeru iz besedila naloge niz t_1 pokril v s-ju znake 10..14, niz t_2 pa znake 7..12; iz obeh skupaj je tako nastal kos `fghxjxfg`, ki smo ga videli zgoraj in ki pokriva v s-ju znake 7..14.) Združevanja se lahko lotimo takole:

naj bo U prazen seznam;
pregleduj nize t_i po naraščajočem začetnem položaju z_i:
  (* *t_i torej pokriva interval od z_i do $z_i + m_i - 1$* *)
  če je U še prazen, dodaj vanj interval $z_i..(z_i + m_i - 1)$;
  naj bo r indeks, pri katerem se konča zadnji interval iz U;
  če je $z_i > r + 1$:
    dodaj $z_i..z_i + m_i - 1$ kot nov interval na konec U-ja;
  sicer:
    (* *t_i-jev interval združimo z zadnjim intervalom iz U* *)
    postavi konec zadnjega U-jevega intervala na $\max\{r, z_i + m_i - 1\}$;

Spotoma si je pametno tudi pri vsakem t_i zapisati, v kateri interval seznama U je prišel (recimo up_i) in kam znotraj tega intervala je prišel (torej kako daleč za začetkom intervala je začetek niza t_i; recimo $\mathsf{uz}_i - 1$ znakov za začetkom intervala). Pregledovanje nizov t_i po naraščajočem z_i pomeni, da moramo nize najprej urediti po z_i, kar pa ni težko, ker jih je le $k \leq 10^4$, torej si lahko privoščimo tudi

uporabo kakšnega od preprostih in neučinkovitih algoritmov za urejanje (s časovno zahtevnostjo $O(k^2)$).

Ko imamo seznam U pripravljen, si za vsak interval iz njega alocirajmo primerno velik blok pomnilnika (dolg toliko znakov, kolikor jih pač pokriva ta interval) in te bloke za začetek zapolnimo z znaki c. Nato se sprehodimo po vseh t_i-jih (v prvotnem vrstnem redu, torej v takem, v kakršnem naloga zahteva, da jih vpisujemo v s) in vsakega vpišimo na primerno mesto v blok, ki pripada tistemu intervalou iz U, ki pokriva niz t_i (v ta namen si pomagamo s prej omenjenimi vrednostmi up_i in uz_i, ki smo si jih shranili ob tvorbi seznama U).

Zdaj smo pripravljeni na iskanje podniza p v nizu s. Pojdimo od leve proti desni po intervalih iz U-ja. Pri vsakem intervalu vemo, kako dolga skupina c-jev je med njim in prejšnjim intervalom; pojavitve c-ja tam notri znamo prešteti, kot smo videli že zgoraj. Zdaj se lotimo iskanja tistih pojavitev p-ja, ki se vsaj delno pokrivajo z trenutnim intervalom, ne pa s kakšnim od bolj levo ležečih intervalov. Recimo, da trenutni inteval pokriva v s-ju indekse od uz_j do uk_j, prejšnji interval pa je pokrival indekse do uk_{j-1} (če je trenutni interval prvi, si mislimo $\mathsf{uk}_0 = 0$). Potem nas zdaj zanimajo pojavitve p-ja z začetnim položajem vsaj $\max\{\mathsf{uz}_{j-1}+1, \mathsf{uz}_j-|p|+1\}$ (da ne segajo tako daleč levo, da bi padle še v prejšnji interval U-ja ali pa bi se končale pred začetkom trenutnega intervala) in kvečjemu uk_j (da se ne začnejo šele za koncem trenutnega intervala). Za vsakega od teh možnih začetnih položajev p-ja lahko preverimo, ali se p pojavlja na tem mestu v s, kar na staromodni način, torej tako, da niza primerjamo znak za znakom; če opazimo neujemanje, še preden pridemo do konca p-ja, pa pač obupamo in se posvetimo naslednjemu možnemu začetnemu položaju p-ja v s-j. Omejitve v nalogi (p je dolg največ 100 znakov, v U pa je največ 10^4 intervalov in vsi skupaj so dolgi največ 10^6 znakov) so podane tako, da nam ni treba komplicirati s kakšnim od učinkovitejših (vendar bolj zapletenih) algoritmov za iskanje podnizov v nizih (npr. Knuth-Morris-Prattovim ali Boyer-Moorovim).

Pri preverjanju, ali se p pojavlja na nekem izbranem začetnem položaju v s ali ne, je treba paziti le še na naslednje: neka pojavitev p-ja, ki se začne v trenutnem intervalu seznama U, lahko sega tudi še čez rob tega intervala, in to ne le v prostor med tem intervalom in naslednjim, pač pa tudi še čez enega ali več naslednjih intervalov (če je p dolg, intervali iz U pa kratki in razmeroma blizu skupaj). Ko primerjamo znake p-ja (pri nekem izbranem začetnem položaju) z znaki s-ja in se premikamo istočasno naprej po obeh nizih, moramo imeti v mislih, da premikanje naprej po s-ju v bistvu pomeni, da beremo znake iz trenutnega intervala iz seznama U, nato c-je iz območja med trenutnim in naslednjim intervalom, nato beremo znake iz naslednjega intervala in tako naprej. Za te reči v spodnjem programu skrbi funkcija SePojavlja.

```
program Nizi;
const MaxK = 10000; MaxDolzT = 1000000;
var
  p: string;
  u, t: packed array [0..MaxDolzT − 1] of char;
  F: text;
  i, j, k, n, r, pc, pd, DolzT, DolzU, StPojavitev, cPred: integer;
  { Niz t_i je shranjen v t[tp[i]..tp[i] + td[i] − 1].
    Podobno je del niza s, namreč s[uz[i]..uz[i] + ud[i] − 1],
    shranjen v tabeli u na mestih u[up[i]..up[i] + ud[i] − 1].
    Takih kosov s-ja je r. }
```

```
  z, td, tp, uz, ud, up, tu: array [1..MaxK] of integer; c, cc: char;
  zu, zi: array [1..MaxK] of integer; { zu[j] = z[zi[j]] in tabela zu je urejena naraščajoče }

  function SePojavlja(Kos, j: integer): boolean;
  var i: integer; sc: char;
  begin
    SePojavlja := false; if j + pd − 1 > n then exit;
    i := 1;
    while i <= pd do begin
      if Kos > r then sc := c
      else if j >= uz[Kos] + ud[kos] then begin
        Kos := Kos + 1; continue
      end else if j < uz[Kos] then sc := c
      else sc := u[up[Kos] + j − uz[Kos]];
      if p[i] <> sc then exit;
      i := i + 1; j := j + 1;
    end; {for i}
    SePojavlja := true;
  end; {SePojavlja}

begin
  { Preberimo vhodne podatke. }
  Assign(F, 'nizi.in'); Reset(F);
  ReadLn(F, n, k); ReadLn(F, c);
  DolzT := 0;
  for i := 1 to k do begin
    Read(F, z[i]); Read(F, cc);
    td[i] := 0; tp[i] := DolzT;
    while not Eoln(F) do begin
      Read(F, t[DolzT]); td[i] := td[i] + 1; DolzT := DolzT + 1;
    end; {while}
    ReadLn(F);
  end; {for i}
  ReadLn(F, p); pd := Length(p);
  Close(F);
  { Koliko c-jev je na začetku p-ja? }
  pc := 0; while pc < pd do if p[pc + 1] = c then pc := pc + 1 else break;
  { Upoštevajmo, da se t-ji lahko prekrivajo. Uredimo jih po začetkih. }
  for i := 1 to k do begin
    j := i − 1;
    while j > 0 do if zu[j] <= z[i] then break
      else begin zi[j + 1] := zi[j]; zu[j + 1] := zu[j]; j := j − 1 end;
    zi[j + 1] := i; zu[j + 1] := z[i];
  end; {for i}
  { Tam, kjer se več t-jev prekriva ali dotika, bo nastal en sam daljši kos. }
  r := 0; j := 1; DolzU := 0;
  for i := 1 to k do begin
    { Trenutni kos se konča pri j − 1; trenutni t se začne pri zu[i]. }
    if zu[i] < j then begin { Podaljšajmo trenutni kos, če je treba. }
      if zu[i] + td[zi[i]] > j then j := zu[i] + td[zi[i]];
      ud[r] := j − uz[r];
    end else begin { Začnimo nov kos. }
      if r > 0 then DolzU := DolzU + ud[r];
      r := r + 1; up[r] := DolzU; uz[r] := zu[i];
      ud[r] := td[zi[i]]; j := uz[r] + ud[r];
    end; {if}
```

```
      { Kam v tabelo u bomo morali skopirati trenutni t? }
      tu[zi[i]] := up[r] + (zu[i] − uz[r]);
   end; {for i}
   DolzU := DolzU + ud[r];
   { Skopirajmo zdaj t-je v tabelo u. }
   for i := 1 to k do
      for j := 1 to td[i] do
         u[tu[i] + j − 1] := t[tp[i] + j − 1];
   { Poiščimo zdaj pojavitve niza v p v s. }
   StPojavitev := 0;
   for i := 1 to r do begin
      { i-ti kos niza s obsega znake s[uz[i]..uz[i] + ud[i] − 1],
        dejansko pa so shranjeni v u[up[i]..up[i] + ud[i] − 1].
        Med prejšnjim in tem kosom je cPred znakov c. }
      if i = 1 then j := 1 else j := uz[i − 1] + ud[i − 1];
      cPred := uz[i] − j;
      { Preštejmo pojavitve p-ja med prejšnjim in trenutnim kosom. }
      if (pc = pd) and (cPred >= pd) then StPojavitev := StPojavitev + cPred − pd + 1;
      { Preštejmo pojavitve p-ja, ki se začnejo v trenutnem kosu
        ali pa v vmesnem območju med prejšnjim in trenutnim ter segajo še v trenutnega. }
      if j < uz[i] − pd + 1 then j := uz[i] − pd + 1;
      while j < uz[i] + ud[i] do begin
         if j + pd − 1 > n then break;
         if SePojavlja(i, j) then StPojavitev := StPojavitev + 1;
         j := j + 1;
      end; {while}
   end; {for i}
   { Preštejmo pojavitve p-ja za zadnjim kosom. }
   if pc = pd then begin
      cPred := n + 1 − (uz[r] + ud[r]);
      if cPred >= pd then StPojavitev := StPojavitev + cPred − pd + 1;
   end; {if}
   { Izpišimo rezultat. }
   Assign(F, 'nizi.out'); Rewrite(F); WriteLn(F, StPojavitev); Close(F);
end. {Nizi}
```

Še rešitev v C-ju:

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

#define MaxK 10000
#define MaxDolzT 1000000
#define MaxP 100

char p[MaxP + 1], u[MaxDolzT], t[MaxDolzT + 1];
/* Niz t_i je shranjen v t[tp[i]..tp[i] + td[i] - 1].
   Podobno je del niza s, namreč s[uz[i]..uz[i] + ud[i] - 1],
   shranjen v tabeli u na mestih u[up[i]..up[i] + ud[i] - 1].
   Takih kosov s-ja je r. */
int n, k, r, pd; /* pd = dolžina niza p */
int z[MaxK], td[MaxK], tp[MaxK], uz[MaxK], ud[MaxK], up[MaxK], tu[MaxK];
int zu[MaxK], zi[MaxK]; /* zu[j] = z[zi[j]] in tabela zu je urejena naraščajoče */
char c;

bool SePojavlja(int kos, int j)
{
```

```
   int i; char sc;
   if (j + pd >= n) return false;
   i = 0;
   while (i < pd)
   {
      if (kos >= r) sc = c;
      else if (j >= uz[kos] + ud[kos]) { kos++; continue; }
      else if (j < uz[kos]) sc = c;
      else sc = u[up[kos] + j - uz[kos]];
      if (p[i] != sc) return false;
      i++; j++;
   }
   return true;
}

int main()
{
   int i, j, pc, dolzT, dolzU, stPojavitev, cPred;
   char cc;

   /* Preberimo vhodne podatke. */
   FILE *f = fopen("nizi.in", "rt");
   fscanf(f, "%d %d\n", &n, &k);
   fscanf(f, "%c\n", &c);
   dolzT = 0;
   for (i = 0; i < k; i++)
   {
      fscanf(f, "%d %s\n", &z[i], &t[dolzT]);
      tp[i] = dolzT; z[i]--;
      while (t[dolzT]) dolzT++;
      td[i] = dolzT - tp[i];
   }
   fscanf(f, "%s\n", p); pd = strlen(p);
   fclose(f);

   /* Koliko c-jev je na začetku p-ja? */
   pc = 0; while (pc < pd && p[pc] == c) pc++;
   /* Upoštevajmo, da se t-ji lahko prekrivajo. Uredimo jih po začetkih. */
   for (i = 0; i < k; i++)
   {
      for (j = i - 1; j >= 0 && zu[j] > z[i]; j--)
         zi[j + 1] = zi[j], zu[j + 1] = zu[j];
      zi[j + 1] = i; zu[j + 1] = z[i];
   }
   /* Tam, kjer se več t-jev prekriva ali dotika, bo nastal en sam daljši kos. */
   r = 0; j = 0; dolzU = 0;
   for (i = 0; i < k; i++)
   {
      /* Trenutni kos se konča pri j - 1; trenutni t se začne pri zu[i]. */
      if (zu[i] < j) { /* Podaljšajmo trenutni kos, če je treba. */
         if (zu[i] + td[zi[i]] > j) j = zu[i] + td[zi[i]];
         ud[r - 1] = j - uz[r - 1]; }
      else { /* Začnimo nov kos. */
         if (r > 0) dolzU += ud[r - 1];
         up[r] = dolzU; uz[r] = zu[i];
         ud[r] = td[zi[i]]; j = uz[r] + ud[r]; r++; }
      /* Kam v tabelo u bomo morali skopirati trenutni t? */
      tu[zi[i]] = up[r - 1] + (zu[i] - uz[r - 1]);
```

```
  }
  doIzU += ud[r];
  /* Skopirajmo zdaj t-je v tabelo u. */
  for (i = 0; i < k; i++) for (j = 0; j < td[i]; j++)
    u[tu[i] + j] = t[tp[i] + j];
  /* Poiščimo zdaj pojavitve niza v p v s. */
  stPojavitev = 0;
  for (i = 0; i < r; i++)
  {
    /* i-ti kos niza s obsega znake s[uz[i]..uz[i] + ud[i] - 1],
       dejansko pa so shranjeni v u[up[i]..up[i] + ud[i] - 1].
       Med prejšnjim in tem kosom je cPred znakov c. */
    j = (i == 0) ? 0 : uz[i − 1] + ud[i − 1];
    cPred = uz[i] − j;
    /* Preštejmo pojavitve p-ja med prejšnjim in trenutnim kosom. */
    if (pc == pd && cPred >= pd) stPojavitev += cPred − pd + 1;
    /* Preštejmo pojavitve p-ja, ki se začnejo v trenutnem kosu
       ali pa v vmesnem območju med prejšnjim in trenutnim ter segajo še v trenutnega. */
    if (j < uz[i] − pd + 1) j = uz[i] − pd + 1;
    while (j < uz[i] + ud[i])
    {
      if (j + pd − 1 > n) break;
      if (SePojavlja(i, j)) stPojavitev++;
      j++;
    }
  }
  /* Preštejmo pojavitve p-ja za zadnjim kosom. */
  if (pc == pd) {
    cPred = n − (uz[r − 1] + ud[r − 1]);
    if (cPred >= pd) stPojavitev += cPred − pd + 1; }
  /* Izpišimo rezultat. */
  f = fopen("nizi.out", "wt");
  fprintf(f, "%d\n", stPojavitev); fclose(f); return 0;
}
```

Preprostejša rešitev. Besedilo naloge obljublja, da bo pri petih testnih primerih (od desetih) niz s dolg kvečjemu 10^6 znakov. Takšne primere lahko dovolj učinkovito rešimo tudi s precej preprostejšim postopkom — v pomnilniku si rezervirajmo tabelo n znakov, jo na začetku napolnimo z znaki c in nato vanjo na prava mesta vpisujmo nize $t_1, t_2, \ldots, t_k$. Na koncu imamo torej v pomnilniku čisto eksplicitno predstavljen celoten niz s in lahko po njem iščemo pojavitve niza p s katerim koli od običajnih postopkov za iskanje podniza v nizu. Spodnji program išče pojavitve p-ja na najpreprostejši način: za vsak možni začetni položaj p-ja primerja njegove znake z znaki s-ja in če pride do konca p-ja, ne da bi opazil neujemanje, ve, da je našel pojavitev p-ja v s-ju. Lahko bi poskusili namesto tega uporabiti tudi kakšnega od učinkovitejših postopkov za iskanje podniza v nizu, npr. Knuth-Morris-Prattov ali pa Boyer-Moorov algoritem.

```
program Nizi;
const MaxN = 1000000000;
type NizT = packed array [0..MaxN − 1] of char;
var s: ↑NizT; p: string; c, cc: char; F: text;
    z, n, pd, i, j, k, StPojavitev: integer;
begin
```

```
  { Preberimo vhodne podatke. }
  Assign(F, 'nizi.in'); Reset(F); ReadLn(F, n, k);
  ReadLn(F, c);
  GetMem(s, n); for i := 0 to n − 1 do s↑[i] := c;
  for i := 1 to k do begin
    Read(F, z); z := z − 1;
    Read(F, cc); { presledek }
    while not Eoln(F) do begin Read(F, cc); s↑[z] := cc; z := z + 1 end;
    ReadLn(F);
  end; {for i}
  ReadLn(F, p); pd := Length(p);
  Close(F);

  { Poiščimo pojavitve p-ja v s-ju. }
  StPojavitev := 0;
  for i := 0 to n − pd do begin
    j := 1;
    while j <= pd do if s↑[i + j − 1] = p[j] then j := j + 1 else break;
    if j > pd then StPojavitev := StPojavitev + 1;
  end; {for i}
  FreeMem(s, n);

  { Izpišimo rezultat. }
  Assign(F, 'nizi.out'); Rewrite(F); WriteLn(F, StPojavitev); Close(F);
end. {Nizi}
```

Zapišimo to rešitev še v C-ju:

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

int main()
{
  char *s, p[MaxP + 1], c, cc;
  int z, n, i, j, k, pd, stPojavitev;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("nizi.in", "rt");
  fscanf(f, "%d %d\n", &n, &k);
  fscanf(f, "%c\n", &c);
  s = (char *) malloc(n + 1);
  for (i = 0; i < n; i++) s[i] = c;
  s[n] = 0;
  for (i = 0; i < k; i++) {
    fscanf(f, "%d", &z);
    fgetc(f); /* presledek */
    while ((cc = fgetc(f)) != '\n') s[z++] = cc; }
  fscanf(f, "%s\n", p); pd = strlen(p);
  fclose(f);

  /* Poiščimo pojavitve p-ja v s-ju. */
  stPojavitev = 0;
  for (i = 0; i <= n − pd; i++)
  {
    j = 0;
    while (p[j] && s[i + j] == p[j]) j++;
    if (j == pd) stPojavitev++;
  }
  free(s);
```

```
    /* Izpišimo rezultat. */
    f = fopen("nizi.out", "wt");
    fprintf(f, "%d\n", stPojavitev); fclose(f); return 0;
}
```

Zgornja rešitev uporablja zanko **while**, da pri pri posameznem začetnem položaju i preveri, če se p pojavlja na tem mestu v s. Še lažje pa je, če si pomagamo s funkcijo strstr, ki je del standardne knjižnice jezika C in ki poišče naslednjo pojavitev podniza v danem nizu; če ne najde nobene take pojavitve, vrne 0.

```
    stPojavitev = 0; r = strstr(s, p);
    while (r) { stPojavitev++; r = strstr(r + 1, p); }
```

To, ali je različica s strstr kaj hitrejša od tiste z zanko **while** ali ne, pa je sodeč po naših poskusih precej odvisno od tega, kateri prevajalnik in standardno knjižnico uporabimo (včasih je ena rešitev 30–50 % hitrejša od druge, včasih pa druga prav toliko od prve; pri mnogih testnih primerih sta obe rešitvi tako ali tako približno enako hitri).

5. Obračanje barv

Če barve v neki vrstici (ali stolpcu) obrnemo dvakrat, je učinek enak, kot da jih ne bi obrnili nobenkrat. Torej ni nobene potrebe po tem, da bi kakšno vrstico ali stolpec obrnili več kot enkrat. Ravno tako tudi ni pomembno, v kakšnem vrstnem redu jih obračamo — edini način, da neko polje prizadene več kot eno od teh obračanj, je ta, da leži to polje na preseku neke vrstice in nekega stolpca, ki ju oba obračamo; v tem primeru je vsekakor vseeno, ali obrnemo najprej vrstico ali najprej stolpec, saj je učinek v obeh primerih ta, da se opazovano polje povrne v prvotno barvo.

Ostalo nam je torej le še vprašanje, *katere* stolpce in vrstice obrniti (ne pa to, kolikokrat ali v kakšnem vrstnem redu), da bo nastalo čim manj črnih polj. Naivno bi se lahko naloge lotili takole:

za vsako možno množico stolpcev, $S \subseteq \{1, \ldots, w\}$:
  za vsako možno množico vrstic, $V \subseteq \{1, \ldots, h\}$:
    preštej, koliko črnih polj ostane, če obrnemo stolpce S in vrstice V;
    če je to najboljša rešitev doslej, si jo zapomni;

Težava je v tem, da si lahko množico stolpcev izberemo na 2^w načinov, množico vrstic pa na 2^h načinov, kar pomeni skupaj $O(2^{w+h})$ načinov; pri omejitvah iz naše naloge je to lahko 2^{71}, kar je odločno preveč. (Pač pa bi bila takšna rešitev dovolj hitra za prvih pet izmed desetih testnih primerov, saj naloga zanje zagotavlja, da bosta w in h največ 10.)

Če smo se že odločili za neko konkretno skupino stolpcev, ki jih bomo obrnili (množica S), in nam ostane le še obračanje vrstic (izbira množice V), so vrstice zdaj neodvisne druga od druge — vsaka vrstica obrača sama taka polja, ki jih ne obrača nobena druga. Torej lahko vsako vrstico izbiramo neodvisno od ostalih. Poglejmo, kakšna slika nam nastane po obračanju stolpcev iz množice S, in za vsako vrstico preštejmo črna polja v njej; če je belih več kot črnih, se splača to vrstico obrniti, ker se bodo bela polja v njej spremenila v črna (in obratno) in bo torej po novem črnih več kot prej. Če pa je imela že prej vsaj toliko črnih polj kot belih, se je ne splača obračati.

za vsako možno množico stolpcev, $S \subseteq \{1, \ldots, w\}$:
  naj bo V prazna množica;
  naj bo $s := 0$;
  za vsako vrstico y od 1 do h:
    naj bo n število belih polj v vrstici y (po tistem, ko
      smo obrnili barve v vseh stolpcih iz S);
    če je $n > w - n$, potem dodaj y v V;
    povečaj s za $\min\{n, w - n\}$;
  če je (S, V) z n črnimi polji najboljša rešitev doslej, si jo zapomni;

Da bomo vse to implementirali čim bolj učinkovito, je koristno vsako vrstico naše tabele predstaviti kar z nekim w-bitnim celim številom (torej številom od 0 do $2^w - 1$). Tudi množico S lahko na enak način predstavimo z nekim w-bitnim celim številom; stanje vrstice po obračanju polj v stolpcih iz S lahko potem dobimo preprosto z operacijo xor. Za štetje črnih polj je koristno, če si na začetku v neki pomožni tabeli izračunamo število prižganih bitov v vseh možnih w-bitnih številih. Tako imamo v vsaki iteraciji notranje zanke le $O(1)$ dela in časovna zahtevnost celotnega postopka je le še $O(2^w \cdot h)$. Ker so si pri tej nalogi stolpci in vrstice čisto enakovredni, je pametno na začetku pogledati, če je $w > h$, in v tem primeru tabelo zasukati okrog diagonale; tako iz vrstic nastanejo stolpci in obratno, mi pa v eksponent dobimo krajšo od obeh stranic tabele. Naloga nam zagotavlja, da je krajša stranica dolga največ 21 celic, tako da je s to dodatno izboljšavo časovna zahtevnost našega postopka že čisto sprejemljiva.

```
program ObracanjeBarv;
const MaxMinStranica = 21; MaxStranica = 50;
type TabelaT = packed array [0..(1 shl MaxMinStranica) − 1] of integer;
var w, h, Stolpci, x, y, NajEnic, StEnic, NajOp, StOp, k: integer;
    a: array [0..MaxStranica − 1] of integer;
    StBitov: ↑TabelaT;
    F: text; Obrni: boolean; c: char;
begin
  { Preberimo vhodne podatke. }
  Assign(F, 'toggle.in'); Reset(F); ReadLn(F, w, h);
  Obrni := (h < w); { da bo prišla v eksponent krajša stranica }
  y := 0; while (y < h) or (y < w) do begin a[y] := 0; y := y + 1 end;
  for y := 0 to h − 1 do begin
    for x := 0 to w − 1 do begin
      Read(F, c);
      if c = '#' then
        if Obrni then a[x] := a[x] or (1 shl y)
        else a[y] := a[y] or (1 shl x);
    end; {for x}
    ReadLn(F);
  end; {for y}
  Close(F);
  if Obrni then begin y := w; w := h; h := y end;
  { Pripravimo si tabelo s številom prižganih bitov za vsako w-bitno celo število. }
  GetMem(StBitov, SizeOf(TabelaT[0]) shl w);
  StBitov↑[0] := 0;
  for y := 0 to w − 1 do
    for x := 0 to (1 shl y) − 1 do
      StBitov↑[x or (1 shl y)] := StBitov↑[x] + 1;;
```

```
  { Preizkusimo vse kombinacije stolpcev, ki bi jih lahko obrnili. }
  NajEnic := w * h + 1; NajOp := 0;
  for Stolpci := 0 to (1 shl w) − 1 do begin
    StEnic := 0; StOp := StBitov↑[Stolpci];
    for y := 0 to h − 1 do begin
      k := StBitov↑[a[y] xor Stolpci];
      if k <= w − k then StEnic := StEnic + k
      else begin StOp := StOp + 1; StEnic := StEnic + w − k end;
    end; {for y}
    if StEnic < NajEnic then begin NajEnic := StEnic; NajOp := StOp end
    else if (StEnic = NajEnic) and (StOp < NajOp) then NajOp := StOp;
  end; {for Stolpci}
  FreeMem(StBitov, SizeOf(TabelaT[0]) shl w);
  { Izpišimo rezultat. }
  Assign(F, 'toggle.out'); Rewrite(F);
  WriteLn(F, NajEnic, ' ', NajOp); Close(F);
end. {ObracanjeBarv}
```

Še rešitev v C-ju:

```
#include <stdio.h>
#include <stdbool.h>
#define MaxStranica 50

int main()
{
  int w, h, stolpci, x, y, najEnic, stEnic, najOp, stOp, k;
  int a[MaxStranica], *stBitov; bool obrni; char c;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("toggle.in", "rt");
  fscanf(f, "%d %d\n", &w, &h);
  obrni = (h < w); /* da bo prišla v eksponent krajša stranica */
  for (y = 0; y < w || y < h; y++) a[y] = 0;
  for (y = 0; y < h; y++) {
    for (x = 0; x < w; x++) {
      c = (char) fgetc(f);
      if (c == '#')
        if (obrni) a[x] |= 1 << y; else a[y] |= 1 << x; }
    fgetc(f); }
  fclose(f);
  if (obrni) y = w, w = h, h = y;

  /* Pripravimo si tabelo s številom prižganih bitov
     za vsako w-bitno celo število. */
  stBitov = (int *) malloc(sizeof(int) << w);
  stBitov[0] = 0;
  for (y = 0; y < w; y++)
    for (x = 0; x < (1 << y); x++)
      stBitov[x | 1 << y] = stBitov[x] + 1;

  /* Preizkusimo vse kombinacije stolpcev, ki bi jih lahko obrnili. */
  najEnic = w * h + 1; najOp = 0;
  for (stolpci = 0; stolpci < (1 << w); stolpci++)
  {
    stEnic = 0; stOp = stBitov[stolpci];
    for (y = 0; y < h; y++) {
      k = stBitov[a[y] ↑stolpci];
      if (k <= w − k) stEnic += k;
```

```
      else stOp++, stEnic += w − k; }
    if (stEnic < najEnic || stEnic == najEnic && stOp < najOp)
      najEnic = stEnic, najOp = stOp;
  }
  /* Izpišimo rezultat. */
  f = fopen("toggle.out", "wt");
  fprintf(f, "%d %d\n", najEnic, najOp); fclose(f); return 0;
}
```

REŠITVI NALOG ZA OGREVANJE

1. Izpis HTMLja

Vhodno datoteko lahko prebiramo znak za znakom. V neki spremenljivki (v spodnjem programu je to Izpisuj) hranimo podatek o tem, ali se trenutno nahajamo med oznakama `<body>...</body>` in moramo prebrano besedilo torej izpisovati na izhod. Ko naletimo na znak „<“, preberemo oznako, ki sledi (vse do znaka „>“), in preverimo, če je to katera od tistih oznak, ki zahtevajo posebno obdelavo: `<body>`, `<body/>`, `<br>` in `<br/>`. Pri slednjih dveh izpišemo znak za konec vrstice. Ker nobena od omenjenih štirih oznak ni daljša od 7 znakov, lahko pri branju oznake po sedmih znakih kar obupamo in preostanek oznake preskočimo — beremo do znaka „>“, vendar si ne zapomnimo preostanka oznake, saj že zdaj vemo, da to ne bo nobena od tistih oznak, ki nas posebej zanimajo. Tako nam ni treba skrbeti, da bi za spremenljivko Oznaka v kakšnem patološkem primeru porabili neugodno veliko pomnilnika.

Besedilo naloge ne pove natančno, ali naj pri oznakah upoštevamo velikost črk, torej ali nam npr. `<BODY>` pomeni isto kot `<body>` ali ne. Tradicionalno v HTMLju ni pomembno, ali se v oznakah uporabljajo velike ali male črke (ali mešanica obojih), zato naš program pri branju oznak vse črke spremeni v velike.

```
program IzpisHTMLja;
var c: char; Oznaka: string; Izpisuj: boolean;
begin
  Izpisuj := false;
  while not Eof do begin
    Read(c);
    if c <> '<' then begin
      if Izpisuj then Write(c);
    end else begin
      Oznaka := c;
      while not Eof do begin
        Read(c);
        if Length(Oznaka) < 7 then Oznaka := Oznaka + UpCase(c);
        if c = '>' then break;
      end; {while}
      if Oznaka = '<BODY>' then Izpisuj := true
      else if Oznaka = '</BODY>' then Izpisuj := false
      else if (Oznaka = '<BR>') or (Oznaka = '<BR/>') then WriteLn;
    end;
  end; {while}
end. {IzpisHTMLja}
```

Še rešitev v C-ju:

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

int main()
{
  int c, d; bool izpisuj = false;
  char oznaka[8];
  while ((c = getc(stdin)) != EOF)
```

```
    if (c != '<') { if (izpisuj) putchar(c); }
    else
    {
      d = 0; oznaka[d++] = c;
      while ((c = getc(stdin)) != EOF)
      {
        if (d < 7) oznaka[d++] = toupper(c);
        if (c == '>') break;
      }
      oznaka[d] = 0;
      if (strcmp(oznaka, "<BODY>") == 0) izpisuj = true;
      else if (strcmp(oznaka, "</BODY>") == 0) izpisuj = false;
      else if (strcmp(oznaka, "<BR>") == 0 || strcmp(oznaka, "<BR/>") == 0)
        putchar('\n');
    }
  return 0;
}
```

2. Popravilo ograje

Nalogo lahko rešimo s požrešnim algoritmom. Oglejmo si najbolj levo poškodovano deščico (torej tisto z najmanjšo številko — recimo ji a_1); očitno mora ena od desk, ki jih bomo namestili ob popravilu ograje, pokriti tudi položaj te deščice. Nobene koristi pa ne bi bilo od tega, da bi se kakšna deska začela levo od a_1, saj tam ni nobene poškodovane deščice. Torej postavimo najbolj levo novo desko tako, da se začne pri a_1. Z njo lahko zamenjamo vse poškodovane deščice od a_1 do vključno $a_1 + k - 1$. V nadaljevanju se nam s temi deščicami torej ni treba več ukvarjati in lahko enak razmislek, kot smo ga zdaj opravili za a_1, ponovimo za prvo naslednjo poškodovano deščico (torej prvo tako, ki je desno od položaja a_1+k-1). Ta postopek ponavljamo, dokler ne pokrijemo vseh deščic.

Reševanje nam še olajša dejstvo, da so poškodovane deščice v vhodni datoteki že urejene naraščajoče, torej od leve proti desni. Vse, kar potrebujemo, je, da si v neki spremenljivki (v spodnjem programu je to Konec) zapomnimo, kje se konča najbolj desna od doslej uporabljenih desk, tako da bomo vedeli, kdaj (pri kateri poškodovani deščici) začeti novo desko.

```
program PopraviloOgraje;
var n, k, c, StDesk, Konec, x: integer; F: text;
begin
  Assign(F, 'ograja.in'); Reset(F);
  ReadLn(F, n, k, c);
  StDesk := 0; Konec := 0;
  while c > 0 do begin
    ReadLn(F, x); c := c − 1;
    if x > Konec then begin { začnimo z novo desko }
      StDesk := StDesk + 1;
      Konec := x + k − 1;
    end; {if}
  end; {while}
  Close(F);
  Assign(F, 'ograja.out'); Rewrite(F); WriteLn(F, StDesk); Close(F);
end. {PopraviloOgraje}
```

Še rešitev v C-ju:

```
#include <stdio.h>

int main()
{
   int n, k, c, stDesk, konec, x;
   FILE *f = fopen("ograja.in", "rt");
   fscanf(f, "%d %d %d", &n, &k, &c);
   stDesk = 0; konec = 0;
   while (c > 0) {
      fscanf(f, "%d", &x); c--;
      if (x > konec) /* začnimo z novo desko */
         stDesk++, konec = x + k - 1; }
   fclose(f);
   f = fopen("ograja.out", "wt"); fprintf(f, "%d\n", stDesk); fclose(f);
   return 0;
}
```

Naloge so sestavili: popravilo ograje — Nino Bašić; darila, elektronska ključavnica — Andrej Bauer; obračanje barv — Tomaž Hočevar; kdo je izdal skrivnost? — Boris Horvat; PINi — Aleksandar Jurišić; roboti — Peter Keše; izpis HTMLja — Jurij Kodre; GrbaveBesede — Mark Martinec; rsync — Mojca Miklavec; piskrc špagetov, Berberi — Mitja Trampuš; predavalnice — Miha Vuk; usklajevanje ur — Klemen Žagar; društvo ljubiteljev ničel, cik cak, redki nizi – Janez Brank.

Hvala Tomažu Hočevarju za implementacijo rešitev nalog za 3. skupino in Ninu Bašiću za implementacijo rešitve nizov.

Naloga o Berberih si jemlje nekaj umetniške svobode: Yusuf je umrl že na začetku 12. stoletja, Muhammad pa se je leta 1125 res skregal, vendar z Yusufovim sinom Alijem; glej članke o vseh naštetih v Wikipediji.

1. Palindromni stavki

Preveriti moramo, če se prva črka ujema z zadnjo, druga s predzadnjo in tako naprej, nečrkovne znake pa moramo pri tem preskočiti. Tega se lahko lotimo tako, da se sprehajamo po nizu z dvema števcema; eden se premika od leve proti desni, drugi pa od desne proti levi. Na vsakem koraku se premaknemo z obema števcema mimo morebitnih nečrkovnih znakov do naslednje črke in nato preverimo, če sta črki, na kateri kažeta oba števca, enaki. Ko se števca srečata, vemo, da smo pregledali že cel niz in če doslej nismo opazili neujemanja, je v našem nizu palindromni stavek, po kakršnem sprašuje naloga.

```
function JePalindromniStavek(S: string): boolean;
var i, j: integer;
begin
  i := 1; j := Length(S); JePalindromniStavek := true;
  while i < j do begin
    while i <= j do if (S[i] < 'A') or (S[i] > 'Z') then i := i + 1 else break;
    while i <= j do if (S[j] < 'A') or (S[j] > 'Z') then j := j – 1 else break;
    if i >= j then break;
    if S[i] <> S[j] then begin JePalindromniStavek := false; break end;
    i := i + 1; j := j – 1;
  end; {while}
end; {JePalindromniStavek}
```

Še rešitev v C-ju:

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

bool JePalindromniStavek(char *s)
{
  int i = 0, j = strlen(s) – 1;
  while (i < j)
  {
    while (i <= j && (s[i] < 'A' || s[i] > 'Z')) i++;
    while (i <= j && (s[j] < 'A' || s[j] > 'Z')) j––;
    if (i >= j) break;
    if (s[i] != s[j]) return false;
    i++; j––;
  }
  return true;
}
```

Še preprostejša rešitev pa je, da črkovne znake iz niza S skopiramo v nek nov niz in potem zanj preverimo, če je palindrom. Tako se nam ni treba toliko ukvarjati s tem, kako preskočiti nečrkovne znake pri preverjanju palindromskosti. Slabost tega pristopa pa je, da moramo ustvariti še eno kopijo niza S, torej porabimo nekaj več pomnilnika kot prej, sploh če je S dolg. Primer take rešitve v pythonu:

```
def JePalindromniStavek(s):
  s = "".join(c for c in s if c.isalpha())
  return s == s[::–1]
```

2. Živi in mrtvi

Recimo, da je bilo na začetku leta i živih z_i in mrtvih m_i ljudi in da se je v tistem letu rodilo r_i ljudi, s_i pa jih je umrlo. (Zagotovo velja, da je $s_i \leq z_i + r_i$.) Najmanjše možno število živih ljudi v tem letu torej dobimo, če nastopijo vse smrti pred vsemi rojstvi; v tem primeru imamo nekaj časa le $z_i - s_i$ živih ljudi. Največje možno število mrtvih pa dobimo v vsakem primeru na koncu leta, po vseh s_i smrtih tega leta, ko se skupno število mrtvih poveča na $m_i + s_i$. Scenarij (1) iz besedila naloge (torej možnost, da je število živih zagotovo vedno večje od števila mrtvih) je torej mogoč v primeru, ko je $z_i - s_i > m_i + s_i$.[11] O pravilnosti tega pogoja se lahko prepričamo tako, da razmislimo o primerih, ko ni izpolnjen, torej ko velja $z_i - s_i \leq m_i + s_i$. Izkaže se, da lahko za vsak tak primer sestavimo zaporedje rojstev in smrti, pri katerem število živih vsaj nekaj časa ni večje od števila mrtvih; torej je prav, da naš pogoj v takem primeru možnost (1) zavrne.

Največje možno število živih ljudi dobimo v primeru, če se vsa rojstva zgodijo pred vsemi smrtmi; takrat imamo nekaj časa kar $z_i + r_i$ živih ljudi, mrtvih pa še vedno le m_i (kar je seveda tudi najmanjše možno število mrtvih pri teh podatkih). Scenarij (2) iz besedila naloge (torej možnost, da je število mrtvih zagotovo vedno večje od števila živih) je torej mogoč v primeru, ko je $m_i > z_i + r_i$.

Če ne drži nič od tega dvojega, torej ne moremo zanesljivo sklepati niti na (1) niti na (2), kar pomeni, da imamo možnost (3). V takem primeru je to, ali bo število živih ves čas večje ali ves čas manjše od števila mrtvih, ali pa celo nekaj časa večje, nekaj časa manjše in nekaj časa enako, odvisno od vrstnega reda posameznih rojstev in smrti.

V vsakem primeru pa lahko za naslednje leto izračunamo $z_{i+1} = z_i + r_i - s_i$ ter $m_{i+1} = m_i + s_i$.

```
program ZiviInMrtvi;
var StZivih, StMrtvih, Leto: integer;
begin
  StZivih := ZacetnoSteviloPrebivalcev; StMrtvih := 0;
  for Leto := 1 to 100 do begin
    if StZivih − SteviloSmrti(Leto) > StMrtvih + SteviloSmrti(Leto) then
      WriteLn('Leta ', Leto, ' je bilo vedno več živih kot mrtvih.')
    else if StMrtvih > StZivih + SteviloRojstev(Leto) then
      WriteLn('Leta ', Leto, ' je bilo vedno več mrtvih kot živih.')
    else
      WriteLn('Leto ', Leto, ' ne pripada nobeni od prvih dveh skupin.');
    StZivih := StZivih + SteviloRojstev(Leto) − SteviloSmrti(Leto);
```

[11] Spodobi se, da posebej razmislimo tudi o primeru, ko je $z_i - s_i$ manjše od 0. V tem primeru (ki se teoretično lahko zgodi, če se veliko ljudi rodi in umre v letu i) je namreč nemogoče, da bi vse smrti nastopile pred vsemi rojstvi, saj bi število prebivalcev v tem primeru padlo pod 0. Naj bo $D = s_i - z_i$; opazimo, da iz $s_i \leq z_i + r_i$ sledi tudi $D \leq r_i$. Mogoč je torej scenarij, po katerem se najprej zgodi D rojstev in nato D smrti; ostane nam spet z_i živih ljudi, v tem letu pa se mora zgoditi še $r_i - D$ rojstev in $s_i - D$ (kar je z_i) smrti. Če je $r_i - D = 0$, bo število prebivalstva padlo na 0, če pa je $r_i - D > 0$, je verjetno najnižje število prebivalstva, ki ga še lahko dosežemo, 1 (težko bi prišlo do še kakšnega rojstva, če bi pred tem celotno prebivalstvo pomrlo). Kakorkoli že, ker iz $z_i - s_i$ sledi tudi, da je $D > 0$, vemo, da bo v trenutku, ko število prebivalcev pade na 0 oz. 1, že mrtvih vsaj $D > 0$ ljudi, torej vsekakor ne moremo z gotovostjo trditi, da bo število živih v vsakem trenutku večje od števila mrtvih. Možnost (1) iz besedila naloge torej v tem primeru odpade. Ker tudi pogoj „$z_i - s_i > m_i + s_i$“ v tem primeru ne bi bil izpolnjen (leva stran je negativna, desna pa pozitivna), bi torej naša rešitev tudi v tem primeru delovala pravilno.

```
      StMrtvih := StMrtvih + SteviloSmrti(Leto);
   end; {for Leto}
end. {ZiviInMrtvi}
```

Še rešitev v C-ju:

```
#include <stdio.h>

int main()
{
   int stZivih, stMrtvih, leto;
   stZivih = ZacetnoSteviloPrebivalcev();
   stMrtvih = 0;
   for (leto = 1; leto <= 100; leto++)
   {
      if (stZivih − SteviloSmrti(leto) > stMrtvih + SteviloSmrti(leto))
         printf("Leta %d je bilo vedno več živih kot mrtvih.\n", leto);
      else if (stMrtvih > stZivih + SteviloRojstev(leto))
         printf("Leta %d je bilo vedno več mrtvih kot živih.\n", leto);
      else
         printf("Leto %d ne pripada nobeni od prvih dveh skupin.\n");
      stZivih += SteviloRojstev(leto) − SteviloSmrti(leto);
      stMrtvih += SteviloSmrti(leto);
   }
   return 0;
}
```

3. Lomljenje besedila

Vhodno besedilo berimo znak za znakom; prebrane presledke dodajajmo na konec niza Presledki, druge znake pa na konec niza Beseda. Ko preberemo presledek, najprej še preverimo, če je niz Beseda trenutno neprazen; če je, to pomeni, da smo pravkar prebrali prvi presledek za tisto besedo, torej smo na koncu besede (cela beseda je shranjena v spremenljivki Beseda; spremenljivka Presledki pa hrani vse presledke med to in prejšnjo besedo) in je zdaj primeren trenutek, da jo izpišemo. V spodnjem programu za izpis skrbi podprogram KonecBesede, ki v globalni spremenljivki SirDoslej hrani skupno širino znakov, ki jih je doslej že izpisal v trenutno vrstico. Če opazimo, da za presledke in besedo skupaj v trenutni vrstici ni dovolj prostora, gremo v novo vrstico (in vanjo izpišemo le besedo). V vsakem primeru pa po izpisu popravimo vrednost SirDoslej in postavimo spremenljivki Beseda in Presledki spet na prazen niz. Na koncu vhodnega besedila še enkrat pokličemo KonecBesede, da izpiše še zadnjo besedo (ki je še nismo izpisali, razen če je bil za njo v vhodnem besedilu še kak presledek).

```
program Lomljenje;
var Presledki, Beseda: string; SirDoslej: integer;

   procedure KonecBesede;
   var SP, SB, i: integer;
   begin
      if Beseda = '' then exit;
      SP := 0; for i := 1 to Length(Presledki) do SP := SP + SirinaZnaka(Presledki[i]);
      SB := 0; for i := 1 to Length(Beseda) do SB := SB + SirinaZnaka(Beseda[i]);
      if SirDoslej + SP + SB > SirinaZaslona
      then begin WriteLn; SirDoslej := 0 end
```

```
      else begin Write(Presledki); SirDoslej := SirDoslej + SP end;
      Write(Beseda); SirDoslej := SirDoslej + SB;
      Beseda := ''; Presledki := '';
   end; {KonecBesede}

var c: char;
begin {Lomljenje}
   Presledki := ''; Beseda := ''; SirDoslej := 0;
   while not Eof do begin
      Read(c);
      if c <> ' ' then Beseda := Beseda + c
      else begin KonecBesede; Presledki := Presledki + c end;
   end; {while}
   KonecBesede;
   WriteLn;
end. {Lomljenje}
```

Zapišimo še rešitev v C-ju. Pri alokaciji pomnilnika za niza beseda in presledki smo predpostavili, da niti besede niti skupine presledkov ne bodo daljše od SirinaZaslona znakov. (V praksi bi bilo seveda koristno, če bi naša rešitev delovala pravilno tudi v takih primerih in bi znala npr. razbiti zelo dolgo besedo na več vrstic.) Kazalca kb in kp kažeta na konec besede oz. niza presledkov, torej tja, kamor je treba dodati naslednji znak.

```
#include <stdio.h>

char *presledki, *beseda, *kp, *kb;
int sirDoslej;

void KonecBesede()
{
   int SP = 0, SB = 0; char *p;
   if (kb == beseda) return;
   for (p = presledki; p < kp; p++) SP += SirinaZnaka(*p++);
   for (p = beseda; p < kb; p++) SB += SirinaZnaka(*p++);
   if (sirDoslej + SP + SB > SirinaZaslona())
      { printf("\n"); sirDoslej = 0; }
   else { *kp = 0; printf("%s", presledki); sirDoslej += SP; }
   *kb = 0; printf("%s", beseda); sirDoslej += SB;
   kp = presledki; kb = beseda;
}

int main()
{
   int c;
   presledki = (char *) malloc(SirinaZaslona() + 1);
   beseda = (char *) malloc(SirinaZaslona() + 1);
   kp = presledki; kb = beseda; sirDoslej = 0;
   while ((c = fgetc(stdin)) != EOF)
      if (c != ' ') *kb++ = c;
      else { KonecBesede(); *kp++ = c; }
   KonecBesede();
   free(presledki); free(beseda);
   printf("\n"); return 0;
}
```

4. Taborniki (I)

Naloga zahteva, naj izpišemo seznam spiskov, ki so omenjeni v arhivu, ne pa tudi na seznamu delujočih spiskov. Torej moramo za vsak spisek iz arhiva ugotoviti, ali se nahaja na seznamu delujočih spiskov. Ena možnost je, da ta seznam preberemo v pomnilnik in se nato vsakič, ko nas za nek spisek zanima, ali je na njem ali ne, sprehodimo po celem seznamu in primerjamo imena spiskov na njem z imenom spiska, ki ga iščemo. Vendar pa naloga pravi, da sta oba seznama (tako arhiv kot seznam delujočih spiskov) urejena naraščajoče po abecedi; s tem si lahko pomagamo do hitrejše in učinkovitejše rešitve naloge. Ko iščemo v seznamu delujočih spiskov nek spisek s, se lahko pri iskanju ustavimo, čim naletimo na kakšen tak spisek, ki je po abecedi večji ali enak s — ker je seznam urejen, takrat namreč vemo, da bodo tudi vsi nadaljnji spiski po abecedi večji od s, torej s-ja med njimi ne bomo našli. In ko potem preberemo iz arhivskega seznama naslednji spisek, recimo s', upoštevajmo, da je tudi arhivski seznam urejen po abecedi, torej je s' po abecedi večji od s; v seznamu delujočih spiskov so vsi spiski pred trenutnim manjši (po abecedi) od s (ker smo se ustavili, čim smo naleteli na takega, ki je večji ali enak s), torej so tudi manjši od s' in se z njimi ni treba ukvarjati, ko iščemo spisek s' v seznamu delujočih. Z branjem tega seznama lahko torej nadaljujemo kar tam, kjer smo končali pri iskanju spiska s.

```
program Taborniki;
var Vsi, Delujoci: text; ImeV, ImeD: string;
begin
  Assign(Vsi, 'seznam-arhiv.txt'); Reset(Vsi);
  Assign(Delujoci, 'seznam-delujoci.txt'); Reset(Delujoci);
  ImeD := '';
  while not Eof(Vsi) do begin
    ReadLn(Vsi, ImeV);
    while (ImeD < ImeV) and not Eof(Delujoci) do ReadLn(Delujoci, ImeD);
    if ImeD <> ImeV then WriteLn(ImeV);
  end; {while}
  Close(Vsi); Close(Delujoci);
end. {Taborniki}
```

Oglejmo si še rešitev v C-ju:

```
#include <stdio.h>

#define MaxDolz 100

int main()
{
  FILE *vsi = fopen("seznam-arhiv.txt", "rt");
  FILE *delujoci = fopen("seznam-delujoci.txt", "rt");
  char imeV[MaxDolz + 2], imeD[MaxDolz + 2];
  imeD[0] = 0;
  while (! feof(vsi))
  {
    if (! fgets(imeV, MaxDolz + 2, vsi)) break;
    while (strcmp(imeD, imeV) < 0 && ! feof(delujoci))
      fgets(imeD, MaxDolz + 2, delujoci);
    if (strcmp(imeD, imeV) != 0) printf("%s", imeV);
  }
```

```
  fclose(vsi); fclose(delujoci); return 0;
}
```

5. Nerimska števila

Pri tej nalogi nas bodo zanimali palindromni nizi z vrednostjo n; mislimo si nek tak niz s in recimo, da je dolg m znakov, torej $s = s_1 s_2 \ldots s_m$.

Če je m sod, npr. $m = 2k$, sta niza $s_1 \ldots s_k$ in $s_{k+1} \ldots s_n$ zrcalni kopiji drug drugega, torej imata oba enako vrednost; ker ima celoten niz s vrednost n, mora imeti vsaka od obeh polovic vrednost $n/2$ (to je torej mogoče le, če je n sod).

Če pa je m lih, npr. $m = 2k+1$, sta zrcalni kopiji drug drugega niza $s_1 \ldots s_k$ in $s_{k+2} \ldots s_m$, vsak od njiju pa mora torej imeti vrednost $(n - v(s_{k+1}))/2$, če je $v(s_{k+1})$ vrednost črke na sredi niza s. Ta varianta je torej mogoča le, če je $n - v(s_{k+1})$ sodo število.

Ker hočemo poiskati najkrajši palindromni niz z vrednostjo n, je pametno preizkusiti obe možnosti. Če je n sod, poiščimo najkrajši niz z vrednostjo $n/2$ in mu pritaknimo na koncu še njegovo zrcalno kopijo; to je zdaj nek palindrom z vrednostjo n. Podobno tudi za vsako črko c (od `A` do `Z`) poglejmo, če je $n - v(c)$ sod, in če je, poiščimo najkrajši niz (recimo s') z vrednostjo $(n - v(c))/2$; potem staknimo s', črko c in še zrcalno kopijo niza s', pa imamo spet nek palindrom z vrednostjo n. Med tako dobljenimi palindromi vrnimo najkrajšega.

Ostalo nam je le še vprašanje, kako poiskati najkrajši niz z neko zahtevano vrednostjo (recimo x). Naj bo t število črk v abecedi, v_i pa vrednost i-te črke; naloga pravi, da je (za vsak i) v_{i+1} večkratnik števila v_i, torej je razmerje v_{i+1}/v_i celo število (recimo mu d_i). Da bo razmislek preprostejši, se delajmo, da obstaja še neka $(t+1)$-va črka abecede, njena vrednost pa je večkratnik v_t in hkrati večja od x.

Naloga pravi, da je vrednost niza definirana kot vsota vrednosti posameznih črk v njem; torej vrstni red črk nič ne vpliva na vrednost niza (ker je seštevanje komutativna operacija), pomembno je le, kolikokrat se posamezna črka pojavlja v njem. Če se črka i pojavlja vsaj d_i-krat, lahko teh d_i pojavitev črke i zamenjamo z eno pojavitvijo črke $i+1$, pa bo niz ohranil svojo vrednost, obenem pa se bo tudi skrajšal (oz. ostal enako dolg, če je $d_i = 1$); ker hočemo čim krajši niz, se lahko torej omejimo na nize, ki imajo (za vsak i) največ $d_i - 1$ pojavitev črke i. Z indukcijo po i se ni težko prepričati, da je skupna vrednost vseh pojavitev vseh črk od 1 do $i-1$ v takem nizu manjša od v_i.

Recimo, da je $v_i \leq x < v_{i+1}$. V najkrajšem nizu z vrednostjo x potem gotovo ne nastopajo črke od $i+1$ naprej, ker bi imel potem niz vrednost, večjo ali enako v_{i+1}. Po drugi strani pa v tem nizu gotovo nastopa vsaj ena črka i, kajti če bi bile vse črke manjše od i, bi imel niz (kot smo videli v prejšnjem odstavku) vrednost, manjšo od v_i (in torej tudi od x). Niz lahko torej začnemo s črko i, vrednost x v mislih zmanjšamo za v_i in nadaljujemo po enakem postopku.

```
procedure NajkrajsiPalindrom(n: integer);
var Najkrajsi: string;

  procedure Obdelaj(Sredina: string; Ostanek: integer);
  var c: char; i: integer; Desno, Skupaj: string;
  begin
```

```
      if Ostanek < 0 then exit;
      if Odd(Ostanek) then exit;
      Ostanek := Ostanek div 2; Desno := ''; Skupaj := '';
      for c := 'Z' downto 'A' do
         while Ostanek >= Vrednost(c) do
            begin Desno := Desno + c; Ostanek := Ostanek − Vrednost(c) end;
      for i := Length(Desno) downto 1 do Skupaj := Skupaj + Desno[i];
      Skupaj := Skupaj + Sredina + Desno;
      if (Najkrajsi = '') or (Length(Skupaj) < Length(Najkrajsi)) then
         Najkrajsi := Skupaj;
   end; {Obdelaj}

var c: char;
begin {NajkrajsiPalindrom}
   Najkrajsi := ''; Obdelaj('', n);
   for c := 'A' to 'Z' do Obdelaj(c, n − Vrednost(c));
   WriteLn(Najkrajsi);
end; {NajkrajsiPalindrom}
```

Oglejmo si še rešitev v C-ju. V tem jeziku žal nimamo pri roki kakšnega prikladnega tipa za poljubno dolge nize (kot je string v pascalu, C++, javi ipd.). Zato imamo spodaj podprogram SteviloVNiz, ki zna pretvoriti dano število v niz in vrniti dolžino tega niza; če pa mu podamo kazalec na tabelo znakov, bo niz tudi shranil vanjo. Ta podprogram uporabljamo, da najprej določimo, kako dolg bo najkrajši primerni palindrom, šele nato pa alociramo blok pomnilnika zanj in nato ta niz tudi zares sestavimo.

```
/* Pretvori niz n v število in ga shrani v s, vrne pa njegovo dolžino.
   Če je s == 0, samo vrne dolžino. */
int SteviloVNiz(int n, char *s)
{
   char c; int dolzina = 0;
   for (c = 'Z'; c >= 'A'; c−−)
      while (n >= Vrednost(c)) {
         n −= Vrednost(c); dolzina++; if (s) *s++ = c; }
   return dolzina;
}

void NajkrajsiPalindrom(int n)
{
   /* Poglejmo, katera črka mora biti na sredini palindroma, da bo ta čim krajši. */
   int najDolz = n + 1, dolzina, m; char najSredina = ' ', sredina;
   if (n % 2 == 0) najDolz = SteviloVNiz(n / 2, 0);
   for (sredina = 'A'; sredina <= 'Z'; sredina++)
   {
      m = n − Vrednost(sredina); if (m % 2 != 0) continue;
      dolzina = SteviloVNiz(m / 2, 0);
      if (dolzina < najDolz) najDolz = dolzina, najSredina = sredina;
   }
   /* Pripravimo zdaj niz, ki ga iščemo. */
   dolzina = 2 * najDolz + (najSredina == ' ' ? 0 : 1);
   char *s = (char *) malloc(dolzina + 1);
   m = (n − (najSredina == ' ' ? 0 : Vrednost(najSredina))) / 2;
   SteviloVNiz(m, s); /* leva polovica */
   if (najSredina != ' ') s[najDolz] = najSredina; /* srednji znak */
   for (m = 0; m < najDolz; m++) /* skopirajmo levo polovico v desno */
```

```
    s[najDolz + (najSredina == ' ' ? 0 : 1) + m] = s[najDolz − 1 − m];
  s[dolzina] = 0;
  printf("%s\n", s);
  free(s);
}
```

6. Taborniki (II)

Dnevniško datoteko berimo vrstico za vrstico; zapise, ki se ne nanašajo na tisti spisek, ki nas zanima, lahko preskočimo. Seznam trenutnih članov spiska hranimo v pomnilniku in pri vsakem zapisu člana bodisi dodamo bodisi pobrišemo iz tega seznama (odvisno od tega, ali govori trenutni zapis o prijavi na spisek ali o izpisu iz njega).

Tovrstne naloge so še posebej primerne za reševanje s skriptnimi jeziki, kot so perl, python in PHP. Oglejmo si primer rešitve v pythonu:

```
imeSpiska = raw_input("Vnesi ime spiska: ")
clani = set()
for vrstica in open("dnevnik.txt"):
  [datum, spisek, akcija, naslov] = vrstica.strip().split('|')
  if spisek != imeSpiska: continue
  if akcija == "subscribe": clani.add(naslov)
  elif naslov in clani: clani.remove(naslov)
for naslov in clani: print naslov
```

7. Nezanesljivi golobi

Ko A pošlje B-ju sporočilo, naj nanj poleg vsebine zapiše še zaporedno številko sporočila, čas trenutnega pošiljanja ali katerokoli drugo vrednost, ki je pri vsakem naslednjem sporočilu večja kot pri prejšnjem.

Ko B sporočilo prejme, naj pošlje A-ju povratnico — torej sporočilo, v katerem A-ju javi, da je sprejel A-jevo sporočilo z določeno številko. Če A povratnice v določenem času (npr. enem dnevu) ne prejme, pošlje isto sporočilo (z isto številko!) še enkrat.

Lahko se zgodi, da je B sporočilo sicer prejel, a potrditev ni prispela do A. V tem primeru se bo rado zgodilo, da bo A sporočilo ponovno poslal in ga bo tudi B prejel še enkrat. Na podlagi številke sporočila lahko B ugotovi, da je sporočilo že sprejel, in ga v tem primeru ignorira. Vseeno pa mora poslati povratnico zanj, saj bi sicer A isto sporočilo pošiljal v nedogled. Do tega, da B prejme več izvodov istega sporočila, lahko pride tudi zato, ker se je bodisi prvi golob z A-jevim sporočilom ali pa golob z B-jevo prvo povratnico predolgo zadržal na poti, pa je A zato mislil, da se je sporočilo izgubilo in da ga mora poslati še enkrat.

Mimogrede, s problemi, kakršnega opisuje ta naloga, se ne srečujemo le pri golobih, ampak tudi pri računalniških omrežjih. Na internetu za reševanje težav z izgubljenimi in podvojenimi sporočili skrbi protokol TCP (ki pa skrbi tudi še za marsikaj drugega, česar naša naloga ne pokriva, npr. za zagotavljanje tega, da dobi prejemnik sporočila v enakem vrstnem redu, v kakršnem jih je pošiljatelj poslal).[12]

[12] Gl. npr. RFCje 793, 1149 in 2549.

8. Začetnice

Vhodno besedilo lahko beremo znak za znakom; če prejšnji znak ni presledek, trenutni pa je, smo na koncu besede. Zadnjih n prebranih besed hranimo v neki tabeli $b[1..n]$, koristno pa je hraniti še njihove začetne črke v nekem nizu $t[1..n]$. Na koncu vsake besede primerjamo t in s; če se niza ujemata, vemo, da smo našli primerno zaporedje n besed, in jih lahko izpišemo.

```
for i := 1 to n: t[i] := ' '; b[i] := prazen niz;
c := ' '; w := prazen niz;
repeat
  p := c;
  if EOF then c := ' ' else c := naslednji znak vhodnega besedila;
  if c ≠ ' ': dodaj c na konec w; continue
  if p = ' ': continue
  (* Smo na koncu besede w. *)
  for i := 2 to n: t[i − 1] := t[i]; b[i − 1] := b[i];
  t[n] := w[1]; b[n] := w;
  if t = s: izpiši zaporedje besed b;
until EOF;
```

Tu imamo zdaj pri vsaki besedi $O(n)$ dela s premikanjem podatkov po tabelah t in b. Majhno izboljšavo lahko dosežemo, če tidve tabeli uporabljamo kot krožni tabeli. Spremenljivka j naj nam pove naš trenutni položaj v teh dveh tabelah; na začetku jo inicializirajmo na 1.

```
  (* Smo na koncu besede w. *)
  t[j] := w[1]; b[j] := w;
  j := j + 1; if j > n then j := 1;
  i := 1; k := j;
  while i ≤ n:
    if s[i] ≠ t[k] then break;
    i := i + 1; k := k + 1; if k > n then k := 1;
  if i > n: izpiši zaporedje besed b;
```

Zdaj se sicer še vedno lahko zgodi, da bomo imeli na koncu besede $O(n)$ dela, vendar le zaradi primerjanja nizov s in t; v večini primerov pa (če gre npr. za besedilo v naravnem jeziku) bo ta zanka že kmalu (pri majhnem i) opazila neujemanje. V vsakem primeru pa nam dodajanje nove besede v b in njene začetne črke v t zdaj vzame le $O(1)$ časa. Pri m besedah je časovna zahtevnost v najslabšem primeru $O(nm)$, če ne štejemo časa branja besedila in morebitnega izpisovanja rezultatov.

Postopek bi lahko še izboljšali takole: začetne črke vseh besed besedila lahko v mislih staknemo v nek dolg niz z; zdaj si želimo ugotoviti, kje vse se s pojavlja kot podniz niza z. Pri tem lahko uporabimo kakšnega od boljših postopkov za iskanje podniza v nizu, ki ne bo imel časovne zahtevnosti $O(nm)$, ampak kaj manjšega; primer sta Knuth-Morris-Prattov in Boyer-Moorov algoritem. Pri tem tudi ni nujno, da si niz z predstavimo eksplicitno (v neki dolgi tabeli m elementov), ampak lahko nanj gledamo skozi tabelo t, enako kot v naši zgornji rešitvi.

9. Ločevanje slik

Najbolj očitna rešitev je verjetno ta, da z nekaj gnezdenimi zankami pregledamo vse x_1, y_1, x_2 in y_2 ter pri vsaki kombinaciji teh koordinat pregledamo vse pare slik in preverjamo, če se vsak par slik razlikuje v barvi vsaj enega od obeh pikslov:

```
function JeParOk(x1, y1, x2, y2: integer): boolean;
var i1, i2: integer;
begin
  JeParOk := true;
  for i1 := 1 to n − 1 do for i2 := i1 + 1 to n do
    if (Slike[i1, y1, x1] = Slike[i2, y1, x1])
    and (Slike[i1, y2, x2] = Slike[i2, y2, x2])
    then begin JeParOk := false; exit end;
end; {JeParOk}

procedure Najdi;
var x1, y1, x2, y2: integer;
begin
  for y1 := 0 to Visina − 1 do for x1 := 0 to Sirina − 1 do
  for y2 := 0 to Visina − 1 do for x2 := 0 to Sirina − 1 do
    if JeParOk(x1, y1, x2, y2) then
      begin WriteLn(x1, ' ', y1, ' ', x2, ' ', y2); exit end;
  WriteLn('Ni primernega para.');
end; {Najdi}
```

Še v C-ju:

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

#define n ...
#define Visina ...
#define Sirina ...

unsigned char slike[n][Visina][Sirina];

bool JeParOk(int x1, int y1, int x2, int y2)
{
  int i1, i2;
  for (i1 = 0; i1 < n − 1; i1++) for (i2 = i1 + 1; i2 < n; i2++)
    if (slike[i1][y1][x1] == slike[i2][y1][x1] &&
       slike[i1][y2][x2] == slike[i2][y2][x2]) return false;
  return true;
}

void Najdi()
{
  int x1, y1, x2, y2;
  for (y1 = 0; y1 < Visina; y1++) for (x1 = 0; x1 < Sirina; x1++)
  for (y2 = 0; y2 < Visina; y2++) for (x2 = 0; x2 < Sirina; x2++)
    if (JeParOk2(x1, y1, x2, y2)) {
      printf("%d %d %d %d\n", x1, y1, x2, y2); return; }
  printf("Ni primernega para.\n");
}
```

Neugodna pri tej rešitvi je njena velika časovna zahtevnost: $O(w^2h^2n^2)$, če imamo n slik velikosti $w \times h$.

Boljša rešitev se lahko znebi ene od zank v JeParOk. Uporabimo tabelo A, v kateri nam vrednost $A[b_1, b_2]$ pove, ali ima kakšna slika na koordnatah (x_1, y_1) piksel barve b_1, na koordinatah (x_2, y_2) pa piksel koordinate b_2. Tako lahko pri vsaki sliki vidimo, če je imela že kakšna od dosedanjih slik na teh dveh mestih piksla iste barve; če se to res zgodi, vemo, da s trenutnim naborom koordinat ne bomo mogli ločiti vseh slik.

```
{ ob inicializaciji postavimo vse na false }
var A: array [0..255, 0..255] of boolean;

function JeParOk2(x1, y1, x2, y2: integer): boolean;
var i, b1, b2: integer;
begin
  JeParOk2 := true;
  for i := 1 to n do begin
    b1 := Slike[i, y1, x1]; b2 := Slike[i, y2, x2];
    if A[b1, b2] then begin JeParOk2 := false; break end;
    A[b1, b2] := true;
  end; {for i}
  for i := 1 to n do begin
    b1 := Slike[i, y1, x1]; b2 := Slike[i, y2, x2];
    A[b1, b2] := false;
  end; {for i}
end; {JeParOk2}
```

In v C-ju:

```
/* ob inicializaciji postavimo vse na false */
bool A[256][256];

bool JeParOk2(int x1, int y1, int x2, int y2)
{
  int i; unsigned char b1, b2; bool ok = true;
  for (i = 0; i < n; i++) {
    b1 = slike[i][y1][x1]; b2 = slike[i][y2][x2];
    if (A[b1][b2]) { ok = false; break; }
    A[b1][b2] = true; }
  for (i = 0; i < n; i++) {
    b1 = slike[i][y1][x1]; b2 = slike[i][y2][x2];
    A[b1][b2] = false; }
  return ok;
}
```

Med inicializacijo (preden prvič pokličemo JeParOk2) moramo postaviti vse elemente tabele A na false. Časovna zahtevnost je le še $O(w^2h^2n)$.

Rešitev bi lahko poskusili izboljšati še z raznimi hevristikami. Namesto da Najdi pregleduje vse piksle po vrsti, bi lahko najprej za vsak par (x, y) izračunal, koliko različnih barv imajo slike na tem položaju. Po tem številu bi piksle uredil in jih nato pregledoval tako, da imajo prednost piksli z veliko različnimi barvami. To bi bilo še posebej koristno, če je slik veliko, so pa razmeroma majhne. Mnogo parov bi lahko kar odrezali: če imamo 20 slik, pa se na položaju (x_1, y_1) pojavljajo le štiri različne barve, na (x_2, y_2) pa le tri, že vemo, da bi lahko s tem ločili največ 12 slik, ne pa 20.

10. Prepoznavanje ulomka

Če imamo število s petimi števkami za decimalno vejico, je ena možnost že ulomek oblike $c/100\,000$, pri čemer c preprosto odčitamo iz decimalk danega števila. Potem je treba le preizkusiti vse manjše imenovalce, to pa ni težko, saj jih je največ 100 000. Pri nekem konkretnem imenovalcu b potem iščemo število a, za katero je $c/100\,000 \leq a/b < (c+1)/100\,000$, torej $cb/10^5 \leq a < (c+1)b/10^5$. Ker je $b < 10^5$, je razlika med mejama $cb/10^5$ in $(c+1)b/10^5$ manjša od 1; primeren celoštevilski a je torej lahko največ en sam, to je $\lceil cb/10^5 \rceil$. Če je ta dober, prav; če ni, pa moramo poskusiti naslednji b.

```
program Ulomki;
var T: text; S: string; a, b, c, d, i: integer;
begin
   Assign(T, 'ulomki.in'); Reset(T); ReadLn(T, S); Close(T);
   d := 1; c := 0;
   for i := 3 to Length(S) do
      begin d := d * 10; c := c * 10 + Ord(S[i]) − Ord('0') end;
   b := 1; a := 0;
   while b <= d do begin
      while c * b > a * d do a := a + 1;
      if a * d < (c + 1) * b then break;
      b := b + 1;
   end; {while}
   Assign(T, 'ulomki.out'); Rewrite(T); WriteLn(T, a, ' ', b); Close(T);
end. {Ulomki}
```

Še rešitev v C-ju:

```
#include <stdio.h>

#define MaxStevk 6

int main(int argc, char** argv)
{
   char s[MaxStevk + 4]; int a, b, c, d, i;
   FILE *f = fopen(argc > 1 ? argv[1] : "ulomki.in", "rt");
   fgets(s, MaxStevk + 3, f); fclose(f);
   for (c = 0, d = 1, i = 2; '0' <= s[i] && s[i] <= '9'; i++)
      { d *= 10; c = c * 10 + s[i] − '0'; }
   for (a = 0, b = 1; b <= d; b++)
   {
      while (c * b > a * d) a++;
      if (a * d < (c + 1) * b) break;
   }
   f = fopen(argc > 2 ? argv[2] : "ulomki.out", "wt");
   fprintf(f, "%d %d\n", a, b); fclose(f); return 0;
}
```

11. Presek dreves

V nalogi se pravzaprav skrivata dva ločena problema: en je predelati nize (iz vhodne datoteke) v drevesa, drugi pa je računati presek dreves. Obojega se lahko lotimo z rekurzijo. Pri predelovanju niza v drevo lahko razmišljamo takole: če trenutni znak ni oklepaj, ustvarimo list drevesa in se po vhodnem nizu ne premaknemo

naprej (ker naloga pravi, da je list predstavljen s praznim nizom). Če pa je trenutni znak oklepaj, se začenja opis nekega notranjega vozlišča. Naslednji znak je torej črka, ki kaže na prvega otroka; sledi opis prvega otroka (ki ga predelamo v drevo z rekurzivnim klicem); podobno sledijo zdaj črke in opisi ostalih otrok; ustavimo pa se, ko preberemo še zaklepaj. Tule je primer rešitve v pythonu (trenutni položaj v nizu s hrani spremenljivka i; funkcija vrne prebrano poddrevo in novi položaj i; drevo pa predstavimo kar z razpršeno tabelo, v kateri so ključi črke na povezavah, pripadajoče vrednosti pa poddrevesa):

```
def DrevoIzNiza(s, i = 0):
    drevo = {}
    if (i == 0 and s == "") or s[i] != "(": return (drevo, i)
    i += 1                              # preskočimo oklepaj
    while s[i] != ')':
        c = s[i]; i += 1                # črka na povezavi
        (otrok, i) = DrevoIzNiza(s, i)  # preberimo poddrevo
        drevo[c] = otrok
    return (drevo, i + 1)               # preskočimo tudi zaklepaj
```

Nize, ki so prisotni v vseh drevesih, pa lahko iščemo tako, da se rekurzivno spuščamo po vseh drevesih hkrati. Če imajo vsa trenutna vozlišča prisotno poddrevo z oznako c na povezavi, se spustimo v ta poddrevesa in nadaljujemo z rekurzivnim klicem; če pa je poddrevo z oznako c prisotno le v nekaterih, ne pa v vseh, se v ta poddrevesa ne spuščamo.

```
def Presek(s, drevesa):
    if not drevesa: return
    print s
    for c in drevesa[0]:
        poddrevesa = [drevo[c] for drevo in drevesa if c in drevo]
        if len(poddrevesa) == len(drevesa): Presek(s + c, poddrevesa)
```

Zapišimo še glavni blok programa:

```
import sys
n = int(sys.stdin.readline())
drevesa = [DrevoIzNiza(sys.stdin.readline().strip())[0] for i in range(n)]
Presek("", drevesa)
```

12. Kratice

Vsako besedo si predstavljajmo kot točko grafa; od w do vsake $x \in A(w)$ gre usmerjena povezava. Graf zdaj luščimo po plasteh, od spodnjih nivojev navzgor:

```
r := 0;
while true do begin
    L_r := {vse preostale besede, ki imajo izhodno stopnjo 0};
    L_r vsebuje vse besede, ki imajo red r, ne pa tudi reda r + 1;
    if L_r = ∅ then break;
    pobriši iz grafa vse besede iz L_r;
    r := r + 1;
end; (* while *);
vse preostale besede so rekurzivne kratice;
```

Pri popolnih kraticah je zadeva precej podobna. Recimo, da smo že odkrili vse besede, ki so popolne kratice reda $r-1$ ali manj, niso pa popolne kratice reda r. Naj bo U množica vseh preostalih besed; vsaka od njih je torej reda r; katere pa so tudi reda $r+1$? Poglejmo neko $w \in U$; ker je reda r, so vse $x \in A(w)$ vsaj reda $r-1$; če je med njimi vsaj kakšna, ki ni reda r, potem je w reda r, sicer pa (če so vse $x \in A(w)$ reda r) je w reda $r+1$.

$r := 0$;
$L_r :=$ {vse besede, ki imajo izhodno stopnjo 0};
$U := W - L_r$;
while $L_r \neq \emptyset$ **do begin**
  $r := r + 1$;
  $L_r := \{w \in U : w$ kaže na kakšno $x \in L_{r-1}\}$;
  L_r vsebuje vse besede, ki so popolne kratice reda r, ne pa reda $r+1$;
  $U := U - L_r$;
end; (* *while* *);
vse besede iz U so popolne rekurzivne kratice;

Oboje se da izvesti v $O(W+E)$ časa, če je W število besed, E pa število povezav ($E = \sum_{w \in W} |A(w)|$).

13. Kopanje lukenj

To je v bistvu čisto navaden problem iskanja v širino, le graf je definiran na malo bolj zamotan način. Recimo, da imamo n držav, oštevilčenih od 1 do n. Zemljevid naj bo podan v neki tabeli velikosti $2w \times 2h$, v kateri je $z(x,y)$ številka države, ki ji pripada zaplata (x,y); če pa je ta zaplata zalita z morjem, naj bo $z(x,y)=0$. Graf lahko zdaj zgradimo z enim prehodom po zemljevidu:

na začetku imejmo graf s točkami $\{1,\ldots,n\}$ in brez povezav;
za vsako zaplato (x,y) na zemljevidu:
  $d := z(x,y)$;
  if $d = 0$ **then continue**; (* *tu je morje* *)
  če je $x < 0$, naj bo $x' := x + w$, sicer $x' := x - w$;
  $d' := z(x', -y-1)$ (* *zaplata na drugi strani sveta* *)
  če $d' \neq 0$, dodaj v graf povezavo med d in d' (če še ne obstaja);

V tako dobljenem grafu sta zdaj dve državi neposredno povezani natanko tedaj, ko lahko iz ene pridemo v drugo s kopanjem luknje navpično navzdol skozi središče zemlje. Naloga zahteva iskanje poti z najmanjšim številom izkopanih lukenj, kar se pri našem grafu prevede v pot z najmanjšim številom povezav. Zato lahko uporabimo iskanje v širino. Graf je verjetno še najlažje predstaviti z matriko sosednosti, saj držav v praksi ni veliko.

Ena podrobnost, na katero zgornja rešitev ne pazi, naloga pa jo omenja, je ta, da se lahko znotraj države prosto gibljemo le, če nam pri tem ni treba iti čez morje ali čez ozemlje kakšne druge države. Če se torej neka država sestavljena iz več nepovezanih delov, ji moramo v grafu dodeliti več točk (za vsak del po enega). Če nas kasneje zanima najkrajša pot od države s do države t, si pri iskanju v širino na začetku označimo kot dosegljive (po poti dolžine 0) vse tiste točke, ki predstavljajo

kose države s; iskanje v širino pa končamo, čim nam uspe doseči kakšno točko, ki predstavlja kak kos države t.

Razdeljevanje držav na kose lahko izvedemo s še enim dodatnim pregledom celega zemljevida (še pred gradnjo grafa):

$m := 0$;
za vsako zaplato (x, y) na zemljevidu postavi $z'(x, y) := 0$;
za vsako zaplato (x, y):
  $d := z(x, y)$;
  if $d = 0$ **then continue**; (* *tu je morje* *)
  if $z'(x, y) \neq 0$ **then continue**; (* *že obdelano* *)
  $m := m + 1$;
  zapomnimo si, da je m eden od kosov države d;
  $Q := \{(x, y)\}$; (* *v praksi vrsta ali pa sklad* *)
  $z'(x, y) := m$;
  while $Q \neq \{\}$:
    naj bo (x', y') poljuben element Q;
    $Q := Q - \{(x', y')\}$;
    za vsako zaplato (x'', y''), ki je soseda zaplate (x', y'):
      if $z(x'', y'') \neq d$ **or** $z'(x'', y'') \neq 0$ **then continue**;
      $z'(x'', y'') := m$;
      $Q := Q \cup \{(x', y')\}$;

Stari zemljevid z smo predelali v z', ki namesto številk držav hrani številke kosov; pri gradnji grafa moramo potem uporabiti z', ne pa z. Na koncu gornjega algoritma nam m pove število kosov in pri gradnji grafa bomo dobili graf na m točkah namesto na n točkah. Notranja zanka **while** obišče vse zaplate, ki pripadajo trenutnemu kosu; pri vsaki pogleda njene sosede in če pripadajo isti državi in še niso bile obiskane (torej če je $z'(x'', y'')$ še 0), jih dodamo v trenutni kos.

V praksi se sicer izkaže, da tu opisani način potovanja ni pretirano prikladen, ker je večina sveta pač pokrita z morji; četudi začneš kopati na kopnem, bo na nasprotni strani sveta zelo verjetno morje.

14. Biblijski kod

Oštevilčimo v našem stolpcu besedila vrstice od zgoraj navzdol s števili od 1 do H. Ko se z oknom premaknemo za eno vrstico navzdol po stolpcu besedila, ne bi radi, da bi bilo treba računati vse znova, torej znova odkrivati vse pojavitve besed in ugotavljati, katere črke pripadajo vsaj eni pojavitvi.

Določena črka (x, y) v oknu lahko pripada več pojavitvam raznih besed; za vsako pojavitev poglejmo njeno najmanjšo y-koordinato; med vsemi pojavitvami si zapomnimo tisto, pri kateri je ta minimum največji; ta maksimum minimumov označimo s $M(x, y)$. Če črka ne pripada nobeni pojavitvi, je $M(x, y) = -\infty$. Ta podatek je zelo koristen iz naslednjega razloga: ko se bomo z oknom premikali navzdol po stolpcu besedila in bo iz okna izpadla vrstica Y, bomo vedeli, da črke, ki imajo $M(x, y) \leq Y$, zdaj niso več pokrite (ker če jih že pokriva kakšna pojavitev kakšne besede, sega ta beseda v vrstico Y ali še kakšno zgodnejšo, torej štrli že ven iz našega okna).

Recimo, da smo že obdelali okno, ki ga tvorijo vrstice $Y, \ldots, Y+h-1$. Zdaj nas zanima okno $Y+1, \ldots, Y+h$. Edina stvar, ki jo novo okno vsebuje, staro pa je ni, je vrstica $Y+h$; vrednosti $M(x,y)$ moramo torej zdaj popraviti tako, da bodo upoštevale tudi tiste pojavitve, ki segajo v vrstico $Y+h$.

Poleg slovarja besed imejmo še obrnjeni slovar, v katerem so besede zapisane od desne proti levi; vsakega od njiju predstavimo z drevesom (*trie*), ki ima za vsako besedo po eno vejo, na povezavah so posamezne črke te besede, če pa se več besed začne na isto zaporedje črk, si ustrezne povezave tudi delijo. Preglejmo vse trojice $(x_0, \triangle x, \triangle y)$, za y_0 pa vzamemo $Y+h$. Za $\triangle y$ seveda pridejo v poštev le nepozitivna števila (da ne pademo ven iz okna). Za vsak nabor vrednosti $x_0, y_0, \triangle x, \triangle y$ poglejmo, če nam določa kakšno pojavitev kakšne besede iz slovarja:

```
algoritem A(x0, y0, △x, △y):
  d := 0; k := 0;
  p := koren drevesa besed; r := koren drevesa obrnjenih besed;
  while true:
    x := x0 + k△x; y := y0 + k△y;
    (* Preverimo, če smo padli ven iz okna (pokriva vrstice Y + 1, . . . , Y + h). *)
    if x < 1 or y ≤ Y or x > w then break;
    c := črka na polju (x, y) našega stolpca besedila;
    p := tisti otrok vozlišča p, v katerega kaže povezava s črko c;
    r := tisti otrok vozlišča r, v katerega kaže povezava s črko c;
      (če takih otrok ni, postavimo p oz. r na NIL);
    if p = NIL and r = NIL then
      break; (* od tu naprej ne bomo našli nobene pojavitve več *)
    d := d + 1;
    if se v vozlišču p ali r konča kakšna beseda then
      d := k; (* našli smo neko pojavitev neke besede iz slovarja *)
  end while;
  u := y0 + (d − 1)△y; (* Najvišja dosežena vrstica. *)
  for k := 0 to d − 1:
    x := x0 + k△x; y := y0 + k△y;
    if M(x, y) < u then M(x, y) := u;
  end for;
```

Tako smo poiskali najdaljšo pojavitev kakšne besede, ki se začne v (x_0, y_0) in uporablja smer $(\triangle x, \triangle y)$ ali pa se konča v (x_0, y_0) in uporablja smer $(-\triangle x, -\triangle y)$ (to slednje smo dosegli tako, da uporabljamo poleg drevesa prvotnih besed še drevo obrnjenih besed). Ko smo videli, da je ta beseda dolga recimo d znakov, lahko tudi izračunamo najmanjšo y-koordinato, ki jo njena pojavitev doseže (namreč $y_0 + (d-1)\triangle y$; spomnimo se, da je $\triangle y$ nepozitivna), s katero lahko zdaj popravimo vrednosti $M(x,y)$ za tista polja, ki jih je naša pojavitev dosegla.

Celoten postopek po premiku okna je torej takšen (recimo, da okno po premiku navzdol pokriva vrstice $Y+1, \ldots, Y+h$):

```
  for x0 := 1 to w do
    for △x := −(w − 1) to w − 1 do
      for △y := −(h − 1) to 0 do if △x ≠ 0 or △y ≠ 0 then
```

```
      poženi algoritem A(x_0, Y + h, △x, △y);
n := 0;
for y := Y + 1 to Y + h do for x := 1 to w do
  if M(x, y) ≥ Y + 1 then n := n + 1;
```

Na koncu imamo v n število črk v trenutnem oknu, ki jih pokriva kakšna taka pojavitev kakšne slovarske besede, ki tudi sama v celoti leži v tem oknu. Med tako dobljenimi n-ji (za vse možne položaje okna, torej vse Y) si moramo zapomniti največjega; to je rezultat, po katerem nas sprašuje naloga.

Za začetni položaj okna čisto na vrhu stolpca, torej takrat, ko pokriva vrstice od 1 do h, lahko postopek inicializiramo tako, da se pretvarjamo, da je nad vrstico 1 še h vrstic, v katerih so sami taki znaki, ki se ne pojavljajo v nobeni besedi (npr. presledki). Na začetku inicializiramo vse $M(x, y)$ na $-\infty$, nato pa okno počasi premikamo navzdol in po prvih h premikih bo pokrilo ravno prvih h vrstic prvotnega stolpca besedila, med tem pa bomo že tudi izračunali pravilne vrednosti $M(x, y)$ za ta položaj okna.

Razmislimo še o časovni zahtevnosti tega postopka. Ko za nek konkreten začetni položaj (x_0, y_0) in zamik $(\triangle x, \triangle y)$ preverjamo, ali nam določa kakšno pojavitev kakšne besede iz slovarja, gotovo ne bo treba preveriti več kot $\min\{w/|\triangle x|, h/|\triangle y|\}$ znakov (saj potem pademo ven iz okna). Glavna zanka algoritma A torej takrat naredi največ toliko iteracij, v vsaki iteraciji pa ima le $O(1)$ dela. Seštejmo zdaj to po vseh $(\triangle x, \triangle y)$ pri fiksnem začetnem položaju (x_0, y_0):[13]

$$\begin{aligned}
&\textstyle\sum_{\triangle x=0}^{w-1} \sum_{\triangle y=0}^{h} \min\{w/\triangle x, h/\triangle y\} = \\
&\textstyle\sum_{\triangle x=0}^{w-1} \left[\sum_{\triangle y=0}^{h} w/\triangle x \cdot [\![w/\triangle x < h/\triangle y]\!] + \sum_{\triangle y=1}^{h} h/\triangle y \cdot [\![w/\triangle x \geq h/\triangle y]\!]\right] = \\
&\textstyle\sum_{\triangle x=0}^{w-1} \left[\sum_{\triangle y=0}^{h} w/\triangle x \cdot [\![\triangle y < \triangle x h/w]\!] + \sum_{\triangle y=1}^{h} h/\triangle y \cdot [\![\triangle y \geq \triangle x h/w]\!]\right] \approx \\
&\textstyle\sum_{\triangle x=0}^{w-1} \left[(\triangle x h/w) w/\triangle x + \sum_{\triangle y=\triangle x h/w}^{h} h/\triangle y\right] \approx \\
&\textstyle\sum_{\triangle x=0}^{w-1} [h + h(\ln h - \ln(\triangle x h/w))] = \\
&\textstyle\sum_{\triangle x=0}^{w-1} h\,[1 + \ln w/\triangle x] = \\
&h[w + (\ln w)^2] = O(wh).
\end{aligned}$$

To je spodbuden rezultat: čeprav imamo $O(wh)$ različnih parov $(\triangle x, \triangle y)$ in imamo pri nekaterih parih kar $O(w)$ ali $O(h)$ dela, pa je pri večini parov (tistih, pri katerih je vsaj ena od $\triangle x$ in $\triangle y$ velika) dela tako malo, da je skupna časovna zahtevnost le $O(wh)$, ne pa npr. $O(w^2 h)$ ali kaj podobnega. Kakorkoli že, doslej smo sešteli časovno zahtevnost algoritma A po vseh parih $(\triangle x, \triangle y)$ pri fiksni vrednosti x_0; ker pa moramo po vsakem premiku okna pregledati kar w možnih vrednosti x_0 (od 1 do w), imamo pri oknu vsega skupaj $O(w^2 h)$ dela. Poleg tega imamo pri oknu na koncu še $O(wh)$ dela s štetjem črk, ki pripadajo kakšni pojavitvi. Ker imamo $O(H)$ položajev okna, je skupna časovna zahtevnost naše rešitve $O(w^2 hH)$. Pri omejitvah, o katerih govori naloga ($w = 100$, $h = 30$, $H = 10000$) bi bilo to še nekako obvladljivo.

[13] Pri izpeljavi smo si pomagali z Iversonovimi oklepaji: $[\![P]\!]$ ima vrednost 1, če je pogoj P izpolnjen, sicer pa ima vrednost 0. Pomagali smo si tudi z dejstvom, da je $\sum_{\triangle y=1}^{b}(1/\triangle y) \approx \gamma + \ln b$ za neko konstanto $\gamma \approx 0{,}577$, zato je $\sum_{\triangle y=a}^{b}(1/\triangle y) \approx \ln b - \ln(a-1) \approx \ln(b/a)$.

15. Izpeljava

Zanima nas, kako iz čim krajšega niza x z uporabo danih pravil dobiti dani niz w. Recimo, da bo x dolg m znakov in ga lahko zapišemo kot $x_1 x_2 \ldots x_m$. To, kar se med uporabo pravil razvije iz npr. znaka x_1, nič ne vpliva na to, kar se razvije iz znaka x_2 in podobno — to dejstvo je posledica tega, da imajo vsa naša pravila na levi strani le po eno črko. Če torej iz x nastane z uporabo pravil niz w, to pomeni, da iz vsakega znaka x_i nastane nek podniz w_i in ti podnizi skupaj tvorijo w: torej $w = w_1 w_2 \ldots w_m$.

Naloga pravi, da so v w same male črke, torej morajo biti tudi w_i sestavljeni iz samih malih črk. Če je x_i že sam po sebi mala črka, je $w_i = x_i$, saj z našimi pravili malih črk ne moremo spreminjati (na levi strani imajo vsa samo velike črke). Za velike črke pa lahko opazimo, da lahko iz vsake velike črke nastane z našimi pravili največ en tak niz, v katerem so same male črke. Če nas na primer zanima velika črka `A`, obstaja največ eno pravilo, ki ima `A` na levi strani; za vsako veliko črko na desni strani tistega pravila obstaja spet največ eno pravilo, ki ima na levi strani tisto veliko črko; z uporabo teh pravil se bomo prej ali slej znebili vseh velikih črk v nizu ali pa prišli v situacijo, ko vsebuje niz same take velike črke, ki niso na levi strani nobenega pravila; v tem primeru bomo vedeli, da iz `A` ni mogoče dobiti niza samih malih črk. Ni se nam treba bati, da bi se ta postopek zaciklal, kajti velike črke na desni strani vsakega pravila so po abecedi kasnejše od tistih na levi strani. To pomeni, da med uporabo pravil nikoli ne bomo dobili kakšnega novega `A`-ja; kakšen nov `B` lahko dobimo le, ko se znebimo kakšnega `A`-ja; kakšen nov `C` lahko dobimo le, ko se znebimo kakšnega `A`-ja ali `B`-ja; in tako naprej. Število novih velikih črk, ki jih lahko dobimo med uporabo pravil, je torej omejeno in postopek se bo prej ali slej končal.

Naj bo zdaj Σ množica, ki vsebuje vse male črke in vse tiste velike črke, iz katerih se da z uporabo naših pravil izpeljati nek niz samih malih črk. Za vsako črko $\alpha \in \Sigma$ označimo z $f(\alpha)$ tisti niz samih malih črk, ki se ga da izpeljati iz α (če je α mala črka, je $f(\alpha) = \alpha$, sicer pa je $f(\alpha)$ tisti niz, ki smo ga dobili po postopku iz prejšnjega odstavka).

Zdaj nam ostane le še vprašanje, kako w sestaviti s stikanjem čim manjšega števila nizov $f(\alpha)$ za razne $\alpha \in \Sigma$. Koristno je to vprašanje še malo posplošiti — namesto le za cel niz w si ga zastavimo tudi za vse začetke (prefikse) tega niza. Recimo, da je w dolg n znakov, $w = a_1 a_2 \ldots a_n$, in da za nek njegov začetek $a_1 a_2 ... a_k$ že poznamo najkrajši niz, iz katerega se ga da izpeljati — recimo, da je to niz u_k (dolg pa je d_k znakov). Potem za vsak $\alpha \in \Sigma$ vemo, da lahko iz niza $u_k \alpha$ izpeljemo niz $a_1 a_2 ... a_k f(\alpha)$. Če je ta niz tudi začetek w-ja (torej če je $f(\alpha)$ začetek niza $a_{k+1} a_{k+2} \ldots a_n$), je $u_k \alpha$ eden od kandidatov za najkrajši niz, iz katerega se da izpeljati ta začetek w-ja; če je krajši od najkrajšega doslej znanega, si ga zapomnimo. Tako nadaljujemo proti vse daljšim začetkom niza w in na koncu dobimo rešitev tudi za niz w kot celoto.

$\Sigma := \{\}$;
za vsako malo črko $\alpha = \mathtt{a}, \ldots, \mathtt{z}$: $f(\alpha) := \alpha$; $\Sigma := \Sigma \cup \{\alpha\}$;
za vsako veliko črko $\alpha = \mathtt{Z}, \ldots, \mathtt{A}$ (v tem vrstnem redu!):
  if ne obstaja pravilo z α na levi strani **then continue**;
  sicer naj bo y desna stran tega pravila;

  if je v y kakšna črka, ki ni v Σ, **then continue**;
  označimo črke niza y z $y_1 y_2 \ldots y_k$;
  $f(\alpha) := f(y_1) f(y_2) \ldots f(y_k)$; (* *stik vseh teh nizov* *)
  $\Sigma := \Sigma \cup \{\alpha\}$;

naj bo n dolžina niza w;
za $i := 1, \ldots, n$: $d_i := \infty$
za $i := 0, 1, \ldots, n$:
  if $i > 0$ **and** $d_i = \infty$ **then continue**;
  za vsako črko $\alpha \in \Sigma$:
    če se $w_{i+1} w_{i+2} \ldots w_n$ začne na $f(\alpha)$:
      $j := i + |f(\alpha)|$;   (* $|f(\alpha)|$ *je dolžina niza* $f(\alpha)$ *)
      če je $d_i + 1 < d_j$:   (* *zapomnimo si to rešitev in uporabljeni znak* α *)
        $d_j := d_i + 1$; $z_j := \alpha$;

(* *S pomočjo tabele z sestavimo primeren niz x.* *)
$x :=$ (prazen niz); $i := n$;
while $i > 0$:
  $x := z_i x$;
  $i := i - |f(z_i)|$;
return x;

Če so nizi $f(\alpha)$ dolgi in jih ima po več tudi skupen začetek, utegne biti za učinkovitejše preverjanje, ali se $w_{i+1} \ldots w_n$ začne na $f(\alpha)$ (oz. pri katerih α to drži, pri katerih pa ne), koristno, če nize $f(\alpha)$ zložimo v drevo (*trie*).

Razmislimo še o težji različici naloge, pri kateri je lahko več pravil z isto črko na levi strani, poleg tega pa ni omejitev glede tega, katere črke se lahko pojavljajo na desni strani pravila. Osnovna ideja naše prejšnje rešitve je še vedno uporabna tudi tu: če poznamo najkrajši niz x_k, iz katerega je mogoče izpeljati $w_1 \ldots w_k$, in če se $w_{k+1} \ldots w_n$ začne na nek niz $w_{k+1} \ldots w_{k+t}$, ki ga je mogoče izpeljati iz črke α, potem je mogoče $w_1 \ldots w_{k+t}$ izpeljati iz $x_k \alpha$, torej je $x_k \alpha$ kandidat za najkrajši tak niz (torej za x_{k+t}). Razlika v primerjavi s prvotno različico naloge je le ta, da se zdaj lahko zgodi, da je mogoče iz posamezne črke α izpeljati veliko različnih nizov, ne le enega samega. Na srečo pa nam tega, kaj je mogoče izpeljati iz posamezne α, ni treba ugotavljati za vse mogoče nize, pač pa le za podnize našega niza w. V ta namen bi si lahko pomagali z znanim algoritmom CYK (Cocke-Younger-Kasami), ki temelji na dinamičnem programiranju.

Mimogrede, skupkom pravil, s kakršnimi se ukvarja ta naloga, pravimo tudi „kontekstno neodvisne gramatike“ (*context-free grammars*) in imajo pomembno vlogo v teoretičnem računalništvu, pri opisovanju sintakse programskih jezikov ipd.

16. Tetris

Naloge se lahko lotimo z rekurzijo: začnemo s prazno igralno površino in na vse možne načine dodamo vanjo prvi lik. Pri vsaki od teh možnosti nato izvedemo rekurzivni klic, ki bo poskusil na vse možne načine dodati v igralno površino drugi lik in tako naprej. Stanje igralne površine je dovolj pri vsakem klicu opisati že s tem, da za vsak stolpec povemo višino najvišjega zasedenega polja v njem — samo od tega je namreč odvisno, na kakšni višini se bo pri padanju ustavil naslednji lik.

Lahko se zgodi, da je mogoče neko stanje igralne površine doseči na več načinov (z isto skupino prvih nekaj likov); v tem primeru bi po nepotrebnem zapravljali čas, če bi poskušali pri vsakem od teh načinov nadaljevati s postavljanjem preostalih likov. Zato je koristno, če si v pomnilniku (npr. v neki razpršeni tabeli) hranimo podatke o tem, katera stanja igralne površine smo že dosegli.

podprogram *Rekurzija*(k, s):
Vhodni podatki: k je zaporedna številka naslednjega lika;
s je w-terica celih števil, ki opisuje stanje igralne površine.
Postopek:
za vsako možno obliko, ki jo lahko k-ti lik doseže z vrtenjem:
za vsako možno x-koordinato (od 1 do $w + 1 -$ *širina lika*):
naj bo s' stanje igralne površine, ki ga dobimo iz s,
če odvržemo lik pri trenutnem zasuku in položaju x;
if $s' \in M_k$ **then continue**;
dodaj s' v množico M_k;
$v := \max\{s'_i : 1 \le i \le w\}$ (* *višina trenutne zgradbe* *)
if $v \ge v^\star$ **then continue**; (* *trenutna zgradba je že previsoka* *)
če je k zadnji lik v zaporedju:
$v^\star := v$; zapomni si trenutno rešitev s';
sicer:
Rekurzija($k + 1$, s');

Pri tem je $v^\star$ globalna spremenljivka, ki hrani višino najboljše doslej dosežene rešitve (na začetku jo inicializiramo na $+\infty$). Množice M_k pa so globalne spremenljivke, ki hranijo podatke o tem, katera stanja smo že dosegli. Ob inicializaciji programa naj bodo vse te množice prazne. V praksi bi jih lahko predstavili z razpršenimi tabelami.

Glavni klic rekurzije naj začne pri prvem liku, torej $k = 1$, in s prazno igralno površino, torej $s = (0, 0, \ldots, 0)$.

Za učinkovitejše ugotavljanje tega, kako globoko pade lik in kakšno je novo stanje igralne površine, si je koristno za vsak lik pripraviti tabelo z y-koordinatami najvišjih in najnižjih zasedenih polj v vsakem stolpcu. Na primer:

2
1
$y = 0$
$x = 0$ 1

	x = 0	1
najnižje zasedeno polje	1	0
najvišje zasedeno polje	1	2

Naj bo torej d širina našega lika, l_i in u_i pa naj bosta y-koordinati najnižjega oz. najvišjega zasedenega polja v stolpcu i (za $i = 0, \ldots, d - 1$). Recimo, da vržemo lik na igralno površino (ki je bila doslej v stanju s) tako, da se njegov levi stolpec ujema z x-tim stolpcem igralne površine; in recimo, da pristane lik na taki višini, da je njegova spodnja vrstica v y-ti vrstici igralne površine. Ker se novi lik ne sme prekrivati s tistimi, ki so bili že od prej na površini, velja pri vsakem i (od 0 do $d - 1$) pogoj $y + l_i > s_i$. Najnižja višina, ki jo lahko lik doseže (in se potem tam njegovo padanje ustavi) je torej $y = \max\{s_{x+i} - l_i + 1 : 0 \le i < d\}$. Novo stanje s'

pa potem dobimo takole: $s'_{x+i} = y + u_i - 1$ za $0 \le i < d$; pri drugih stolpcih (tistih, na katere novi lik ne vpliva) pa je $s'_i = s_i$.

Takšni tabeli l_i-jev in u_i-jev si je koristno pripraviti vnaprej za vsak lik v vseh možnih rotacijah.

17. Letalski poleti

(a) Pri tej nalogi imamo pravzaprav opravka s problemom najkrajših poti v grafu, le da vnaprej še ne vemo, katere povezave bomo sploh lahko uporabili — nek let od u do v lahko uporabimo le, če uspemo v u priti dovolj časa pred začetkom tega leta. Pri običajnem problemu najkrajših poti v grafu pa lahko neko povezavo uporabimo ne glede na to, kako smo prišli v začetno krajišče te povezave. Oglejmo si, kako lahko za potrebe te naloge prilagodimo Dijkstrov algoritem za iskanje najkrajših poti po grafu.

Recimo, da so letališča oštevilčena od 1 do n, začnemo v času T_0 na letališču s in bi radi čim hitreje prišli na letališče t; čas, potreben, da presedemo z enega letala na drugo, pa je p enot. Vzdrževali bomo množico M, v kateri bodo vsa letališča, do katerih že poznamo neko pot, ki pa še ni nujno najkrajša. Najzgodnejši čas, do katerega znamo priti na letališče u, označimo z d_u; zadnji let na tej poti (torej tisti let, s katerim smo pristali v u ob času d_u) pa označimo z λ_u.

for $u = 1, \ldots, n$: $d_u := \infty$;
$M := \{s\}$; $d_s := T_0$; $\lambda_s :=$ **nil**;
while $M \neq \{\}$:
  $u :=$ tisto letališče iz M, ki ima najzgodnejši d_u;
  (* *Pokazati je mogoče, da če izberemo u na ta način, je najboljša doslej znana pot do u že tudi najboljša pot do u sploh.* *)
  za vsak let l, ki odleti iz u in to ne prej kot ob času $d_u + p$:
    naj bo v cilj tega leta in t čas pristanka;
    if $t < d_v$: $d_v := t$; $\lambda_v := l$; $M := M \cup \{v\}$;

Množico M bi lahko hranili v tabeli ali seznamu, bolj učinkovito pa je, če jo predstavimo s kopico (*heap*); takrat lahko u z najzgodnejšim časom d_u poiščemo v času $O(\log n)$ namesto $O(n)$. Časovna zahtevnost tega postopka je potem $O((n + m) \log n)$, če je m število vseh letov.

Na koncu nam d_z pove najzgodnejši čas, v katerem lahko pridemo do z; če pa hočemo izpisati tudi zaporedje uporabljenih letov, jih lahko odčitamo iz tabele λ.

(b) Pri tej podnalogi ni dovolj, da poiščemo za vsako letališče zgolj najcenejšo pot do njega. Recimo, da imamo do u dve poti, pri čemer prva doseže u ob času t_1, druga pa ob času t_2 in je $t_1 < t_2$; in recimo še, da je prva pot dražja od druge. Lahko se zgodi, da obstaja nek zelo poceni let iz u v z, ki zapusti u nekje v časovnem intervalu $[t_1 + p, t_2 + p)$, in lahko se hkrati tudi zgodi, da so vsi kasnejši leti iz u zelo dragi. V tem primeru, če bi gledali le najcenejšo pot do u, bi bila to tista s pristankom ob času t_2, pri kateri pa je potem nadaljevanje poti (do z) zelo drago; mogoče je boljša kakšna taka pot, ki najprej na malo dražji način pride do u (ob času t_1), zato pa lahko nato ceneje nadaljuje pot v z.

Namesto ene najcenejše poti za vsak u bomo raje za vsak polet l gledali najcenejšo pot, ki se konča s tem letom. Za let l naj bo d_l cena te poti, λ_l pa predzadnji polet na tej poti. Cena leta l samega po sebi pa naj bo c_l.

za vsak let l: $d_l := \infty$;
$M := \{\}$;
za vsak let l, ki poleti iz s in to ne pred časom T_0:
  $M := M \cup \{l\}$; $d_l := c_l$;
while $M \neq \{\}$:
  $l :=$ tisti let iz M, ki ima najnižjo ceno d_l;
  naj bo u letališče, na katero prileti ta let, in naj bo t čas pristanka;
  za vsak let l', ki poleti z u in to ne prej kot ob času $t + p$:
    if $d_l + c_{l'} < d_{l'}$: $d_{l'} := d_l + c_{l'}$; $\lambda_{l'} := l$; $M := M \cup \{l'\}$;

Na koncu moramo med vsemi leti, ki pristanejo v z najkasneje ob času T_1, poiskati tistega z najmanjšo d_l — to je potem cena najcenejšega zaporedja letov, ki pravočasno doseže letališče z.

Množico M lahko tudi tokrat predstavimo s kopico in časovna zahtevnost postopka bo v tem primeru $O(m^2 \log m)$. Časovna zahtevnost je večja kot pri podnalogi (a) zato, ker lahko lete s posameznega letališča u pregledamo večkrat (po enkrat za vsak let, ki pristane na u), pri (a) pa smo jih pregedali le enkrat (ko smo vzeli u iz množice M).

Oglejmo si še dve drobni izboljšavi, ki bi lahko mogoče v nekaterih primerih pospešili delovanje algoritma in se izognili nepotrebnemu večkratnemu pregledovanju nekaterih letov. Glavna zanka pregleduje lete (jih jemlje iz M) po naraščajoči ceni d_l; recimo, da smo pravkar vzeli iz M nek let l, ki pristane v u ob času t, in da smo pred tem nekoč prej že vzeli iz M nek let l', ki je tudi pristal v u ob času t ali prej. Ker smo l' vzeli iz M prej kot l, je pot do njega cenejša ($d_{l'} \leq d_l$), in ker je poleg tega l' pristal v u prej kot l, je vsako nadaljevanje poti, ki je možno iz l, možno tudi iz l'. Zato se nam z l-jem sploh ni treba uvarjati, ker ne bomo z njim prišli do nobene cenejše poti (do z) kot z letom l'. Koristno je torej, če si za vsak u v neki tabeli hranimo najzgodnejši čas pristanka med vsemi leti, ki pristanejo na u in smo jih doslej vzeli iz M; recimo temu času t_u. Ko vzamemo nek l (ki pristane na u ob času t) iz M, poglejmo, če je $t < t_u$; če ni, se z l-jem ni treba več ukvarjati; če pa je, postavimo $t_u := t$ in nadaljujmo na običajen način.

Druga izboljšava pa je tale: na začetku lahko uporabimo postopek, zelo podoben tistemu iz podnaloge (a), da za vsako letališče u ugotovimo, kateri je najkasnejši čas, ob katerem moramo odpotovati z u, če hočemo doseči U ne kasneje kot ob času T_1:

for $u = 1, \ldots, n$: $d_u := -\infty$;
$M := \{z\}$; $d_z := T_1$;
while $M \neq \{\}$:
  $u :=$ tisto letališče iz M, ki ima najkasnejši d_u;
  za vsak let l, ki prileti na u in to ne kasneje kot ob času $d_u - p$:
    naj bo t čas začetka tega leta in v letališče, s katerega prileti;
    if $t > d_v$: $d_v := t$; $M := M \cup \{v\}$;

Zdaj lahko pobrišemo vse lete, ki poletijo prepozno: če nek let poleti z letališča u kasneje kot ob času d_u, je za naše potrebe neuporaben, ker s takim letom gotovo ne bomo mogli pravočasno priti do z-ja. Na tako okleščenem grafu poletov lahko uporabimo enak postopek za iskanje najcenejših poti kot prej na prvotnem grafu.

18. Osem

Praznemu polju v mislih pripišimo številko 9; potem si lahko stanje igre predstavljamo kot permutacijo števil od 1 do 9. Takih permutacij je $9! = 362\,880$. Mislimo si jih lahko kot točke neusmerjenega grafa, v katerem obstaja povezava med π in ρ natanko tedaj, ko lahko iz π naredimo ρ (ali obratno) v eni potezi (torej da na prazno polje premaknemo eno od sosednjih štirih ploščic). Vsakemu zaporedju potez pri igri zdaj ustreza neka pot po tem grafu, mi pa moramo poiskati najkrajšo pot od začetnega stanja igre do končnega (tistega, v katerem je vsaka ploščica na svojem mestu; to je ravno identična permutacija). Tega se lahko lotimo z iskanjem v širino. Ker ima graf le 362 880 točk in vsaka točka največ štiri sosede, je to kar obvladljivo.

Izkaže se, da je igra rešljiva ravno pri polovici izmed vseh možnih 362 880 stanj. Če pot do ciljnega stanja obstaja, je dolga največ 31 potez; je pa tako dolga pot potrebna le pri dveh začetnih stanjih:

6	4	7
8	5	
3	2	1

8	6	7
2	5	4
3		1

Pri vseh drugih stanjih je najkrajša pot do cilja krajša; v povprečju je dolga le 21,97 potez.[14]

Pri iskanju v širino moramo biti sposobni hitro ugotoviti, ali smo neko stanje že dosegli kdaj v preteklosti; v ta namen je koristno, če znamo stanja oštevilčiti. Oglejmo si postopek, ki vsaki permutaciji števil od 1 do n pripiše neko zaporedno številko od 0 do $n! - 1$. Zgledovali se bomo po postopku za preračunavanje med različnimi številskimi sestavi, le da je pri permutaciji stvar taka, kot da bi se nam osnova našega sestava ves čas spreminjala: na prvem mestu v permutaciji imamo n možnosti, katero število postavimo tja; na drugem mestu imamo le še $n - 1$ možnosti (ker prvega števila ne smemo uporabiti še enkrat), na tretjem le $n - 2$ možnosti in tako naprej. V neki množici A hranimo podatek o tem, katerih števil še nismo uporabili. Ko gledamo na primer i-to število v permutaciji, recimo mu $\pi(i)$, imamo v A še $n - i + 1$ kandidatov za to število. Najmanjšemu med njimi pripišimo vrednost 0, naslednjemu 1 in tako naprej (v spodnji psevdokodi je ta vrednost označena z d); nato to vrednost še pomnožimo z $(n - i)!$ (to v spodnji psevdokodi hrani spremenljivka m), ker se da preostalih $n - i$ števil v permutaciji razporediti na $(n - i)!$ načinov.

algoritem ŠtevilkaPermutacije(π):

[14] Glej tudi zaporedje A001122 v *The On-Line Encyclopedia of Integer Sequences*, ki za vsako število potez pove, pri koliko stanjih je najkrajša pot do rešitve dolga natanko toliko potez. Igra „Osem“, s katero se ukvarjamo pri tej nalogi, je poenostavljena različica bolj razširjene igre „Petnajst“, pri kateri imamo mrežo 4×4 namesto 3×3. Tam je možnih že 16! stanj, kar je skoraj 21 bilijonov, zato tu opisani postopek z iskanjem v širino tam ni več primeren. Izkaže se, da je pri igri „Petnajst“ največje potrebno število potez enako 80 (gl. OEIS, zaporedje A087725).

$m := 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1)$;
$A := \{1, 2, \ldots, n\}$;
$x := 0$;
za $i = 1, 2, \ldots, n$:
  $d :=$ število elementov A, ki so manjši od $\pi(i)$;
  $x := x + d \cdot m$;
  $A := A - \{\pi(i)\}$; **if** $i < n$ **then** $m := m/(n-i)$;
vrni x;

Še obraten postopek:

algoritem PermutacijaIzŠtevilke(x):
$m := 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1)$;
$A := \{1, 2, \ldots, n\}$;
za $i = 1, 2, \ldots, n$:
  $d := \lfloor x/m \rfloor$; $x := x \bmod m$;
  $\pi(i) := (d+1)$-vi najmanjši element množice A;
  $A := A - \{\pi(i)\}$; **if** $i < n$ **then** $m := m/(n-i)$;
vrni π;

V praksi je koristno množico A predstaviti kar z n-bitnim celim številom.

19. Črna škatlica

(a) Da bo manj pisanja, označimo vhodni bit z a, namesto tabele Prehod si mislimo funkcijo $\delta(q, a)$, namesto tabele Izhod pa funkcijo $\beta(q, a)$.

Vrednost q v trenutku, ko se začnemo ukvarjati z našo črno škatlico, označimo s q_0. Hitro lahko vidimo, da tega začetnega stanja ne bomo mogli ugotoviti, vendar nas to tudi ne bo čisto nič oviralo pri delu. Mislimo si namreč namesto naše škatlice s stanjem q_0 in funkcijama δ in β neko drugo škatlico z začetnim stanjem $1 - q_0$ in funkcijama δ' in β', definiranima takole: $\delta'(q, a) = \delta(1-q, a)$ in $\beta'(q, a) = \beta(1-q, a)$. Očitno je, da bi dajali pri katerem koli zaporedju vhodov obe škatlici tudi enako zaporedje izhodov, pri tem pa bi bila druga škatlica ves čas v ravno nasprotnem stanju kot prva. Med tema dvema škatlicama torej ne bomo mogli ločiti, kar pa nas ne bo motilo, saj naloga zahteva le, da znamo napovedovati izhod škatlice, ta pa je pri obeh škatlicah enak. V nadaljevanju lahko torej brez izgube za splošnost predpostavimo, da je začetno stanje škatlice kar 0 (torej $q_0 = 0$).

Zdaj moramo ugotoviti še kaj o funkcijah $\delta(q, a)$ in $\beta(q, a)$. Ker imata tako q kot a po dve možni vrednosti, lahko funkcijo δ računamo pri štirih parih (q, a); pri vsakem imamo za vrednost funkcije δ dve možnosti (0 ali 1), tako da si lahko funkcijo δ izberemo na $2^4 = 16$ načinov. Enako opazimo tudi pri funkciji β. To nam da skupaj $16 \cdot 16 = 256$ možnih črnih škatlic; lahko jih kar oštevilčimo od 0 do 255. Škatlici $f \in \{0, \ldots, 255\}$ pripadata funkcija prehodov $\delta_f(q, a)$ in izhodna funkcija $\beta_f(q, a)$. Zdaj bi radi ugotovili, katera med temi 256 škatlicami je naša.

Recimo, da smo škatlici na vhod že poslali nekaj bitov in na podlagi dosedanjih izhodov za nekatere škatlice že vemo, da naša gotovo ni ena izmed njih. Vzdrževali bomo torej neko množico $M \subseteq \{0, \ldots, 255\}$ škatlic, za katere še obstaja možnost, da je naša ena izmed njih. (Na začetku, preden pošljemo škatlici kakšen vhod, so seveda odprte še vse možnosti in imamo $M = \{0, \ldots, 255\}$.) Za vsako škatlico

$f \in M$ vodimo tudi podatek o tem, v katerem stanju $q_f \in \{0, 1\}$ bi se nahajala, če bi bila na začetku v stanju 0 in bi ji nato poslali takšno zaporedje vhodov, karkšnega smo doslej zares poslali naši opazovani škatlici.

Mogoče so v M zdaj ostale same take škatlice, ki se bodo v nadaljevanju vse obnašale enako: z drugimi besedami, mogoče za vsako zaporedje vhodov $a_1, \ldots, a_t$ velja, da je zaporedje izhodov, ki ga dobimo, če postavimo škatlico f v stanje q_f in ji na vhod pošljemo bite $a_1, \ldots, a_t$, enako pri vseh $f \in M$. Če to drži, pomeni, da z nadaljnjim opazovanjem ne bomo mogli ugotoviti, katera od teh škatlic je naša, kar pa za nas tudi ni pomembno, saj se glede izhodov vse obnašajo enako. Zato lahko delovanje naše opazovane škatlice simuliramo tako, da si izberemo poljubno $f \in M$, jo postavimo v stanje q_f in simuliramo njeno delovanje. V tem primeru je naša naloga končana.

Recimo pa, da pogoj iz prejšnjega odstavka ne drži. To pa pomeni, da obstaja neko zaporedje vhodov $a_1, \ldots, a_t$ in nek par škatlic $f, g \in M$, ki nam dasta, če ju postavimo v stanji q_f oz. q_g in jima na vhod pošljemo bite $a_1, \ldots, a_t$, dve različni zaporedji izhodov. Med takšnimi zaporedji $a_1, \ldots, a_t$ izberimo najkrajše; to tudi pomeni, da do razlike v izhodu pride šele v zadnjem koraku; torej f vrača na primer izhode $b_1, \ldots, b_{t-1}, b_t$, škatlica g pa vrača izhode $b_1, \ldots, b_{t-1}, b'_t$ in je $b'_t \neq b_t$. Označimo z r_0 trenutno stanje škatlice f, torej $r_0 = q_f$; nadaljnja stanja (pri našem zaporedju vhodov) lahko torej dobimo po formuli $r_{i+1} = \delta_f(r_i, a_i)$. Podobno vzemimo $r'_0 = q_g$ in $r'_{i+1} = \delta_g(r'_i, a_i)$. Oglejmo si zdaj pare $(r_0, r'_0), (r_1, r'_1), \ldots, (r_{t-1}, r'_{t-1}), (r_t, r'_t)$. Za vsak par so le štiri možnosti — stanje škatlice f je lahko 0 ali 1, pa tudi stanje škatlice g je lahko le 0 ali 1. Tu pa imamo, če odmislimo zadnjega, t parov; torej, če je $t \geq 5$, je tu vsaj pet parov in je nemogoče, da bi bili vsi različni. Recimo, da je $r_i = r_j$ in $r'_i = r'_j$ za neka i in j, za katera velja $0 \leq i < j < t$. To potemtakem pomeni, da nas zaporedje vhodov $a_1, \ldots, a_{i-1}, a_i, \ldots, a_{j-1}$ pripelje v isti par stanj kot zaporedje $a_1, \ldots, a_{i-1}$. Vhode $a_i, \ldots, a_{j-1}$ bi torej lahko preprosto izpustili in potem še vedno enako kot doslej nadaljevali z vhodi $a_j, \ldots, a_t$, pa bi na koncu ravno tako izvedli prehod iz (r_{t-1}, r'_{t-1}) v (r_t, r'_t) in pri tem dobili prav taka izhoda kot doslej, torej b_t in b'_t. Z drugimi besedami, $a_1, \ldots, a_t$ ne bi bilo najkrajše zaporedje, ki privede škatlici f in g do različnih izhodov, saj bi to doseglo že tudi krajše zaporedje $a_1, \ldots, a_{i-1}, a_j, \ldots, a_t$. Mi pa smo t na začetku izbrali tako, da je pomenil dolžino najkrajšega takega zaporedja; tako nas je torej domneva, da je $t \geq 5$, privedla v protislovje.

Zdaj torej vemo, da če sploh obstaja kakšno zaporedje vhodov, pri katerem se izhodi naših škatlic iz M razlikujejo med seboj, je najkrajše tako zaporedje dolgo največ štiri člene. Ker imamo za vsak vhod le dve možnosti, je torej dovolj že, če pregledamo $2^4 = 16$ vhodnih zaporedij dolžine 4 in opazujemo, kakšne izhode bi pri njih dajale škatlice iz M. Če dajo pri vseh teh zaporedjih vse škatlice enak izhod, potem vemo, da se bodo tudi v bodoče obnašale vse enako (in tudi enako kot naša opazovana škatlica). Če pa pri kakšnem od teh vhodnih zaporedij dajejo različne škatlice različna izhodna zaporedja, pošljimo zdaj kakšno od teh vhodnih zaporedij naši opazovani škatlici, pa bomo na podlagi njenih izhodov lahko iz množice M zavrgli nekatere kandidatke, pri katerih se izhodi ne ujemajo z izhodi naše opazovane škatlice. (Poleg zaporedij dolžine 4 lahko seveda pregledamo tudi krajša vhodna zaporedja — mogoče bomo tako identificirali našo črno škatlico s še manj klici.)

$M := \{\,(f, 0) : f \in \{0, \ldots, 255\}\,\}$;
while $|M| > 1$:
  končaj := **true**;
  for $t := 1$ **to** 4:
    za vsako zaporedje $(a_1, \ldots, a_t) \in \{0, 1\}^t$:
      za vsako $(f, q_f) \in M$:
        $q := q_f$;
        for $i := 1$ **to** t:
          $b_i := \beta_f(q, a_i)$; $q := \delta_f(q, a_i)$;
        $b_f := (b_1, \ldots, b_t)$; $q'_f := q$;
      if niso vse dobljene b_f enake: *končaj* := **false**; **break**;
    if not *končaj* **then break**;
  if *končaj* **then break**;
  pošlji opazovani škatlici zaporedje vhodov $a_1, \ldots, a_t$;
  naj bodo $b_1, \ldots, b_t$ njeni izhodi ob teh vhodih;
  $M := \{(f, q'_f) : (f, q_f) \in M \wedge b_f = (b_1, \ldots, b_t)\}$;

Ko se postopek konča, nam v M ostane ena ali več črnih škatlic, ki se bodo v prihodnje vse obnašale enako kot naša opazovana črna škatlica (t.j. se bodo na poljubno zaporedje vhodov odzvale z enakimi izhodi kot opazovana škatlica) in lahko torej katero koli od njih uporabimo za napovedovanje izhodov naše opazovane škatlice.

Izkaže se, da pošlje naš postopek črni škatlici največ 10 (in vsaj 5) vhodnih bitov, preden se jo nauči simulirati. Pri tem je izbira kasnejših vhodnih bitov seveda odvisna od izhodov črne škatlice pri zgodnejših vhodnih bitih. Če pa bi hoteli najti eno samo fiksno zaporedje vhodov, ki bi ga lahko pokazali katerikoli škatlici in na koncu vedeli o njej dovolj, da bi ga simulirali, se izkaže, da potrebujemo vsaj 13 vhodnih bitov; primerno zaporedje je na primer 0001 001 01 1110, obstaja pa še 47 drugih enako dolgih.

(*b*) Opisani postopek lahko posplošimo tudi na funkcije z več vhodi, izhodi in notranjimi stanji, vendar pa stvari hitro postanejo neobvladljive. Če imamo n vhodnih simbolov, m izhodnih simbolov in k notranjih stanj, imamo m^{nk} možnih funkcij β in k^{nk} možnih funkcij δ. To je skupaj $(mk)^{nk}$ različnih črnih škatlic; pri podnalogi (*a*) smo imeli opravka z $n = m = k = 2$ in smo imeli 256 škatlic. Če zdaj dodamo še en bit notranjega stanja, se k poveča na 4 in možnih črnih škatlic je že $8^8 = 2^{24}$ = dobrih 16 milijonov. Tudi zgornja meja za dolžino vhodnih zaporedij, ki jih moramo pregledati v vsaki iteraciji glavne zanke, bi se povečala — iti bi morali do dolžine k^2, pri tej dolžini pa je zdaj možnih že n^{k^2} različnih vhodnih zaporedij. Pri $n = m = 2$ in $k = 4$ bi to pomenilo, da moramo pregledati do 2^{16} zaporedij na množici do 2^{24} črnih škatlic; to je pravzaprav že neobvladljivo, reši nas lahko le še upanje, da bomo na začetku, ko je množica M še velika, našli primerno zaporedje vhodov že pri kakšnem majhnem t, ne šele pri k^2, kasneje pa, ko bodo v M mogoče res ostale same enakovredne škatlice (torej take, ki bi v prihodnosti zagotovo dajale enake izhode), bo ta množica že dovolj majhna, da ne bo pretežko na vseh škatlicah iz nje preizkusiti vseh n^{k^2} vhodnih zaporedij.

Lahko pa to rešitev še malo predelamo in se tako izognemo skrbem glede neobvladljive časovne zahtevnosti pri $n = m = 2$, $k = 4$. Mislimo si dve škatlici iz množice

M, recimo f (v stanju q_f) in g v stanju q_g. Definirajmo graf, v katerem bodo točke vsi pari $(q, q') \in V := \{0, \ldots, k-1\}^2$, povezave pa naj bodo takšne: za vsak $a \in \{0, \ldots, n-1\}$ in vsak $(q, q') \in V$, za katerega je $\beta_f(q, a) = \beta_g(q', a)$, ustanovimo povezavo od (q, q') do $(\delta_f(q, a), \delta_g(q, a))$, na to povezavo pa kot oznako napišimo število a. Naj bo zdaj U množica vseh tistih parov stanj, v katerih lahko črni škatlici s primernim vhodnim simbolom pripravimo do tega, da izpišega dva različna izhodna simbola; torej $U = \{(q, q') \in V : \exists a \in \{0, \ldots, n-1\} : \beta_f(q, a) \neq \beta_g(q', a)\}$. Naš graf ima torej k^2 točk in največ nk^2 povezav; v njem lahko v času $O(nk^2)$ preverimo (npr. z iskanjem v širino), če je iz točke (q_f, q_g) dosegljiva kakšna točka $(q, q') \in U$.

Če je kakšna taka točka dosegljiva, odčitajmo oznake povezav na poti od (q_f, q_g) do (q, q') in dodajmo na konec tega zaporedja še kakšen tak simbol a, pri katerem je $\beta_f(q, a) \neq \beta_g(q', a)$ (tak a gotovo obstaja, saj je $(q, q') \in U$). Dobili smo neko zaporedje vhodnih simbolov, na katero bi se škatlici f in g odzvali z različnima zaporedjema izhodnih simbolov. Zdaj lahko torej pošljemo to zaporedje vhodnih simbolov naši opazovani črni škatlici, pa bomo na podlagi njenih izhodov lahko vsaj eno od škatlic f in g zavrgli iz množice M.

Če pa ni iz (q_f, q_g) v našem grafu dosegljiva nobena točka iz U, potem vemo, da ne obstaja nobeno zaporedje vhodov, ki bi lahko črni škatlici f in g v nadaljevanju pripravilo do tega, da izpišeta različne izhode; torej ni nobene koristi od tega, da v M hranimo obe — eno od njiju, na primer g, lahko kar zavržemo, potem pa isti postopek spet ponovimo z isto f in kakšno drugo g. Na ta način bomo prej ali slej našli v M dve škatlici, ki se ne obnašata vedno enako (in bomo tudi videli, s kakšnim vhodnim zaporedjem ju lahko ločimo), ali pa bomo M oklestili do te mere, da nam bo v njej ostala le še ena črna škatlica in lahko celoten postopek končamo (ter tisto edino preostalo škatlico uporabimo za napovedovanje izhodov naše opazovane škatlice). Pri naših poskusih smo uspeli s tem postopkom poljubno črno škatlico (pri $n = m = 2$, $k = 4$) prepoznati (oz. izvedeti o njej toliko, da se je dalo potem napovedovati njene izhodne bite) po največ 88 vhodnih bitih (v povprečju pa celo po samo 37,3 vhodnih bitih).

Ena od slabosti naše rešitve je, da mora med delom vzdrževati množico M, ki ima lahko do $(mk)^{nk}$ elementov. Pri $m = n = 2$, $k = 4$ je bilo to še obvladljivo, pri $k = 5$ z nekaj truda tudi še (za vsako od $(mk)^{nk}$ možnih črnih škatlic lahko v treh bitih hranimo podatek o tem, ali je v M ali ne, in če je, v kakšnem stanju je; tako porabimo $3 \cdot 10^{10}$ bitov, kar je približno 3,5 GB), pri $k = 6$ pa ne več. Oglejmo si zdaj še rešitev, ki porabi zelo malo pomnilnika, je pa zato malo počasnejša od zgoraj opisane.

Možne škatlice imejmo, enako kot doslej, oštevilčene s celimi števili od 0 do $(mk)^{nk} - 1$. Med delovanjem bo naš algoritem ves čas vodil podatek o tem, katera je prva škatlica (torej tista z najmanjšo številko; recimo ji f), ki daje pri doslej sestavljenem zaporedju vhodov (recimo mu s) enake izhodne vrednosti kot opazovana črna škatlica. V vsakem koraku potem poskušamo poiskati prvo naslednjo škatlico (recimo ji g), ki se na dosedanjih vhodih tudi ujema s f (in z opazovano škatlico), bi pa lahko v prihodnje kdaj dajala (pri primerno izbranih dodatnih vhodnih simbolih) tudi drugačne izhode kot f. Če take g ne najdemo, potem vemo, da opazovana škatlica ne more biti nič drugega kot f oz. ena od tistih, ki se bodo v bodoče obnašale enako kot f, tako da je naš problem rešen. Če pa tako g najdemo, vzemimo neko

zaporedje dodatnih vhodnih simbolov, pri katerem vračata f in g različni izhodni zaporedji. To zaporedje vhodov (recimo mu t) pokažemo tudi opazovani črni škatlici in njene izhode primerjamo z izhodi škatlic f in g. To nam bo omogočilo, da kot možno kandidatko zavržemo ali f ali g ali pa celo obe.

```
f := 0; q := 0; g := 1; s := prazen niz;
while g < (mk)^{nk}:
  (* Invarianta:
     — s je niz vhodnih simbolov, ki smo jih doslej pokazali opazovani črni
       škatlici; naj bo z x zaporedje izhodnih simbolov, ki jih je ob tem izpisala;
     — škatlice 0, ..., f − 1 vračajo, če jih poženemo z začetnim stanjem 0 na
       vhodnem nizu s, zaporedja izhodov, različna od x;
     — škatlica f, če jo poženemo z začetnim stanjem 0 na vhodnem nizu s,
       konča v stanju q in vrne zaporedje izhodov x;
     — za vsako od škatlic f + 1, ..., g − 1, če jo poženemo z začetnim stanjem 0
       na vhodnem nizu s, velja eno od naslednjega dvojega:
       (1) vrne zaporedje izhodov, različno od x;
       (2) vrne zaporedje izhodov x in na poljubnem dodatnem zaporedju
       vhodov bi se njeni izhodi nadaljevali enako kot pri škatlici f. *)
  while g < (mk)^{nk}:
    primerjaj zaporedji izhodov, ki ju vračata f in g, če ju postavimo
      v stanje 0 in ju poženemo na vhodnem zaporedju s;
    če sta tidve izhodni zaporedji različni:
      naj bo r stanje, v katerem je zdaj škatlica g;
      preveri, če obstaja neko vhodno zaporedje t, za katerega velja, da
        če f poženemo iz stanja q in g poženemo iz stanja r ter obema podamo
        zaporedje vhodov t (dolgo ≤ k^2), bosta njuni izhodni zaporedji različni;
        (to preverimo z iskanjem po grafu, enako kot v prejšnji rešitvi);
      če tako zaporedje t obstaja, si ga zapomnimo in prekinimo
        notranjo zanko while;
    g := g + 1;
  end while;
  if g ≥ (mk)^{nk} then break;
  dodaj t na konec niza s;
  poženi f iz stanja q na vhodnem nizu t (recimo, da izpiše niz x in
    pristane v stanju q');
  poženi g iz stanja r na vhodnem nizu t (recimo, da izpiše niz y in
    pristane v stanju r');
  poženi opazovano črno škatlico na vhodnem nizu t (recimo, da izpiše niz z);
  if x = z then q := q'; g := g + 1; continue;
  if y = z then f := g; q := r'; g := g + 1; continue;
  (* Če pridemo do sem, opazovana škatlica ni niti f niti g.
     Poiščimo naslednjo primerno. *)
  g := g + 1;
  while g < (mk)^{nk}:
    primerjaj zaporedji izhodov, ki ju vračata f in g, če ju postavimo
      v stanje 0 in ju poženemo na vhodnem zaporedju s;
```

```
      if sta tidve zaporedji izhodov različni then g := g + 1; continue;
      nadaljuj izvajanje g-ja še na vhodnem zaporedju t; recimo, da g ob
         tem izpiše nek niz y in pristane v stanju r;
      if y = z then f := g; q := r; break;
      g := g + 1;
   end while;
   if g ≥ (mk)^nk then sporoči napako (nobena izmed vseh možnih
      škatlic se ne obnaša tako kot opazovana);
   g := g + 1;
end while;
```

Na koncu lahko za poljubno nadaljnje zaporedje vhodnih simbolov napovedujemo izhode opazovane črne škatlice tako, da računamo izhode škatlice f (z začetkom v stanju q).

Lepo pri tej rešitvi je, da porabimo zelo malo pomnilnika; edina struktura, ki lahko potencialno postane dolga, je vhodni niz s, vendar je bil v praksi pri naših poskusih tudi ta vedno kratek. Opisani postopek smo uporabili pri $n = m = 2$, $k = 4$ za različne naključno izbrane črne škatlice. Po dobrih 260 000 poskusih je bil najdaljši dobljeni s dolg 76 bitov, v povprečju pa so bili dolgi okoli 39 bitov. Pri naših poskusih je bila tale rešitev približno 70 % počasnejša od tiste, ki si množico M hrani eksplicitno (v tabeli), vendar ni rečeno, da se ne bi dalo s kakšno spremembo v implementaciji te razlike še zmanjšati.

(c) Za škatlico brez vhoda (torej z $n = 1$) bi sicer lahko uporabili kakšnega od postopkov, ki smo ju opisali pri točki (b), vendar je skupno število možnih črnih škatlic še vedno veliko (namreč $(mk)^k$) in bi bila množica M lahko pri malo večjih m in k neobvladljivo velika. Preprosteje in učinkoviteje bo, če si bomo strukturo škatlice predstavljali malo drugače kot doslej. Ker škatlica nima vhoda, sta tako izhod kot naslednje notranje stanje odvisna le od trenutnega notranjega stanja. Na zaporedje stanj, skozi katera se škatlica pri svojem delovanju premika, ne moremo nikakor vplivati, ravno tako pa tudi ne na pripadajoče zaporedje izhodnih simbolov. Recimo, da se škatlica na začetku nahaja v stanju q_0, v prvem koraku se premakne v q_1 (in izpiše izhodni simbol b_1), nato se premakne v q_2 (in izpiše izhodni simbol b_2) in tako naprej. Ker ima le k notranjih stanj, sta med stanji $q_0, \ldots, q_k$ gotovo vsaj dve enaki — recimo, da je $q_i = q_j$ (za i in j pa velja $0 \leq i < j \leq k$), kar pomeni, da škatlica iz q_i sčasoma pride v q_{j-1}, iz tega stanja pa spet v q_i. Torej se odtlej ves čas giblje po tem ciklu in tudi njeni izhodi se temu primerno ponavljajo: škatlica odtlej v neskončnost izpisuje le še $b_{i+1} b_{i+2} \ldots b_j$.

Recimo torej, da našo opazovano črno škatlico pustimo teči k korakov in si zapišemo njen izhodni niz $b_1 b_2 \ldots b_k$. Zdaj že tudi vemo, da se v tem nizu skriva nek podniz $b_{i+1} \ldots b_j$, ki ga bo škatlica od tistega trenutka naprej ponavljala v neskončnost; le tega še ne vemo, kakšna sta i in j. Zaradi omejitve $0 \leq i < j \leq k$ si lahko par (i, j) izberemo le na $k(k + 1)/2$ načinov; med temi $k(k + 1)/2$ črnimi škatlicami moramo poiskati pravo. Pravzaprav se nam tudi z izbiro i-ja ni treba pretirano obremenjevati; če škatlica po prvih i korakih začne ponavljati cikel dolžine j, bo ista škatlica ponavljala cikel dolžine j tudi po prvih k korakih. Na primer, izhod 12345345345 . . . si lahko namesto kot 12(345)(345)(345) . . . razlagamo

kot 1234534(534)(534)..., pa gre za eno in isto zaporedje. V nadaljevanju lahko torej predpostavimo, da je $i = n$, in se ukvarjamo le še z ugotavljanjem dolžine cikla, torej $d = j - i$. Lepo v primerjavi z rešitvijo podnaloge (b) je to, da je število kandidatk tukaj le $O(k)$, ne več eksponentno v odvisnosti od n in k.

Primerjajmo zdaj za škatlici s cikloma dolžine d in d' zaporedje izhodnih simbolov, ki ga izpisujeta, če odmislimo prvih k korakov. Pri prvi škatlici se izhodno zaporedje ponavlja s periodo d, pri drugi pa s periodo d'; obe se torej ponavljata s periodo D, če za D vzamemo najmanjši skupni večkratnik števil d in d'. Če torej dajeta obe škatlici na indeksih od b_n do b_{n+D-1} enake izhodne simbole, vemo, da se bo njun izhod tudi v bodoče ves čas ujemal; torej je za potrebe napovedovanja izpisa vseeno, ali si mislimo, da je naša škatlica enaka prvi ali drugi od njiju.

Ker sta d in d' manjša ali enaka n, bo njun najmanjši skupni večkratnik največ $n(n-1)$. Če hočemo opazovati našo črno škatlico do simbola b_{n+D-1}, jo bomo morali izvajati največ n^2 korakov. Pri vsakem koraku moramo za največ n različnih možnih d-jev preveriti, če se izhod opazovane škatlice še ujema s tem, kar bi škatlica izpisovala, če bi res imela periodo d. Na koncu lahko za napovedovanje izhodov škatlice uporabimo katerega koli od še preostalih d-jev (če pa nam ostane en sam, lahko postopek ustavimo tudi že prej).

za $t := 1, 2, \ldots, n$:
  poženi črno škatlico za še en korak in v b_t shrani simbol, ki ga je izpisala;
$M := \{1, 2, \ldots, n\}$;
za $t := n+1, \ldots, n^2$:
  poženi črno škatlico za še en korak in v b shrani simbol, ki ga je izpisala;
  za vsak $d \in M$:
    $r := (t - n) \bmod d$;
    če je $r = 0$, postavi r na d;
    (* *Če je to pravi d, pomeni, da se podniz $b_{n-d+1} \ldots b_n$ ponavlja v neskončnost. Trenutni znak, $b = b_t$, je v tem primeru enak r-temu znaku tega podniza.* *)
    če $b \neq b_{n-d+r}$, pobriši d iz M;
  če je $|M| \leq 1$, se ustavi;

Na koncu lahko vzamemo poljuben $d \in M$ in napovedujemo izhode škatlice po enaki formuli, s kakršno smo jih pred tem preverjali. Če pa se zgodi, da nam iz množice M izpadejo vsi kandidati, je to znak, da se škatlica ne obnaša tako, kot obljublja naloga; bodisi imamo premajhen k ali pa se obnašanje škatlice spreminja oz. je odvisno še od česa drugega kot od trenutnega notranjega stanja.

Naloge so sestavili: črna škatlica, prepoznavanje ulomka — Andrej Bauer; lomljenje besedila — Matija Grabnar; osem — Miha Grčar; taborniki (I in II) — Klemen Kenda; najdaljši skupni niz — Mojca Miklavec; letalski poleti, tetris — Miha Vuk; nezanesljivi golobi — Klemen Žagar; biblijski kod, kopanje lukenj, izpeljava, kratice, ločevanje slik, nerimska števila, palindromni stavki, začetnice, živi in mrtvi — Janez Brank.

REZULTATI

Tabele na naslednjih straneh 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 vsaki skupini podeli dve prvi, dve drugi in dve tretji nagradi. Letos smo v prvi skupini izjemoma podelili samo eno tretjo nagrado, v tretji skupini pa tri tretje nagrade, ker sta si dva tekmovalca delila šesto mesto z istim številom točk, petouvrščeni pa je bil le eno točko pred njima, zato se je zdelo pošteno, da dobijo nagrado vsi trije.

Letos smo prvič uvedli tudi pohvale, in sicer jih prejmejo tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil nagrade; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 30 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen te novosti je, da izkažemo priznanje in spodbudo vsem, ki se po rezultatu prebijejo v zgornjo polovico svoje skupine. Podobno prakso poznajo tudi na nekaterih mednarodnih tekmovanjih; na primer, na mednarodni računalniški olimpijadi (IOI) prejme medalje kar polovica vseh udeležencev.

V tabelah so prejemniki nagrad označeni z „1“, „2“ in „3“ v prvem stolpcu, prejemniki pohval pa s „P“.

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1	1	Matjaž Leonardis	1	ZRI	20	17	20	18	16	91
1	2	Matevž Mihalič	3	ŠC Novo mesto	19	15	20	18	12	84
2	3	Anže Rožman	4	SŠER Ljubljana	6	9	20	17	12	64
2	4	Gregor Lah	3	ŠC Novo mesto	2	11	18	16	12	59
3	5	Jernej Legiša	3	Licej Fr. Prešeren, Trst	10	2	20	18	8	58
P	6	Grega Kamenšek	3	ŠC Celje, SŠEK	7	15	13	16	2	53
P	7	Sebastian Kraševec	3	SŠER Ljubljana	19	9	3	20	1	52
P	8	Ludvik Zobec	3	Licej Fr. Prešeren, Trst	8	10	14	18	1	51
P	9	Blaž Kovačič	2	SERŠ Maribor	10	0	20	18	0	48
P	10	Darjan Govednik	4	SŠER Ljubljana	10	7	14	12	4	47
P	10	Blaž Kostanjšek	2	SŠER Ljubljana	15	6	9	16	1	47
P	12	Tomo Markočič	3	Sr. teh. šola Koper	0	16	5	16	8	45
P	13	Pavel Kos	1	Gimnazija Bežigrad	13	0	8	16	2	39
P	13	Matej Šnajder	3	SERŠ Maribor	17	1	1	18	2	39
P	13	Anže Žitnik	3	SŠER Ljubljana	0	17	5	10	7	39
P	16	Gregor Cimerman	3	SŠER Ljubljana	8	6	4	17	3	38
P	17	Klemen Kastelic	3	ŠC Novo mesto	2	1	18	15	0	36
P	18	David Jesenko	3	ŠC Celje, SŠEK	15	1	0	17	2	35
P	18	Miha Rataj	2	ŠC Celje, SŠEK	5	8	7	11	4	35
	20	Aljaž Božič	1	Gimnazija Bežigrad	5	15	5	1	8	34
	21	Dejan Erjavec	2	ŠC Novo mesto	8	10	0	12	3	33
	22	Klemen Forstnerič	2	SERŠ Maribor	14	0	3	14	0	31
	23	Anton Zvonko Gazvoda	3	ŠC Novo mesto	8	5	0	16	0	29
	24	Miha Javoršek	3	SŠER Ljubljana	15	10	0	0	3	28
	25	Črtomir Majer	2	SERŠ Maribor	0	0	12	14	1	27
	26	Jure Vengušt	3	ŠC Celje, SŠEK	0	0	0	18	5	23
	27	Samo Pahor	1	Gimnazija Kranj	0	0	2	18	1	21
	28	Blaž Papič	3	ŠC Novo mesto	18	0	0	2	0	20
	29	Denis Trstenjak	2	SPTŠ Murska Sobota	0	6	9	2	2	19
	30	Gregor Grgurič	2	SPTŠ Murska Sobota	8	1	0	2	0	11
	31	Patrik Huber	2	SPTŠ Murska Sobota	0	0	0	10	0	10
	32	Marko Senčar	3	SERŠ Maribor	2	1	0	2	2	7
	32	Marko Slavec	4	Gimnazija Kranj	0	0	0	5	2	7
	34	Jan Jug	1	Gimnazija Bežigrad	2	1	0	0	3	6
	35	Filip Kralj	1	Gimnazija Kranj	0	2	0	1	2	5
	36	Mojca Rozmarič	2	SPTŠ Murska Sobota	1	1	0	0	1	3
	36	Egej Vecelj	1	Gimnazija Kranj	0	0	3	0	0	3
	38	Marko Kavaš	2	SPTŠ Murska Sobota	0	1	0	0	0	1
	38	Miha Rožac	3	Sr. teh. šola Koper	0	0	0	0	1	1

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	∑
1	1	Žiga Ham	2	ZRI + SŠER Ljubljana	17	17	20	18	20	92
1	2	Peter Koželj	2	ZRI + Gim. Bežigrad	13	20	20	17	20	90
2	3	Matjaž Drolc	1	Gimnazija Litija	19	20	20	17	10	86
2	4	Rok Kralj	2	Gimnazija Vič	20	2	20	16	20	78
3	5	Jošt Stergar	2	Gimnazija Bežigrad	17	20	20	0	15	72
3	6	Marko Zabreznik	3	ŠC Velenje, PTERŠ	13	8	12	12	19	64
P	7	Tadej Gorenc	4	ŠC Novo mesto	13	20	10	0	19	62
P	8	Tomaž Treven	3	Škof. klas. gimn. Ljubljana	18	1	13	7	17	56
P	9	Matija Rezar	2	Gimnazija Kranj	18	1	19	17	0	55
P	10	Matjaž Lovše	4	ŠC Novo mesto	18	15	11	0	10	54
P	11	Erik Plestenjak	3	SŠER Ljubljana	18	12	16	1	5	52
P	11	Rok Slamek	3	SPTŠ Murska Sobota	9	20	18	0	5	52
P	13	Mitja Rešek	3	ŠC Celje, Gimnazija Lava	13	4	9	15	7	48
P	13	Andrej Slapnik	3	ŠC Velenje, PTERŠ	13	1	14	10	10	48
P	15	Tomaž Turner	3	SPTŠ Murska Sobota	18	1	0	0	19	38
	16	Matjaž Payrits	3	Gimnazija Bežigrad	14	2	8	0	9	33
	16	David Vidmar	4	ŠC Novo mesto	13	20	0	0	0	33
	18	Sandi Srkoč	3	SPTŠ Murska Sobota	13	3	11	0	1	28
	19	Dejan Knez	3	ŠC Celje, Gimnazija Lava	3	1	4	13	5	26
	19	Leon Pajk	4	ŠC Novo mesto	6	0	16	1	3	26
	21	Miha Jerič	4	SPTŠ Murska Sobota	13	1	7	1	0	22
	21	Patrick Zver	4	SPTŠ Murska Sobota	5	0	11	1	5	22
	23	Blaž Berglez	4	ŠC Celje, SŠEK	19	1	0	0	0	20
	24	Robi Cvirn	3	ŠC Celje, SŠEK	0	4	0	15	0	19
	25	Matic Černač	3	SŠER Ljubljana	5	7	0	3	0	15
	26	Zoran Felbar	3	SPTŠ Murska Sobota	0	8	5	0	0	13
	27	Aleš Jenič	4	ŠC Novo mesto	0	1	0	0	10	11
	27	Matjaž Trček	3	SŠER Ljubljana	8	0	2	1	0	11
	29	Aleksander Kozlar	4	SPTŠ Murska Sobota	3	1	2	4	0	10
	29	Roman Šuster	4	SPTŠ Murska Sobota	0	7	1	0	2	10
	31	Mišel Keser	4	ŠC Celje, SŠEK	0	0	0	0	0	0

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj)					
					1	2	3	4	5	$\sum$
1	1	Blaž Peterlin	4	ŠC Novo mesto	97	87	40		30	254
1	2	Jan Berčič	4	ZRI		100	30	78	30	238
2	3	Domen Blenkuš	4	ZRI	21	90	50			161
2	4	Nace Hudobivnik	3	Škof. klas. gimn. Ljubljana		54	7			61
3	5	Gašper Černevšek	4	SŠER Ljubljana		51				51
3	6	Matej Drame	4	ŠC Celje, Gimnazija Lava	10			20	20	50
3	6	Miha Lunar	4	SŠER Ljubljana	0	40	10			50
	8	Žan Doberšek	1	I. gimnazija Celje		40		0	0	40
	9	Gašper Mlinar	4	SŠER Ljubljana		0	37	0	0	37
	9	Alen Stanko	4	ŠC Celje, SŠEK	37					37
	11	Armin Djogić	4	SŠER Ljubljana					20	20
	12	Aljaž Čeru	4	SŠER Ljubljana			7			7
	12	Davor Gaberšek	4	ŠC Celje, SŠEK	0			7		7
	14	Marko Balkovec	4	ŠC Novo mesto	0					0
	14	Jaka Hudoklin	4	Gimnazija Ledina	0	0				0
	14	Primož Korošec	4	SŠER Ljubljana	0					0
	14	Matej Pinter	4	ŠC Celje, SŠEK	0					0

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Matjaž Leonardis	8 GB iPod nano B. Greene: *Čudovito vesolje*
1	1	Matevž Mihalič	750 GB zunanji disk B. Greene: *Čudovito vesolje*
1	2	Anže Rožman	500 GB zunanji disk B. Greene: *Čudovito vesolje*
1	2	Gregor Lah	predvajalnik DVD + VCR D. Bodanis: $E = mc^2$
1	3	Jernej Legiša	predvajalnik MP3 D. Bodanis: $E = mc^2$
2	1	Žiga Ham	22" monitor J. R. Weeks: *Oblika prostora* Wilson, Watkins: *Uvod v teorijo grafov*
2	1	Peter Koželj	8 GB iPod nano V. Omladič: *Matematika in odločanje* J. Strnad: *Mala kvantna fizika*
2	2	Matjaž Drolc	750 GB zunanji disk V. Omladič: *Matematika in odločanje* J. Strnad: *Mala kvantna fizika*
2	2	Rok Kralj	750 GB zunanji disk B. Greene: *Tkanina vesolja*
2	3	Jošt Stergar	4 GB iPod nano B. Greene: *Tkanina vesolja*
2	3	Marko Zabreznik	1 GB iPod shuffle B. Greene: *Tkanina vesolja*
3	1	Blaž Peterlin	80 GB iPod classic J. R. Weeks: *Oblika prostora* Wilson, Watkins: *Uvod v teorijo grafov*
3	1	Jan Berčič	8 GB iPod nano J. R. Weeks: *Oblika prostora* Wilson, Watkins: *Uvod v teorijo grafov*
3	2	Domen Blenkuš	8 GB iPod nano J. R. Weeks: *Oblika prostora* Wilson, Watkins: *Uvod v teorijo grafov*
3	2	Nace Hudobivnik	750 GB zunanji disk D. Bodanis: *Električno vesolje*
3	3	Gašper Černevšek	750 GB zunanji disk D. Bodanis: *Električno vesolje*
3	3	Matej Drame	1 GB iPod shuffle V. Omladič: *Matematika in odločanje* J. Strnad: *Mala kvantna fizika*
3	3	Miha Lunar	1 GB iPod shuffle V. Omladič: *Matematika in odločanje* J. Strnad: *Mala kvantna fizika*
Tekmovanje programov — Štiri v vrsto			
7×6		Primož Bajželj	predvajalnik MP3
20×20		Rok Kralj	1 GB iPod shuffle I. Vidav: *Števila in matematične teorije*
Off-line naloga — Kino			
		Tomaž Hočevar	USB rocket launcher
		Rok Kralj	slušalke + B. Greene: *Čudovito vesolje*

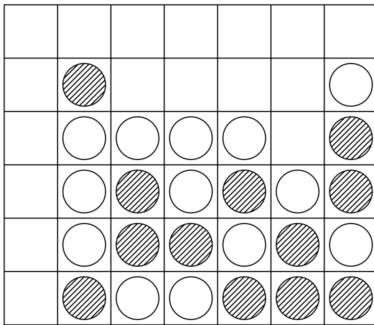
Poleg tega je vsak od nagrajencev prejel tudi izvod knjige *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004* (v dveh zvezkih, IJS, 2006).

SODELUJOČE ŠOLE IN MENTORJI

Šola	Mentorji
Gimnazija Bežigrad, Ljubljana	Andrej Šuštaršič
Gimnazija Kranj	Matevž Jekovec
Gimnazija Ledina	Gregor Anželj
Gimnazija Litija	Jan Maver
Gimnazija Vič	
Srednja elektro-računalniška šola Maribor (SERŠ)	Vida Motaln, Slavko Nekrep, Manja Sovič Potisk
Srednja poklicna in tehniška šola (SPTŠ) Murska Sobota	Simon Horvat, Karel Maček, Milan Petrijan
Srednja tehniška šola Koper	Jernej Maučec
Srednja šola za elektrotehniko in računalništvo (SŠER), Ljubljana	Matjaž Kodela, Nataša Makarovič, Darjan Toth
Šolski center Celje, Srednja šola za elektrotehniko in kemijo (SŠEK)	Damjan Luc, Boštjan Resinovič
Šolski center Celje, Splošna in strokovna gimnazija Lava	Borut Slemenšek
Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija	Mile Božič, Tomaž Ferbežar, Tanja Jerič, Aljaž Štrucelj, Albert Zorko
Šolski center Velenje, Poklicna in tehniška elektro in računalniška šola (PTERŠ)	Damjan Borovnik, Miran Zevnik
Škofijska klasična gimnazija, Ljubljana	Helena Medvešek
Zavod za računalniško izobraževanje (ZRI), Ljubljana	Jelko Urbančič
Znanstveni licej France Prešeren, Trst	Valentina Busechian

TEKMOVANJE PROGRAMOV — ŠTIRI V VRSTO

Podobno kot lani in predlani smo tudi letos organizirali tekmovanje programov. Okvirni opis naloge smo objavili novembra 2007 skupaj z razpisom za tekmovanje v znanju, podrobnosti pa v začetku januarja 2008. Tekmovalci so imeli čas do 21. marca 2007 (teden dni pred tekmovanjem), da pošljejo svoje programe. Letošnja naloga je od tekmovalcev zahtevala, da napišejo nadzorno logiko za igranje priljubljene igre „štiri v vrsto“. Igra poteka na igralnem polju z w stolpci, v vsakem stolpcu pa je prostora za h žetonov. Igralca izmenično mečeta žetone (prvi ima modre, drugi pa rdeče) v igralno polje; pri vsaki potezi si igralec izbere, v kateri stolpec bo dodal svoj žeton, nato pa žeton pristane na vrhu ostalih žetonov v tem stolpcu. Zmaga igralec, ki prvi doseže, da stojijo štirje njegovi žetoni skupaj v vrsti, stolpcu ali po diagonali. Če se po $w \cdot h$ potezah zapolni celo igralno polje, ne da bi kateri od igralcev dosegel štiri v vrsto, je igra neodločena.



Primer možnega stanja igre pri igralnem polju velikosti 7×6. Igralec z belimi žetoni je bil prvi na potezi in je pravkar v svoji 13. potezi dosegel štiri v vrsto in s tem zmagal.

Tekmovanje programov smo izvedli v dveh težavnostnih kategorijah, ki se razlikujeta po velikosti igralne površine: tradicionalno polje velikosti 7×6 (sedem stolpcev, šest vrstic) in večje polje 20×20. Vsak tekmovalec je moral napisati podprogram za krmiljenje igralca; le-ta je ob vsakem koraku dobil podatke o stanju mreže (kje so žetoni in kakšne barve so), ob inicializaciji pa dobi tekmovalčev podprogram tudi podatke o višini in širini igralnega polja, 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 120 sekund procesorskega časa (na 2-gigahercnem opteronu) za igro na mreži 7×6 oz. do 600 sekund za igro na mreži 20×20.

Po roku za oddajo rešitev smo izvedli navzkrižno tekmovanje: za vsak par igralcev smo izvedli več iger (potek igre namreč ni nujno determinističen, saj si lahko igralci pri izbiri potez pomagajo tudi z generatorji psevdonaključnih števil), in sicer 30 iger na polju 7×6 in 20 iger na polju 20×20. Zmagovalec igre dobi $+1$ točko, poraženec -1 točko, v primeru neodločene igre pa oba dobita 0 točk; na koncu izračunamo povprečno število točk vsakega tekmovalca. Nato izvedemo še eno tako skupino iger, pri čemer pa je zdaj prvi na potezi tisti igralec, ki je bil prej drugi, in obratno; pri igri štiri v vrsto je namreč tisti, ki je prvi na potezi, v boljšem položaju. Na opisani način se vsak tekmovalec pomeri z vsemi ostalimi tekmovalci in pri vsakem dobi neko povprečno število točk med -1 in $+1$; na koncu tako dobljene točke

za vsakega tekmovalca seštejemo. Če je tekmovalcev n, je igral vsak tekmovalec proti $n-1$ drugim tekmovalcem, enkrat kot prvi na potezi in enkrat kot drugi, tako da bi lahko, če bi vedno zmagal, dobil največ $2(n-1)$ točk; če bi vse igre izgubil, pa bi dobil $-2(n-1)$ točk.

Rezultati

Dobili smo prispevke sedmih tekmovalcev (šestih dijakov in enega študenta). Naslednji tabeli kažeta povprečno število točk, ki jih je dobil posamezni tekmovalec v vseh odigranih igrah (torej z vsemi ostalimi tekmovalci).

Standardno polje 7×6

Mesto	Ime	Šola	Točke
1	Tomaž Hočevar	FRI	6,35
2	Primož Bajželj	TŠC Kranj	5,40
3	Rok Kralj	Gimnazija Vič	2,60
4	Žiga Ham	ZRI	−1,80
5	Gašper Černevšek	SŠER Ljubljana	−4,91
6	Blaž Kostanjšek	SŠER Ljubljana	−7,64

Večje polje 20×20

Mesto	Ime	Šola	Točke
1	Tomaž Hočevar	FRI	9,11
2	Rok Kralj	Gimnazija Vič	3,83
3	Žiga Ham	ZRI	−0,26
4	Gašper Černevšek	SŠER Ljubljana	−2,93
5	Jan Berčič	ZRI	−3,58
6	Blaž Kostanjšek	SŠER Ljubljana	−6,18

Rešitev

Za mrežo 7×6 obstaja optimalna strategija, s katero lahko prvi igralec zanesljivo zmaga; drugi igralec pa lahko s to strategijo vsaj izsili neodločen rezultat, če prvi igralec svojega prvega žetona ni odvrgel ravno v srednji stolpec polja. Odkril jo je Victor Allis leta 1988,[15] najbolj znana implementacija te strategije (z nekaj izboljšavami) pa je program Velena, ki ga je napisal Giuliano Bertoletti; na njej temelji na primer program *gnect*, klon igre štiri v vrsto v okviru projekta GNOME.

Vendar pa ta strategija ni tako preprosta in z implementacijo ne bi bilo malo dela, poleg tega pa je tudi ni enostavno posplošiti na igralno polje 20×20. Preprostejša pot do neoptimalne, vendar vseeno precej dobre strategije pa je, da z rekurzijo preiščemo možna nadaljevanja igre nekaj potez vnaprej. Več časa ko smo pripravljeni porabiti, več potez naprej lahko gledamo; v vsakem stanju je možnih največ w nadaljevanj, torej je po k potezah možno priti do največ w^k stanj igre. Pri $w = 7$ si lahko privoščimo gledati vsaj kakšnih šest potez naprej, pri $w = 20$ pa vsaj štiri. Vsako od stanj po k potezah pa moramo nekako oceniti, da bomo imeli občutek za to, kako verjetno je, da bi igro iz tistega stanja uspeli pripeljati do naše zmage; recimo tej oceni $f(s)$ za stanje s.

[15] Victor Allis: *A knowledge-based approach of Connect-Four.* Masters Thesis, Dept. of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1988.

Za f si lahko izmislimo neko hevristično funkcijo, na primer takole. Mogoče ima v tistem stanju eden od igralcev že štiri v vrsto; v tem primeru je tisto stanje že samo po sebi zmagovalno za enega od njiju in lahko vzamemo $f(s) = \infty$ (če je stanje zmagovalno za nas) oz. $f(s) = -\infty$ (če je stanje zmagovalno za nasprotnika). Drugače pa lahko pregledamo vse možne skupine štirih celic, ki bi lahko tvorile štiri v vrsto; če vsebuje taka celica za zdaj le žetone ene barve (in eno ali več praznih celic), pripišemo v mislih temu stanju neko število točk (pozitivno, če so že obstoječi žetoni v tej skupini naši, in negativno, če so žetoni nasprotnikovi; absolutna vrednost pa naj bo večja za skupine z več žetoni. Za $f(s)$ na koncu vzamemo vsoto tako dobljenih točk.

Ocene, ki nam jih vrača funkcija f, lahko uporabimo za izbiro potez po načelu min-max. Naj bo $N(s)$ množica stanj, v katera se da priti iz s z eno potezo. Stanje, v katerem je na potezi nasprotnik, lahko ocenimo tako, da nasprotnika ne podcenjujemo in raje domnevamo, da bo naredil potezo, ki je za nas najbolj neugodna; stanje, v katerem smo na potezi mi, pa ocenimo tako, da si izberemo med možnimi nadaljevanji tisto, ki je za nas najbolj ugodno:

$$\hat{f}(s) = \begin{cases} f(s) & \text{če je stanje } k \text{ potez naprej od trenutnega} \\ & \text{ali pa je zmagovalno za enega od igralcev} \\ \min\{\hat{f}(s') : s' \in N(s)\} & \text{če je na potezi nasprotnik} \\ \max\{\hat{f}(s') : s' \in N(s)\} & \text{če smo na potezi mi} \end{cases}$$

Ko se moramo odločiti, katero potezo naj naredimo v nekem stanju s, izračunamo $\hat{f}(s)$ po formuli $\max\{\hat{f}(s') : s' \in N(s)\}$ in pogledamo, pri katerem s' je bil ta maksimum dosežen; potem predlagamo tisto potezo, ki iz stanja s naredi s'.

Opisani pristop temelji na rešitvi, ki jo je poslal Tomaž Hočevar, podobne ideje pa je uporabilo tudi več drugih tekmovalcev.

OFF-LINE NALOGA — KINO

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. Podobno kot lani smo tudi letos razpisali off-line nalogo 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.

Okvirni opis naloge smo objavili v novembru 2007 skupaj z razpisom tekmovanja v znanju računalništva; podroben opis, ocenjevalni program in vhodne podatke pa smo objavili v začetku januarja 2008. Tekmovalci so imeli možnost oddajati svoje rešitve prek naše spletne strani vse do vključno 28. marca 2008, torej do dneva pred tekmovanjem. Naš računalnik je sproti preverjal pravilnost vsake oddane rešitve in prikazoval razvrstitev tekmovalcev glede na kakovost prejetih rešitev.

Opis naloge

Kinematografi, ki v nekem manjšem mestecu upravljajo s kinom, te prosijo za pomoč pri sestavljanju filmskih sporedov. Njihov kino ima eno samo dvorano in obratuje cel dan in to vse dni v letu. Leto ima 366 dni, znano pa je tudi število sedežev v dvorani; največ toliko ljudi lahko torej hkrati gleda neko predstavo.

Kinematografi si želijo, da bi prodali čimveč vstopnic, zato so nedavno izvedli med prebivalci svojega mesteca podrobno anketo. Zdaj točno poznamo število prebivalcev in za vsakega prebivalca vemo tudi to, kdaj (ob kateri uri) najraje zahaja v kino in katere žanre filmov ima rad.

Dan je tudi seznam filmov, ki jih v našem kinematografu smemo predvajati. Za vsak film je znano njegovo trajanje (v minutah), ocena priljubljenosti (ki vpliva na to, kako verjetno je, da ga bodo ljudje prišli gledat) in to, katerim žanrom pripada. Glede tega, kolikokrat (in kdaj) smemo predvajati posamezni film, ni posebnih omejitev — lahko ga predvajamo enkrat, večkrat, nobenkrat, lahko tudi večkrat v istem dnevu ipd.

Tvoja naloga je sestaviti celoletni spored predstav za ta kinematograf. Pri tem lahko uporabljaš le filme z danega seznama. Za vsako predstavo moraš navesti številko filma ter dan in uro začetka predstave. Ker ima kino eno samo dvorano, se predstave med seboj ne smejo prekrivati. Poleg tega tudi ne sme nobena predstava segati v naslednje leto (npr. če bi začeli nek film, ki traja več kot eno uro, predvajati na 366. dan ob enajstih zvečer).

Testni primeri

Pripravili smo deset testnih primerov. Podatke o prebivalcih smo zgenerirali naključno, podatke o filmih pa smo pobrali s spletne strani IMDB (Internet Movie Database, www.imdb.com). Za osnovo smo vzeli vse filme iz let 2004–2007; v IMDB je iz tega obdobja 78 238 filmov, le za 21 822 izmed njih pa so prisotni vsi podatki, ki jih za našo nalogo potrebujemo (žanri, trajanje in popularnost). Deset testnih primerov smo pripravili tako, da smo skombinirali različne podmnožice teh 21 822 filmov in različno velike množice naključno generiranih prebivalcev:

Testni primer	Število in izbor filmov	Št. prebivalcev	Št. sedežev
1	100 najbolj priljubljenih	10	5
2	1000 najdaljših	100	20
3	1000 najmanj priljubljenih	1000	100
4	1000 naključno izbranih	5000	200
5	1000 najbolj priljubljenih	10000	300
6	vseh 21822	10000	300
7	5000 naključno izbranih	15000	400
8	vseh 21822	15000	400
9	10000 naključno izbranih	20000	500
10	vseh 21822	20000	500

Ocenjevanje in točkovanje

Sporede, ki so jih pošiljali tekmovalci našemu ocenjevalnemu računalniku, smo ocenjevali s programom, ki je simuliral obnašanje gledalcev. Za vsako predstavo se vsak prebivalec našega kraja z neko verjetnostjo odloči, ali bo šel kupit karto ali ne. Verjetnost nakupa je odvisna med drugim od ocene filma, od tega, ali pripada žanru, ki ga gledalec rad gleda, in od tega, ali gledalcu ustreza ura, ob kateri se film predvaja. Za oceno sporeda se vzame povprečno število prodanih kart v treh ponovitvah simulacije.

Tekmovalec lahko pri posameznem testnem primeru odda več sporedov, upošteva pa se najboljši med njimi. Pri vsakem testnem primeru smo razvrstili tekmovalce po številu prodanih kart, nato pa je prvi tekmovalec dobil 10 točk, drugi 9, tretji 8 in četrti 7. Na koncu smo za vsakega tekmovalca sešteli njegove točke po vseh desetih testnih primerih.

Postopek simulacije, ki smo ga uporabili za ocenjevanje posameznih sporedov, je naslednji:

Pregleduj predstave v sporedu po naraščajočem dnevu in uri začetka.
Pri vsaki predstavi izvedi naslednje:
  Ponovi za vsakega prebivalca:
    Naj bo P verjetnost, da bi šel ta gledalec kupit vstopnico za to predstavo.
    Z verjetnostjo P se odločimo, da bo šel res kupit vstopnico zanjo;
      z verjetnostjo $1 - P$ pa, da je ne bo šel.
  Če smo se za nakup vstopnice odločili pri več kot s gledalcih
    (pri čemer je s število sedežev v kino dvorani),
    obdržimo izmed njih naključno množico s gledalcev

(pri čemer imajo vsi enake možnosti, da bodo izbrani).
Izbrani gledalci zdaj kupijo vstopnice za to predstavo.
Na koncu izpiši skupno število prodanih vstopnic.

Ker postopek uporablja psevdonaključna števila, smo ga pognali trikrat in za oceno vzeli povprečno število prodanih vstopnic.[16]

Verjetnost, da gre nek gledalec kupit vstopnico za neko predstavo, smo izračunali po naslednji formuli:

$$P = P_{\check{c}as} \cdot P_{\check{z}anri} \cdot P_{prej} \cdot P_{\check{z}e} \cdot P_{ocena}.$$

Pri tem so posamezni členi določeni takole:

- $P_{\check{c}as}$ je odvisna od časa predstave. Za vsakega gledalca vemo, ob katerem času v dnevu je najraje v kinu. Poglejmo razdaljo med tem časom in časom, v katerem poteka predstava. Na primer, če je nekdo v kinu najrajši ob 02:00, neka predstava pa poteka od 23:00 do 00:45, je razdalja v tem primeru 75 minut (od 00:45 do 02:00). Za nekoga, ki je v kinu najrajši ob 22:30, pa bi bila razdalja v tem primeru le 30 minut (od 22:30 do 23:00). Za nekoga, ki je v kinu najrajši ob npr. 23:15, bi bila razdalja v tem primeru 0, saj predstava pokriva tudi njegov najljubši čas. Če zdaj tako opisano razdaljo (v minutah) označimo z r, bomo vzeli

  $$P_{\check{c}as} = 1/(1 + e^{0{,}05 \cdot (r-90)}).$$

  To na primer pomeni, da pri $r = 90$ minutah verjetnost $P_{\check{c}as}$ pade na 50 %; pri 120 minutah že na samo 18 %; pri 180 minutah pa že na samo 1 %.

- $P_{\check{z}anri}$ je odvisna od tega, koliko izmed tistih žanrov, ki jim pripada film, je takih, da jih opazovani gledalec rad gleda. Če ni nobenega takega žanra, vzamemo $P_{\check{z}anri} = 0{,}1$; če je tak žanr eden, vzamemo $P_{\check{z}anri} = 0{,}9$; če pa sta taka žanra dva ali več, vzamemo $P_{\check{z}anri} = 1$.

- P_{prej} je odvisna od tega, koliko časa je minilo, odkar si je ta gledalec nazadnje ogledal kakšno predstavo v našem kinu. Če se naša predstava začne na dan d_1, on pa si je nazadnje pred tem ogledal neko predstavo, ki se je začela na dan d_2, naj bo $r = d_1 - d_2$. Potem vzamemo $P_{prej} = 1 - 0{,}8^r$. To na primer pomeni, da po sedmih dneh verjetnost P_{prej} naraste na 79 %, po štirinajstih dneh pa že na 96 %. (Za gledalca, ki ni bil še nikoli v kinu, je $P_{prej} = 1$.)

- $P_{\check{z}e}$ je odvisna od tega, kolikokrat je ta gledalec doslej že videl ta film. Če ga ni videl še nikoli, vzamemo $P_{\check{z}e} = 1$. Če pa ga je videl že k-krat (za nek $k > 0$), vzamemo $P_{\check{z}e} = 0{,}1 \cdot 0{,}5^{k-1}$.

- P_{ocena} je odvisna od ocene filma; ocene filma so realna števila med 1 in 10. Če je ocena nekega filma enaka r, vzamemo $P_{ocena} = g(r)/g(9)$, pri čemer je $g(r)$ definirana takole:

[16] Izkazalo se je, da razlike v številu prodanih vstopnic od ene simulacije do druge niti niso velike. Večje število simulacij bi bilo že neugodno, ker je pri zelo dolgih sporedih na največjih testnih primerih ena simulacija trajala dobre tri minute. Tekmovalec je moral torej na oceno svojega sporeda že zdaj včasih čakati skoraj deset minut, če pa bi hoteli izvesti več simulacij, bi moral čakati še dlje.

r	1	2	3	4	5	7	8	9
$g(r)$	50	100	160	170	190	210	300	350

Med temi vrednostmi je g linearna funkcija. Pri $r < 1$ vzamemo $g(r) = g(1)$, pri $r > 9$ pa $g(r) = g(9)$. (Vrednosti v tej tabeli smo dobili na podlagi povprečnega izkupička od prodaje vstopnic, ki ga za nekatere filme navajajo na IMDB.)

Na naši spletni strani smo objavili tudi izvorno kodo programa, s katerim smo izvajali simulacije in ocenjevali prejete sporede.

Rezultati

Na tekmovanju so sodelovali trije dijaki in en študent. Pokazali so obilno mero tekmovalnega duha. Prve sporede smo prejeli že v prvem dnevu po objavi testnih podatkov; kasneje so pri mnogih testnih primerih tekmovalci še večkrat oddajali nove sporede in izboljševali svoje rezultate; pravi finiš pa so uprizorili še v zadnjih dneh tekmovanja, med drugim so v zadnji uri (28. marca med enajsto zvečer in polnočjo) poslali še deset sporedov.

Končna razvrstitev je naslednja:

Tomaž Hočevar (FRI)	94 točk
Rok Kralj (Gimnazija Vič)	86 točk
Domen Blenkuš (ZRI)	69 točk
Jan Berčič (ZRI)	19 točk

Rešitev

Pri tej nalogi pravzaprav rešujemo nek precej obsežen optimizacijski problem. V prostoru vseh možnih celoletnih sporedov (ki je resnično ogromen) moramo poiskati nek čim boljši spored, pri čemer je ocena sporeda določena z zgoraj opisanim postopkom, ki simulira prodajo vstopnic. Pri takšnih problemih ne bi bilo razumno pričakovati, da bomo našli postopek, ki nas bo zanesljivo in učinkovito vodil do najboljše možne rešitve; je pa takšna naloga načeloma primerna za številne bolj ali manj hevristične postopke in prijeme, ki jih poznamo s področja optimizacijskih metod in operacijskih raziskav: lokalna optimizacija, iskanje s tabujem, simulirano ohlajanje, genetski algoritmi, iskanje v snopu (*beam search*), optimizacija z roji (*swarms*) ipd. Paziti pa moramo, da izbrani postopek ne bo porabil nesprejemljivo veliko časa; ker lahko, kot smo videli zgoraj, ocenjevanje posameznega sporeda traja tudi več minut, si ne moremo privoščiti kakšnega preveč temeljitega preiskovanja prostora, pri katerem bi morali oceniti veliko število možnih sporedov. V tem razdelku bomo opisali preprost požrešni algoritem, ki daje precej dobre rezultate (pri vsakem od desetih testnih primerov proda več vstopnic kot sporedi, ki so jih poslali naši tekmovalci), poraba časa pa je ravno še nekako sprejemljiva.

Spored sestavljajmo lepo po vrsti, od prvega januarja proti enaintridesetemu decembru. Nove predstave bomo le dodajali na konec sporeda, nikoli pa ne bomo spreminjali že sestavljenega dela sporeda. Med predstavami ne bomo puščali časovnih vrzeli, ustavili pa se bomo šele, ko nam bo do konca leta ostalo tako malo časa, da vanj ne moremo stlačiti nobene predstave več. Ostane nam le še vprašanje,

kateri film izbrati na vsakem koraku. Pomagamo si lahko s podobnim postopkom simulacije, kot smo ga opisovali že na str. 128. Recimo, da smo si že izbrali film, ki bi ga dodali na konec sporeda; potem preglejmo vse gledalce, pri vsakem izračunajmo verjetnost, da bi šel kupit vstopnico za to predstavo tega filma, in nato s pomočjo generatorja psevdonaključnih števil določimo število prodanih vstopnic. Ko na ta način za vsak film ocenimo, koliko vstopnic bi prodali, če bi zdajle na konec sporeda dodali predstavo tega filma, se lahko s pomočjo teh podatkov odločimo, kateri film bi res dodali na konec sporeda. Zdaj lahko še enkrat odsimuliramo, kateri prebivalci kupijo vstopnice za to predstavo; o vsakem prebivalcu moramo namreč hraniti podatke o tem, kdaj je bil nazadnje v kinu in katere filme je že videl (in kolikokrat), saj jih bomo potrebovali pri računanju verjetnosti nakupa vstopnice pri kasnejših predstavah. Postopek je torej tak:

naj bo S prazen spored;
za vsakega gledalca g si zapomnimo, da ni bil še nikoli v kinu;
ponavljaj:
  za vsak film f:
    če je f predolg, da bi ga lahko dodali na konec sporeda S, ga preskoči;
    $k_f := 0$;
    za vsakega gledalca g:
      naj bo P verjetnost, da bi g kupil vstopnico za ogled
        predstave filma f, če bi to predstavo dodali na konec sporeda S;
      z verjetnostjo P povečajmo k_f za 1;
    $k_f := \min\{k_f, s\}$ (če je s število sedežev v dvorani);
    zdaj vemo, da če bi dodali na konec S-ja predstavo filma f,
      bi prodali pri tej predstavi približno k_f vstopnic;
  če so bili vsi filmi predolgi, končaj;
  sicer izberi najugodnejši film f in ga dodaj na konec sporeda S; ($\star$)
  s simulacijo se odloči, kateri prebivalci so kupili vstopnico za to predstavo;
  zanje popravi podatek o tem, kdaj so bili nazadnje v kinu,
    in si zapomni, da so že videli film f;
na koncu vrni spored S;

Dogovoriti se moramo še, za kateri film naj se odločimo v vrstici ($\star$). Najbolj očitna zamisel je gotovo ta, da si izberemo film, ki je prodal največ vstopnic, torej tistega z največjo vrednostjo k_f. Izkaže pa se, da je v podatkih z IMDBja precej zelo kratkih „filmov“, dolgih le po nekaj minut; kar 651 filmov je dolgih 5 minut ali manj, še nadaljnjih 1568 filmov pa je dolgih od 6 do 10 minut. Ker ocenjevanje pri naši nalogi upošteva le število prodanih vstopnic, je načeloma čisto vseeno, ali prodamo določeno število vstopnic pri petminutnem „filmu“ ali pa pri 90- ali 120-minutnem celovečercu; smo pa pri kratkem filmu na boljšem zaradi tega, ker bomo lahko v eno leto stlačili tem več predstav, čim krajše filme bomo predvajali. Če torej na primer nek film traja 20-krat dlje kot drugi, bi morali pri njem tudi prodati nekajkrat več vstopnic, da bi se ga splačalo uvrstiti na spored. Izkaže se, da se to ponavadi ne dogaja. Z drugimi besedami, ponavadi prodamo več vstopnic, če predvajamo 20 zaporednih 5-minutnih filmov (in je dvorana ves čas skoraj prazna), kot če predvajamo eno predstavo nekega 100-minutnega filma. Zato je pametno v vrstici ($\star$) dodati na konec sporeda tisti film, ki ima najugodnejše razmerje med

številom prodanih vstopnic k_f in trajanjem filma (recimo d_f), torej največjo vrednost k_f/d_f. Pri nekaterih izmed naših testnih primerov se je, ko smo začeli v vrstici ($\star$) namesto k_f maksimizirati razmerje k_f/d_f, število prodanih vstopnic več kot potrojilo. Dobljeni sporedi so predvajali skoraj same kratke filme in imeli v celem letu več kot 200 000 predstav (povprečna dolžina filma je bila torej približno 2,6 minute).

Kot smo omenili že zgoraj, je glavna težava tega sicer preprostega postopka njegova velika časovna potratnost. Zunanja zanka naredi toliko ponovitev, kolikor je predstav v končnem sporedu (recimo 200 000), znotraj nje pa imamo še dve gnezdeni zanki po vseh filmih (ki jih je lahko čez 20 000) in vseh gledalcih (ki jih je tudi lahko do 20 000). V vsak iteraciji najbolj notranje zanke pa moramo računati verjetnost po formulah s str. 129, s čimer je tudi nekaj dela. Na največjem testnem primeru je porabil gornji postopek približno 12 dni procesorskega časa (na računalniku z 2-GHz opteronom), na vseh desetih testnih primerih skupaj pa približno 38 dni. (Za primerjavo: tekmovalci so imeli od objave podrobnega opisa naloge in vhodnih podatkov približno 85 dni časa za pripravo in oddajo svojih sporedov.)

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 138–145.

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 bi znal v programu uporabiti 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 bi znal v programu uporabiti 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
Iskanje v širino (po grafu)	☐ da	☐ ne
Dinamično programiranje	☐ da	☐ ne

[če misliš, da to pomeni uporabo new, GetMem, malloc ipd., potem obkroži „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: ____________________

Ali poznaš zapis z velikim O za časovno zahtevnost algoritmov?
[npr. $O(n^2)$, $O(n \log n)$ ipd.] ☐ da ☐ ne

[Le pri 1. in 2. skupini.] V besedilu nalog trenutno objavljamo deklaracije tipov in podprogramov v pascalu, C/C++ in pythonu.
— Ali razumeš kakšnega od teh jezikov dovolj dobro, da razumeš te deklaracije v besedilu naših nalog? ☐ da ☐ ne
— So ti prišle deklaracije v pythonu kaj prav? ☐ 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++, 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++: «, ») ☐ ☐ ☐
[če pišeš v C++] razred map iz STLa ☐ ☐ ☐
[če pišeš v C++] razred priority_queue iz STLa ☐ ☐ ☐

[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?

____

____

Ali veš, da bo naslednjo soboto, 5. aprila 2008, na FRI potekalo 32. državno srednješolsko tekmovanje iz računalništva? ☐ da ☐ ne
Ali se ga nameravaš udeležiti? ☐ da ☐ ne
Zakaj / zakaj ne? ______________________

____

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 38 tekmovalcev prve skupine, 27 tekmovalcev druge in 16 tekmovalcev tretje. Letošnja anketa je bila zelo podobna lanski, nekaj manjših sprememb je bilo le pri vprašanjih o poznavanju podatkovnih struktur in raznih programskih konstruktov, dodali pa smo tudi vprašanje, ali tekmovalci vedo za državno tekmovanje v znanju računalništva (ki je bilo teden dni po našem).

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. 139. 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 so se zdele tekmovalcem v vseh skupinah naloge še kar težke. 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: 3,57 v prvi skupini (lani 3,18), 3,62 v drugi skupini (lani 3,36) in 3,74 v tretji (lani 3,72). Komisija sicer pri pripravi nalog ni imela občutka, da so naloge letos kaj težje kot ponavadi (niti ni bil njen namen, da bi bile).

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), je nekaj korelacije, vendar je šibka ($R^2 = 0{,}44$).

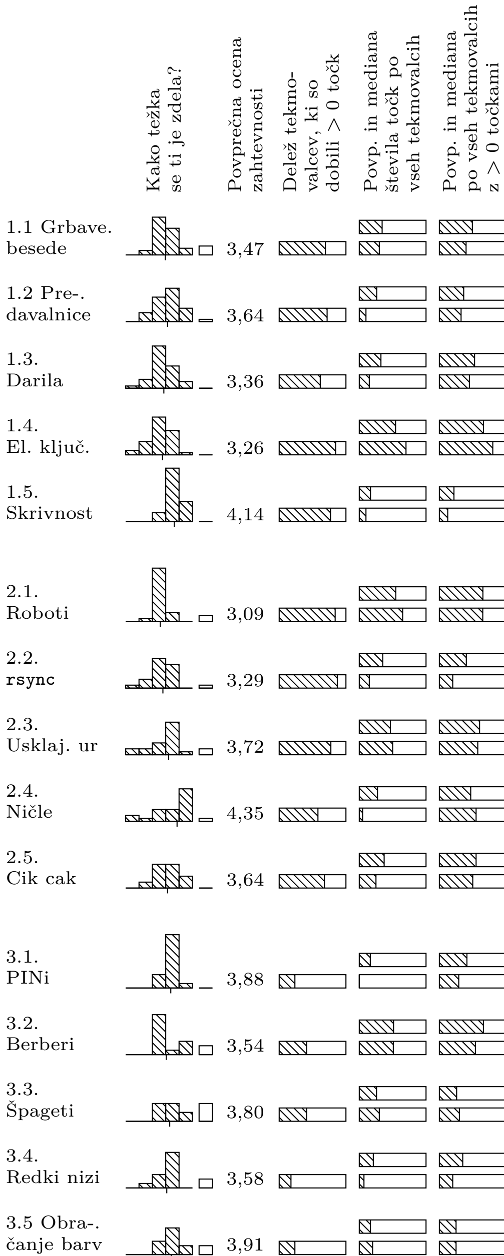
Največ pripomb, češ da je težka (ali celo pretežka), je dobila naloga 2.4 (društvo ljubiteljev ničel); to je najbrž zato, ker ta naloga zahteva poznavanje operacij na bitih, teh pa mnogi tekmovalci ne poznajo preveč dobro. Težka se jim je zdela tudi naloga 1.5 (skrivnost), kar je pričakovano, saj je bila ta tudi v resnici mišljena kot težja naloga v 1. skupini. Presenetilo pa nas je, da so nalogo 3.1 (PINi) šteli za eno od najtežjih v 3. skupini, saj se je nam zdelo, da bo to ena od najlažjih nalog v njej.

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 140. Nad razumljivostjo besedil ni veliko pripomb, razen pri nalogi 2.3 (usklajevanje ur), ki se je večini ljudi zdela težko razumljiva ali celo nerazumljiva. Razmeroma precej pripomb, da je besedilo težko razumljivo, je tudi pri nalogah 1.5 (skrivnost) in 2.4 (društvo ljubiteljev ničel). Težko je reči, ali je težava v besedilu samem ali pa v tem, da gre v teh primerih bodisi za naloge, ki so malo drugačnega tipa od večine ostalih (npr. 2.3, ki je „realnočasovna“ naloga), ali pa so preprosto malo težje od ostalih v svoji skupini (npr. 1.5, pa tudi 2.4, ker mnogi tekmovalci ne poznajo operacij na bitih).

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, le pri Berberih (3.2) je nekaj več pripomb, češ da je besedilo predolgo. Res je, da je malo daljše kot pri drugih nalogah.

Naloge se jim večinoma zdijo zanimive; še največ pripomb, češ da so dolgočasne, je bilo v prvi skupini, še posebej pri nalogi 1.5 (skrivnost).

Pripomb, da bi naloga vzela preveč časa, večinoma ni bilo veliko; največ jih je bilo v tretji skupini, še posebej pri nalogi PINi (3.1), kar nas je presenetilo, ker se



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

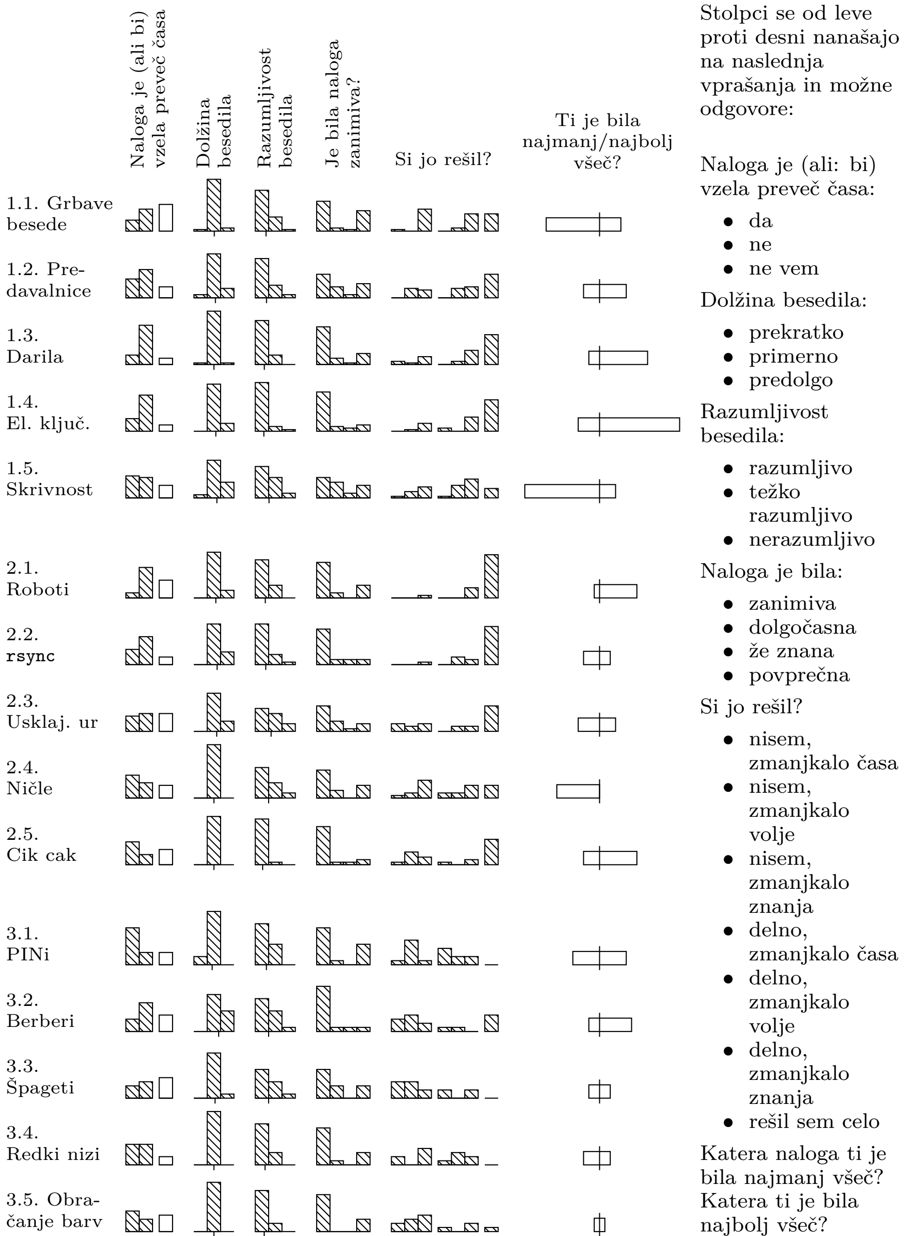
Pomen stolpcev v vsaki vrstici:

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.



nam ta naloga res ni zdela tako zamudna. Nekaj pripomb, da so prezamudne, je bilo tudi pri nalogah skrivnost (1.5), društvo ljubiteljev ničel (2.4) in cik cak (2.5).

Glasovi o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, letos niso tako razpršeni kot prejšnja leta; tokrat so se večinoma precej jasno izoblikovale bolj in manj priljubljene naloge. V prvi skupini je bila najbolj priljubljena naloga elektronska ključavnica (1.4), v drugi cik cak (2.5) in v tretji Berberi (3.2). Najbolj nepriljubljene pa so skrivnost (1.5), GrbaveBesede (1.1) in društvo ljubiteljev ničel (2.4). Zanimiv pojav je nastopil v tretji skupini, ko je vsako naloga razen Berberov dobila skoraj enako število glasov tekmovalcev, ki jim je bila najbolj všeč, in tisth, ki jim je bila najmanj všeč.

Programersko znanje, algoritmi in podatkovne strukture

Ko sestavljamo naloge, še posebej tiste za prvo skupino, nas pogosto skrbi, če tekmovalci poznajo ta ali oni jezikovni konstrukt, programerski prijem, algoritem ali podatkovno strukturo. Zato jih v anketah zadnjih nekaj let sprašujemo, če te reči poznajo in bi jih znali uporabiti v svojih programih.

	Prva skupina	Druga skupina	Tretja skupina
priority_queue v C++	3 %	9 %	0 %
map v C++	3 %	14 %	0 %
zamikanje s shl, shr	22 %	58 %	50 %
operatorji na bitih	58 %	48 %	63 %
strukture	29 %	79 %	93 %
naštevni tipi	21 %	54 %	47 %
gnezdenje zank	76 %	88 %	94 %
zanka while	95 %	100 %	100 %
zanka for	95 %	100 %	100 %
kazalci	21 %	48 %	69 %
rekurzija	21 %	69 %	81 %
podprogrami	55 %	88 %	88 %
več-d tabele (array)	47 %	71 %	75 %
2-d tabele (array)	66 %	100 %	94 %
1-d tabele (array)	79 %	100 %	100 %
delo z datotekami	32 %	92 %	100 %
std. vhod/izhod	71 %	92 %	100 %

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.

Rezultati pri vprašanjih o programerskem znanju so podobni lanskim, le tekmovalci v prvi skupini letos pravijo, da znajo manj, kot so o sebi napisali v lanski anketi tekmovalci takratne prve skupine. Stvari, ki jih poznajo slabše, so približno iste kot lani in predlani: rekurzija, kazalci, naštevni tipi in operatorji na bitih.

Podobno kot lani je poznavanje nekaterih stvari v tretji skupini ni nič boljše kot v prvi (ali pa je celo malo slabše), kar je po svoje presenetljivo. Primer tega so npr. drevesa in Evklidov algoritem.

Vprašanje o algoritmih in podatkovnih strukturah smo letos malo preoblikovali in namesto „ali poznaš naslednje algoritme?“ vprašali „ali bi znal v programu uporabiti

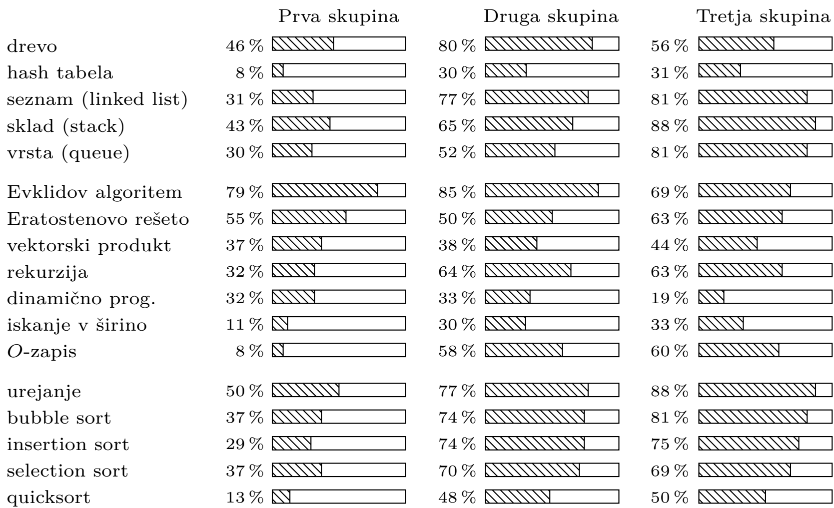


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

---

naslednje algoritme?“. Upali smo, da bodo rezultati zato bolje odražali to, kaj lahko na tekmovanju dejansko pričakujemo od tekmovalcev. Najbrž je posledica te spremembe tudi dejstvo, da je v povprečju na ta vprašanja odgovarjalo pritrdilno malo manj tekmovalcev kot lani, niso pa te razlike zelo velike. Lepo je, da precej tekmovalcev pozna drevesa (tudi v 1. skupini), pa tudi poznavanje iskanja v širino je višje kot lani. Podobno kot lani pa smo bili tudi malo razočarani nad nizkim deležem tekmovalcev, ki poznajo razpršene tabele (celo v 3. skupini jih je le 31 %).

Ena sprememba v primerjavi z lansko anketo je tudi vprašanje o dinamičnem programiranju. Lani je bil delež tekmovalcev, ki so trdili, da ga poznajo, nepričakovano visok, kar smo si razlagali s tem, da nekateri tekmovalci vprašanje mogoče razumejo narobe in si mislijo, da se nanaša na uporabo dinamične alokacije pomnilnika (malloc, free ipd.). Zato smo v letošnji anketi posebej napisali, da nimamo v mislih tega. Mogoče je ravno zato letos delež tekmovalcev, ki pravijo, da znajo uporabljati dinamično programiranje, veliko manjši kot lani (npr. le 19 % v tretji skupini, lani pa 70 %). Verjetno so letošnje številke bolj realistične.

Pri sestavljanju nalog za 3. skupino zna biti včasih neugodno, ker ponujajo različni programski jeziki v svojih standardnih knjižnicah precej različne nabore algoritmov in podatkovnih struktur. Tekmovalec, ki npr. piše v C-ju, ima pri roki učinkovit algoritem za urejanje (funkcija qsort), uporabnik pascala pa tega nima. V C++u imamo na voljo tudi standardne razrede za binarno iskalno drevo (map) in kopico (priority_queue), v javi in C# pa na primer za razpršeno tabelo (Hashtable, Dictionary). Radi bi se, kolikor je to pač mogoče, izogibali nalogam, pri katerih takšni algoritmi in podatkovne strukture iz standardnih knjižnic tekmovalcu bistveno olajšajo delo (bodisi ker mu prihranijo čas ali pa mu celo dajo na voljo neko funkcionalnost, ki je sam sploh ne bi znal napisati) in mu tako na nek način dajo nepošteno prednost pred tekmovalci, ki delajo v drugih jezikih. Zato smo letos dodali v anketo vprašanji, ali

znajo tekmovalci uporabljati dve konkretni podatkovni strukturi iz standardne knjižnice jezika C++: `map` (binarno iskalno drevo) in `priority_queue` (prioritetna vrsta); letos bi prišla kakšna od teh dveh podatkovnih struktur prav pri nalogi s špageti (3.3). Izkazalo pa se je, da ju zna uporabljati zelo malo tekmovalcev.

Novo vprašanje v letošnji anketi je bilo tudi, ali tekmovalec pozna zapis z velikim O, ki pride prav pri opisovanju časovne in prostorske zahtevnosti algoritmov. V drugi in tretji skupini je kar dobro znan (pozna ga približno 60 % tekmovalcev).

Uporaba programskih jezikov

Pascal je skoraj povsem odmrl — letos ga je uporabljalo le pet tekmovalcev, je pa res, da so ti večinoma dosegali zelo dobre uvrstitve (zmagovalec v prvi skupini, drugo- in petouvrščeni v drugi skupini in tretje- in četrtouvrščeni v tretji skupini). Tudi v anketi je delež ljudi, ki so med jeziki, ki jih dobro poznajo, navedli tudi pascal, veliko manjši kot lani.

Velika večina tekmovalcev uporablja C in C++, v prvi skupini pa je precej tudi uporabnikov jave in pythona (slednjega v prejšnjih letih ni uporabljal nihče). Zanimivo je, da v tretji skupini jave ni uporabljal nihče (kot že tudi lani ne), čeprav je tam podprta. Pač pa smo letos v tretjo skupino kot novost uvedli podporo za C#, kar se je dobro obneslo, saj so v njem pisali trije uspešni tekmovalci (zmagovalec ter peto- in šestouvrščeni).

V anketi je precej tekmovalcev napisalo, da dobro poznajo tudi PHP, vendar sta ga na tekmovanju uporabljala le dva.

	Leto in skupina												
	2008			2007			2006			2004			2003
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	3
pascal	$1\frac{1}{2}$	2	2	$8\frac{1}{2}$	2	1	6	5	5	23	20	13	17
C	$4\frac{1}{2}$	11	$2\frac{1}{2}$	$5\frac{1}{2}$	11	$6\frac{1}{2}$	4	16	$1\frac{1}{2}$	13	$7\frac{1}{2}$	1	4
C++	$17\frac{1}{2}$	11	$9\frac{1}{2}$	7	14	$15\frac{1}{2}$	13	5	$10\frac{1}{2}$	5		6	5
java	$9\frac{1}{2}$	3			$2\frac{1}{2}$				3		$\frac{1}{2}$		–
PHP		2	–	1		–	1		–			–	–
basic			–		1	–		1	–			–	–
C#			3		$\frac{1}{2}$	–			–			–	–
python	6	1	–			–			–			–	–
nič		1		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. Znak „–“ označuje jezike, ki se jih tisto leto v tretji skupini ni dalo uporabljati.

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, C/C++ in letos po novem tudi v pythonu. Tekmovalce smo v anketi vprašali, če te deklaracije razumejo ali pa bi morale biti še v

kakšnem drugem jeziku; veliki večini sedanje deklaracije zadostujejo (35/38 v prvi skupini in 25/26 v drugi). Veliko pa jih je predlagalo tudi deklaracije v javi.

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 (26/36 v prvi skupini, 23/26 v drugi in 14/16 v tretji). Deset ljudi je predlagalo, da bi bile rešitve tudi v C++, dvanajst v javi, osem pa v pythonu. Še vedno je tudi nekaj presenetljivih primerov (letos štirje), ko ljudje pravijo, 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?

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. Letos v tretji skupini precej bolj kot lani prevladujejo dijaki četrtega letnika.

Skupina	Št. tekmovalcev po letnikih				Povprečni letnik
	1	2	3	4	
prva	7	11	18	3	2,4
druga	1	5	14	11	3,1
tretja	1		1	15	3,8

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 večina navaja našo stran (`rtk.ijs.si`), nekateri pa tudi portal Slo-Tech (`www.slo-tech.com`), ki je prijazno objavil novico o našem tekmovanju in reklamno pasico s povezavo na našo spletno stran. Delež tekmovalcev, ki so za tekmovanje izvedeli kako drugače kot prek mentorjev, je letos bistveno manjši kot lani.

Pri vprašanju, kje so se naučili programirati, je v primerjavi z lanskimi (in predlanskimi) odgovori zanimiva razlika ta, da je letos delež samoukov manjši, precej pa je zdaj tekmovalcev, ki so se programirati naučili predvsem v šoli. Samouki prevladujejo le v tretji skupini.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. Precej je tudi tekmovalcev, ki želijo manj nalog in/ali več časa (še posebej v tretji skupini).

Iz odgovorov na vprašanje, kakšne potekmovalne dejavnosti bi jih zanimale, je težko zaključiti kaj posebej konkretnega.

Novost v letošnji anketi je bilo tudi vprašanje o 32. državnem srednješolskem tekmovanju v znanju računalništva, ki je potekalo 5. aprila, torej teden dni za našim tekmovanjem. Letos je za obe tekmovanji začetni izbor nalog pripravila enotna komisija, zato nas je zanimalo, kakšen je presek udeležencev obeh tekmovanj. V anketi smo vprašali, ali tekmovalci vedo za državno tekmovanje in ali se ga imajo

Skupina	Kje si izvedel za tekmovanje				Kje si se naučil programirati					Čas reševanja			Število nalog			Potekmovalne dejavnosti							
	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	34	2	2	0	19	24	4	1	2	3	5	29	3	14	19	12	13	8	14	13	10	17	18
II	20	2	2	3	13	19	1	1	0	7	2	10	0	5	13	9	5	7	4	5	6	7	9
III	12	4	0	0	12	7	3	0	1	7	1	7	1	8	5	3	2	2	2	7	5	3	5

namen udeležiti. Izkaže se, da večina anketiranih tekmovalcev ve tudi za to drugo tekmovanje (20/28 v prvi skupini, 12/14 v drugi in 7/8 v tretji) in večina tistih, ki zanj vedo, se ga namerava tudi udeležiti (10 iz prve, 9 iz druge in 6 iz tretje skupine). Tisti, ki se ga ne bodo udeležili, kot razlog večinoma navajajo pomanjkanje časa in druge obveznosti. Med tistimi, ki se ga bodo udeležili, pa je bilo tudi nekaj zanimivih razlogov tipa „ker me je profesor/mentor določil/prijavil“.

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb. Letos se je, podobno kot že v nekaterih prejšnjih letih, spet pojavilo nekaj predlogov, da bi tekmovalci v prvi in drugi skupini pisali odgovore z računalnikom namesto na papir.

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 ima spremenljivko, ki šteje grbe v besedi; no, to je čisto smiselno, cvetka je v tem, da to spremenljivko imenuje „števnik“ namesto „števec“.

(1.1) Opomba nekega tekmovalca pod rahlo negotovim pascalskim programom:

> Moja rešitev temelji na predpostavki, da se `c` avtomatično dviguje, in da sem pravilno razumel uporabo `ReadLn` in `WriteLn`, saj drugače programiram v javi in mi je ta programski jezik do sedaj bil neznan.

Zanimivo vprašanje je, ali si to bolj zasluži nagrado za pogum (ker se loti na tekmovanju programirati v jeziku, ki ga sploh ne pozna) ali za cagavost (ker ne upa vprašati, ali sme programirati v javi). Drugo nalogo (1.2) je šel potem vendarle reševat v javi. Mi pa moramo prihodnje leto v navodilih za tekmovalce očitno bolj jasno povedati, da lahko tekmovalci načeloma pišejo v katerem koli programskem jeziku, ki ga kdo od članov komisije razume.

(1.1) Letos je bilo primerov mazohizma v izvorni kodi manj kot lani. Tole je vendarle eden od njih:

```
if (A[x] == "a" || A[x] == "b" || A[x] == "c" || ..... || A[x] == z) {
```

Na srečo vsaj ni eksplicitno napisal vseh 26 pogojev, ampak je sam uporabil tiste pike za opuščeni del izraza.

(1.1) Nekaj za ljubitelje Emersona:[17]

```
char a[30]; /* Predpostavimo, da ima vsaka beseda manj kot 32 črk. */
```

(1.2) Zelo upravičen komentar na začetku neke rešitve:

```
(Program bi bil sicer zelo zamuden.)
⋮
st = 0
while st <= len(zz):
   vz = zz[st] + vz
   vk = vk + kk[st]
```

Hja, to bi res znalo trajati malce dlje. . .

(1.3) Takole je nek tekmovalec prilagodil deklaracijo podprograma, za katerega naloga pravi, da je že podan:

```
def Darilo(1, 2): ... # to sem spremenil v številke zato ker so ljudje številke
```

Zadnja beseda komentarja je sicer malo nečitljiva; mogoče je mislil kaj drugega?

(1.4) Spodnji blok kode je navkljub herojskemu trudu za las zgrešil letošnjo nagrado za najglobljo indentacijo (zmagovalca imamo pri nalogi 3.4, str. 151), zasluži pa si jo za najbolj nenavadno narečje pythona:

[17] “A foolish consistency is the hobgoblin of little minds.”—R. W. Emerson.

```
def Odkleni(): . . .
def PreberiTipko(): . . .
while 1:
   geslo = ’. . . .’
   import PreberiTipko()
   if PreberiTipko() == Preklici:
   elif PreberiTipko() == a:
      import PreberiTipko()
         if PreberiTipko() == Preklici:
         elif PreberiTipko() == b:
            import PreberiTipko()
               if PreberiTipko() == Preklici:
               elif PreberiTipko() == c:
                  import PreberiTipko()
                     if PreberiTipko() == Preklici:
                     elif 1000 * a + 100 * b + 10 * c + PreberiTipko() == geslo:
                        import Odkleni()
```

(1.4) Eksotičen, toda čisto pravilen način, kako doseči neskončno zanko:

```
for (i = 0; i < 3; i++) { // neskončna zanka
   i = 0;
   ⋮
}
```

(V preostanku zanke spremenljivke i seveda ne spreminja.)

(1.4) Še en pristop k neskončni zanki:

```
boolean a = true;
⋮
while (a) { . . . }
```

a-ja nikoli ne spreminja, tako da je zanka res neskončna. Ampak zakaj ne napiše kar **while** (**true**)?. . .

(1.4) Rešitev za ljubitelje represivnih organov:

```
if (vtipkano == geslo) {
   vtipkano = ""; stevec = 0; Odkleni(); } else {
   vtipkano = ""; stevec = 0; Policija(); }
```

(1.5) En tekmovalec je predlagal neke vrste iskanje v širino po grafu klicev, pri čemer pa se je malo zapletal pri ustavitvenem pogoju:

> Ta postopek ponovimo ~~maksimalno 13-krat in če še vedno ne najdemo stika, ga verjetno tudi ni, kajti tokrat smo morali pogledati že~~ tolikokrat, dokler ne najdemo stika.

(1.5) Iz opisa postopka pri eni od rešitev:

> Za preverjanje, koga so klicali in od koga so bili klicani, bi bilo dobro uporabiti zanko in funkcije, saj se postopek ponavlja.

(1.5) Še en primer ustvarjalnega pristopa k ustavitvenim pogojem zank:

> Če upoštevamo dejstvo, da se pri igri telefončki ponavadi konec in začetek ne ujemata, nam zanke verjetno ne bi bilo treba ponavljati prav velikokrat.

(1.5) Iz ene od rešitev:

> Ustvariti si moramo nek seznam (po možnosti velik), na katerega bomo shranjevali potencialne skrivnostneže.

(1.5) Le čemu bi se človek obremenjeval s podrobnostmi:

> Tole bi se dalo izboljšati z nekaj if stavki, da bi prihranili na prostoru in času.

(2.1) Nagrado za predrznost dobi tale komentar na koncu opisa rešitve, ki pri mreži $n \times n$ porabi $O(n^2)$ pomnilnika:

> Menim, da bi se ta postopek odlično odrezal v velikih mrežah, saj koordinate vsakega robota pregledujemo samo enkrat.

Naloga pa je tekmovalce spodbujala k razmišljanju o velikih mrežah ravno z namenom, da tekmovalci ne bi pisali rešitev, ki porabijo $O(n^2)$ pomnilnika...

(2.1) Simpatično preimenovanje stavka **break**:

```
brake 2;
```

(Drugače pa je to primer multilevel break statementa, ki ga podpira PHP — lepo!)

(2.1) Komentar na koncu ene od rešitev:

> Ta metoda bi delovala dokaj hitro, še posebej v primeru, če so podatki shranjeni v bazo.

Hm, ja, naloga pravi, da so podatki v seznamu, on pa bi jih zdaj natihoma preložil v bazo in zgradil nad njimi kopico indeksov :)

(2.3) Eden od tekmovalcev je v naslovu svoje rešitve prekrstil to nalogo v „Usklajevanje ur“.

(2.4) Nagrada za boj proti zapostavljanju stavka else:

```
if (max <= r)
   max = r;
else
   max = max;
```

(2.4) Rešitev nekega tekmovalca najprej izračuna napačen rezultat, nazadnje pa ga še dokončno dotolče takole:

```
   ⋮
   c = c ^ c; // besedo, ki bo zxorala bite tako, da bo
   return c;  // največ bitov = 0, zxoramo samo s
}             // sabo in jo vrnemo
```

(2.4) Nek tekmovalec je napisal rešitev, ki vrne kar povprečno vrednost vseh števil v bloku, nato pa je dodal še komentar:

> Povprečna vrednost bi bila za nekatere primere najboljša, za nekatere pa ne. Za primere, kjer bi povprečna vrednost odstopala, bi lahko uporabili varianco ali modus.

Očitno je ljubitelj statistike, še zlasti modusa, saj ga je omenjal tudi v svoji rešitvi naloge (2.1).

(2.4) Pošten komentar:

> Ni optimalno, ampak deluje pa menda :)

Pa še res je — deluje pravilno, ampak zelo neučinkovito; porabi $O(n2^b)$ časa za n besed s po b biti.

(2.4) Izviren pogled na operacijo xor:

```
Blok[i] := Blok[i] xor 0; ⟵ število, katerim ugasnemo vse bite
```

(2.5) Iz ene od rešitev:

> Oblika, ki ni popolna in je na začetku, jo najprej izpišemo, nato pa najvišja in srednja oblika, nato pa zadnja oblika.

(2.5) Kaj, če sploh kaj, je ta reševalec mislil?

```
for (int i = 0; i < dolzina; i + 2)
{
   for (int j = 0; j < dolzina; j++)
   {
      if (niz[i] == '/') // preverimo, če je v nizu na indeksu i znak /
      {
         if (niz[i + 1] == '\')
         {
            cout << "....." << niz[i] << niz[i + 1] << "...." << niz[i];
            cout << endl;
         }
         else
         {
            cout << niz[i + 1] << niz[i] << "ooooooo"<< niz[i + 1] << niz[i];
            cout << "oo" << endl;
         }
      }
   }
}
```

(2.5) Nekateri tekmovalci imajo pesniško žilico:

```
// izriše celo stvar, in to je to za ta denar
```

(3.1) Podobno kot lani se je tudi letos dogajalo, da so tekmovalci oddali programe, ki se sploh ne prevedejo. Spodnji primer se na primer ne prevede zato, ker je ifstream::eof metoda, ne polje, zato pri klicu eof manjkajo oklepaji. Toda tudi če bi jih dodali, program ne bi deloval pravilno, saj v zanki ne bere iz datoteke (ki si jo je pred tem pravilno odprl), pač pa kar s standardnega vhoda (z getc)...

```
ifstream beri("pini.in");
:
while (! beri.eof)
{
   getc(b[i]);
   i++;
}
```

Naslednja stvar, ki jo je ta tekmovalec oddal, je bil ta isti program, *popolnoma nespremenjen*, dobre pol ure po prvi oddaji.

(3.1) Zanimivo ime za funkcijo, ki računa potence števila 10:

```
int kratdeset(int steviloN)
{
   int a = 1, b = 1;
   while (a < steviloN)
   {
      b *= 10;
      a++;
   }
   return b;
}
```

(3.1) Za tem tekmovalcem je bila očitno naporna seansa debuggiranja :)

```
vsota = vsota + temp % 10; // KONČNO DELA
:
int zadna = vsota % 10; // DELA
:
} // po 4 urah prva delujoča zadeva
```

Program sicer vedno izpiše le dve števili, od katerih je prvo pravilen odgovor za podnalogo (a), drugo število pa je vedno kar $3 \cdot n$, karkoli naj bi to že pač pomenilo. . .

(3.1) Hvalevredna odkritosrčnost:

```
//                                   ne da se mi |
fo.WriteLine("{0}\n{1}\n{2}", binc, num, 12);
```

Je pa škoda, da se mu ni dalo — rezultata pri podnalogah (a) in (b) sta pravilna (razen tega, da je pri (b) pozabil pregledati PIN, ki ga sestavljajo same devetice).

(3.2) Očitno pozna tale tekmovalec še kakšnega Berbera, za katerega mi ne vemo:

```
for (y = razkolna_linija − 1; y >= 0; y−−) // Yukahumov zaselki
```

(3.2) Namesto 2-d tabele znakov si je tale tekmovalec predstavil zemljevid kot 1-d tabelo nizov. In ker nizov v C# ne moremo spreminjati, je spreminjanje te tabele precej draga zadeva:

```
static string[] map;
:
map[p.y] = map[p.y].Substring(0, p.x) + (yus ? "Y" : "M") + map[p.y].Substring(p.x + 1);
```

Spomnimo se še, da so ti nizi lahko dolgi do 3000 znakov. Au.

(3.4) Nek tekmovalec si je najprej pripravil celoten niz s eksplicitno v neki tabeli, dolgi 10^6 znakov. S tem ni nič narobe — se je pač odločil, da bo implementiral preprosto rešitev, ki bi mu prinesla 50 % točk. Toda potem je iskanje niza p v nizu s implementiral takole:

```
if (dolzina == 6)
  { for (i = 0; i < strlen(zacetni); i++)
      { if (zacetni[i] == iskani[j])
          if (zacetni[i+1] == iskani[j+1])
            if (zacetni[i+2] == iskani[j+2])
              if (zacetni[i+3] == iskani[j+3])
                if (zacetni[i+4] == iskani[j+4])
                  if (zacetni[i+5] == iskani[j+5])
                  {
                    printf("nasli smo p v nizu\n");
                    stevec++;
                  }
      }
  }
```

To je za primere, ko je p dolg 6 znakov. Podobne bloke je imel še za druge dolžine od 2 do 5, le da z manj gnezdenimi **if**-i. Če je p daljši od 6 znakov, pa program izpiše 0. Škoda — že če bi uporabil dobro staro funkcijo strstr iz standardne knjižnice, bi zlahka (in s precej manj dela) dobil pravilen rezultat za poljuben p. Vsekakor pa dobi letošnjo nagrado za najglobljo indentacijo.

(3.4) Res temeljito odpiranje datoteke:

```
for i := 1 to n do

Assign(txt, 'nizi.out'); Rewrite(txt); WriteLn(txt, count); Close(txt);
```

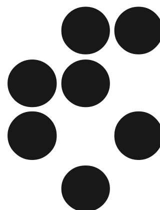
Očitno mu je tisti **for** i... ostal od prej, pa ga je pozabil pobrisati. Pri tej nalogi je stvar še posebej neugodna, ker gre lahko n do 10^9.

(Štiri v vrsto) Nekdo res rad uporablja operator za logično negacijo:

```
rekurzija(igralno_polje, tip.zmaga, −1, −1, !true, 0, 1, ref min, ref vrzi);
⋮
rekurzija(igralno_polje, tip.prepreci, −1, −1, !false, 0, 1, ref min, ref vrzi);
```

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.

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

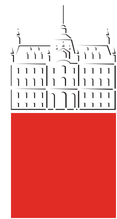
Osnovni cilji Odseka za inteligentne sisteme so raziskave računalniških osnov inteligence in razvoj naprednih aplikacij s področja inteligentnih informacijskih storitev, analize podatkov, inteligentnega preiskovanja spleta, podpore odločanja, inteligentnih agentov, medicine, ekologije, jezikovnih tehnologij, 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 POKROVITELJ

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.

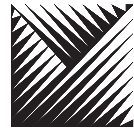
BRONASTI POKROVITELJI

alpineon

HERMES SoftLab

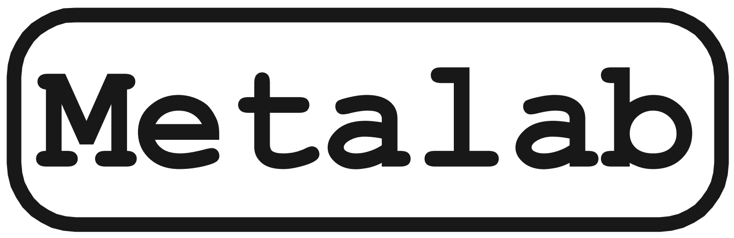
cosylab

CONTROL SYSTEM LABORATORY



MARAND

Napredna računalniška hiša



XLAB

NOT IDLE

IZOBRAŽEVANJE
INFORMACIJSKE STORITVE

finance IT your way

GLOBUS MARINE
INTERNATIONAL

NEVRON
rešitve e-izobraževanja