7. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 24. marca 2012

Bilten

Bilten 7. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2013

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Andrej Bauer, Andrej Brodnik, Primož Gabrijelčič, Matija Grabnar, Tomaž Hočevar, Nace Hudobivnik, Aleš Košir, Jurij Kodre, Mitja Lasič, Mark Martinec, Mojca Miklavec, Mitja Trampuš, Klemen Žagar, Janez Brank.

Tisk: Tiskarna knjigoveznica Radovljica, d. o. o.
Naklada: 200 izvodov

Ta bilten je dostopen tudi v elektronski obliki na domači strani tekmovanja:
`http://rtk.ijs.si/`

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 ACM v znanju računalništva (7 ; 2012 ; Ljubljana)
Bilten / 7. tekmovanje ACM v znanju računalništva, Ljubljana, 24. marca 2012 ; [avtorji nalog Nino Bašić ... [et al.] ; uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2013

Dostopno tudi na: `http://rtk.ijs.si/2012/rtk2012-bilten.pdf`

ISBN 978-961-264-051-4
ISBN 978-961-264-052-1 (PDF)
1. Bašić, Nino 2. Brank, Janez, 1979–
268121344

KAZALO

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 lahko svoje rešitve napišejo na papir ali pa jih natipkajo na računalniku, 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. Podobno kot v zadnjih nekaj letih smo tudi letos ponudili možnost, da tekmovalci v prvi in drugi skupini svoje odgovore natipkajo na računalniku; tudi tokrat so se zanjo odločili skoraj vsi tekmovalci.

V tretji skupini rešujejo vsi 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 program izpisal pravilni rezultat. (Podrobnosti točkovanja v 3. skupini so opisane na strani 22.) 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. 7–9 za 1. in 2. skupino, str. 21–23 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 142–143.

Podobno kot v zadnjih treh letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 27. januarja 2012. To je imelo eno samo težavnostno skupino, naloge (ki jih je bilo pet) pa so pokrivale precej širok razpon težavnosti. Tekmovalci so pisali odgovore na papir in dobili enak list z nasveti in navodili kot na državnem tekmovanju v 1. in 2. skupini (str. 7–9). Odgovore tekmovalcev na posamezni šoli so ocenjevali mentorji z iste šole, za pomoč pa smo jim pripravili nekaj strani z nasveti in kriteriji za ocenjevanje (str. 129–132). Namen šolskega tekmovanja je bil tako predvsem v tem, da pomaga šolam pri odločanju o tem, katere tekmovalce poslati na državno tekmovanje in v katero težavnostno skupino jih prijaviti. Šolskega tekmovanja se je letos udeležilo 273 tekmovalcev z 29 šol.

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;
```

(Opomba: samo zato, ker je tu primer psevdokode, to še ne pomeni, da moraš tudi ti pisati svoje odgovore v psevdokodi.)

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

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

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

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

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

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

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

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

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

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

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

```
#include <stdio.h>

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

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
```

Še isti trije primeri v javi:

```
// Branje dveh števil in izpis vsote:
import java.io.*;
import java.util.Scanner;

public class Primer1
{
    public static void main(String[] args) throws IOException
    {
        Scanner fi = new Scanner(System.in);
        int i = fi.nextInt(); int j = fi.nextInt();
        System.out.println(i + " + " + j + " = " + (i + j));
    }
}
```

```
// Branje standardnega vhoda po vrsticah:
import java.io.*;

public class Primer2
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader fi = new BufferedReader(new InputStreamReader(System.in));
        int i = 0, d = 0; String s;
        while ((s = fi.readLine()) != null) {
            i++; d += s.length();
            System.out.println(i + ". vrstica: \"" + s + "\""); }
        System.out.println(i + " vrstic, " + d + " znakov.");
    }
}
```

```
// Branje standardnega vhoda znak po znak:
import java.io.*;

public class Primer3
{
    public static void main(String[] args) throws IOException
    {
        InputStreamReader fi = new InputStreamReader(System.in);
        int i = 0, c;
        while ((c = fi.read()) >= 0) {
            System.out.print((char) c); if (c != '\n' && c != '\r') i++; }
        System.out.println("Skupaj " + i + " znakov.");
    }
}
```

NALOGE ZA PRVO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del v datoteki. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev samodejno shranjuje in na zaslonu ti bo pisalo, kdaj se bo shranjevanje naslednjič zgodilo. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) **Zaradi varnosti priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku** (npr. kopiraj in prilepi v Notepad in shrani v datoteko).

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite (bolj učinkovite rešitve dobijo več točk). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

1. Prepletene besede

Napiši podprogram ali funkcijo Prepletene, ki kot parameter dobi niz `s` in iz njega sestavi nov niz (recimo mu `t`) tako, da vse velike črke premakne na začetek niza, vse male črke pa na konec niza. Pri tem naj se vrstni red velikih in malih črk ohrani. Tako dobljeni niz `t` lahko tvoj podprogram vrne kot funkcijsko vrednost ali pa kot drugi parameter po referenci (kar ti je lažje). Predpostavi, da se v nizu pojavljajo le črke angleške abecede.

Primer: `PbREPeseLEdTiEloNO` → `PREPLETENObesedilo`

Tvoj podprogram naj bo takšne oblike:

```
procedure Prepletene(s: string; var t: string);          { v pascalu }
function Prepletene(s: string): string;                  { vrne t }

void Prepletene(const char *s, char *t);      /* v C/C++; predpostavi, da t že kaže na
                                                 primerno velik kos pomnilnika */

string Prepletene(string s);                             // v C++; vrne t
void Prepletene(string s, string& t);                    // v C++

public static String Prepletene(String s);               // v javi

public static string Prepletene(string s);               // v C#; vrne t
public static void Prepletene(string s, out string t);   // v C#

def Prepletene(s): ...                                   # v pythonu; vrne t
```

2. Manjkajoča števila

Na vhodni datoteki imamo zapisana naravna števila v naraščajočem vrstnem redu, vsako število v svoji vrstici. Večinoma gre za zaporedna števila, vendar včasih kakšno manjka, lahko manjka tudi več zaporednih števil.

Napiši program, ki bo bral števila z vhodne datoteke in jih **sproti** prepisoval na izhodno datoteko, pri tem pa izpisal tudi vsa morebitna manjkajoča števila, a ta naj izpiše v oklepajih. Manjkajočih naravnih števil pred prvim prebranim in za zadnjim prebranim ne izpišemo. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `stevila.txt` (kar ti je lažje). Program naj bere vse do konca vhoda (EOF).

Primer vhodnih podatkov:

```
4
5
6
7
9
10
11
17
18
```

Pripadajoči izpis:

```
4
5
6
7
(8)
9
10
11
(12)
(13)
(14)
(15)
(16)
17
18
```

3. Kazenski stavek

V šoli je bilo včasih za posebne dosežke treba ostati v šoli in večkrat na tablo napisati „kazenski stavek“. Tako se je po pol ure vrtenja krede in razmišljanja o tem, kako te nihče ne mara, na tabli pojavilo nekaj takega:

```
NE BOM VEC METAL PAPIRCKOV PO TLEH NE BOM VEC METAL PAPIRCKOV PO
TLEH NE BOM VEC METAL PAPIRCKOV PO TLEH NE BOM VEC METAL PAPIRCKOV
PO TLEH NE BOM VEC METAL PAPIRCKOV PO TLEH NE BOM VEC METAL
PAPIRCKOV PO TLEH NE BOM VEC METAL PAPIRCKOV PO TLEH NE BOM
VEC METAL PAPIRCKOV PO TLEH NE BOM VEC METAL PAPIRCKOV PO TLEH NE
BOM VEC METAL PAPIRCKOV PO TLEH
```

Napiši podprogram ali opiši postopek, ki bo iz zapisa na tabli ugotovil, kolikokrat se v njem pojavi kazenski stavek. Predpostavi, da je zapis na tabli podan kot en sam dolg niz (torej ni razdeljen na več vrstic). Kazenski stavek vzemi kot najkrajši tak kos besedila, za katerega velja, da je mogoče dani zapis s table sestaviti iz več ponovitev tega stavka, ločenih s po enim presledkom. Predpostaviš lahko, da se pisec ni zmotil ter da se v vhodnem nizu pojavljajo le velike črke angleške abecede in presledki.

V zgornjem primeru se kazenski stavek „`NE BOM VEC METAL PAPIRCKOV PO TLEH`“ pojavi 10-krat.

Če pišeš podprogram, naj bo takšne oblike:

```
function KazenskiStavek(s: string): integer;      { v pascalu }
int KazenskiStavek(char *s);                      /* v C/C++ */
int KazenskiStavek(string s);                     // v C++
public static int kazenskiStavek(String s);       // v javi
public static int KazenskiStavek(string s);       // v C#
def KazenskiStavek(s): ...                        # v pythonu; vrne int
```

4. Mase

V neki tovarni na koncu proizvodnega procesa vsak izdelek stehtajo, maso v gramih pa zapišejo v bazo. Kot vsaka tehtnica ima tudi ta znano svojo natančnost, ki je $\pm a$ gramov (torej, če tehtamo izdelek z maso m, lahko tehtnica pokaže katero koli število od vključno $m - a$ do vključno $m + a$).

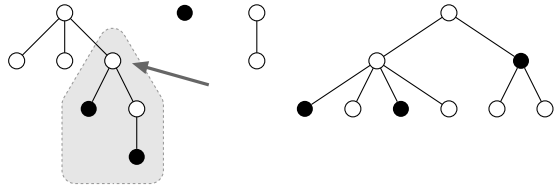
Izdelki so različnih vrst, vsi izdelki iste vrste imajo enako maso, izdelki različnih vrst pa imajo različne mase. Težava je v tem, da ne vemo, koliko vrst izdelkov obstaja in kakšne so njihove prave mase. **Opiši postopek**, ki kot vhodne podatke dobi natančnost tehtnice a in zaporedje meritev (za vsako meritev imamo izmerjeno maso v gramih) ter iz teh podatkov ugotovi, kakšno je najmanjše število različnih vrst izdelkov, pri katerem bi se (glede na dano natančnost tehtnice a) dalo s tehtanjem dobiti takšno zaporedje mas, kot je podano v vhodnih podatkih. Predpostavi, da je vhodno zaporedje meritev podano v naraščajočem vrstnem redu.

Primer: če imamo $a = 5$ in zaporedje meritev 15, 24, 26, je pravilni odgovor 2. Če bi imeli $a = 5$ in zaporedje meritev 15, 24, 25, pa bi bil pravilni odgovor 1 (kajti do vseh teh treh meritev lahko pride med drugim tako, da imamo eno samo vrsto izdelkov z maso 20).

5. V Afganistan!

Imamo n vojakov, oštevilčenih s števili od 0 do $n - 1$. Med njimi vlada stroga hierarhija: nekaj vojakov ima najvišji možen čin (in jim ni nihče nadrejen), vsak od preostalih vojakov pa ima natanko enega neposredno nadrejenega vojaka.

Nekatere od vojakov — recimo jim gajstni — želimo poslati v Afganistan. Kateremu koli vojaku (gajstnemu ali ne) lahko damo ukaz, naj se odpravi, in bo to seveda naredil, pri tem pa bo s seboj odpeljal tudi vse sebi podrejene vojake. Za primer glej sliko: vsaka točka predstavlja vojaka, vojaki z višjimi čini so narisani višje. Gajstni vojaki so pobarvani črno. Če damo ukaz vojaku, označenemu s puščico, bo s seboj v Afganistan odpeljal še preostale tri vojake iz črtkano obrobljenega območja.



Na primeru s slike se očitno ne da izdati enega samega ukaza, s katerim bi poslali na pot vse gajstne vojake. Tudi dva ukaza ne zadoščata; lahko pa to naredimo s tremi ukazi, celo na dva različna načina. Zanima nas, koliko ukazov potrebujemo v splošnem.

Napiši program, ki kot podatke dobi opis vojaške hierarhije ter gajstnih vojakov in vrne najmanjše število ukazov, s katerim lahko dosežemo, da bodo v Afganistan odpotovali vsi gajstni vojaki. Če pri tem na pot ukažemo še kakšnim ne-gajstnim vojakom, ni nič narobe. Za dostop do podatkov uporabi naslednji funkciji:

- Nadrejeni(i), ki vrne zaporedno številko vojaka, neposredno nadrejenega vojaku i, oziroma -1, če ima vojak i najvišji možen čin in mu ni nihče nadrejen;
- Gajsten(i), ki vrne 1, če je vojak i gajsten (in ga torej moramo poslati v Afganistan), in 0 sicer.

Predpostavi še, da je na voljo spremenljivka n, ki vsebuje število vseh vojakov.

NALOGE ZA DRUGO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del v datoteki. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev samodejno shranjuje in na zaslonu ti bo pisalo, kdaj se bo shranjevanje naslednjič zgodilo. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) **Zaradi varnosti priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku** (npr. kopiraj in prilepi v Notepad in shrani v datoteko).

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite (bolj učinkovite rešitve dobijo več točk). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

1. Ovce

Novozelandski kmet ima n ovac. V času t morajo biti vse ovce ostrižene, saj bo takrat odprodal volno. Ima seznam m strižcev. Za vsakega ima dva podatka:

- p_i je plačilo, ki ga zahteva i-ti strižec za striženje ene ovce;
- t_i je čas, ki ga porabi i-ti strižec za striženje ene ovce.

Aparat za striženje pokuri za c denarnih enot elektrike na časovno enoto. **Opiši postopek**, ki iz teh podatkov ugotovi, kolikšen je najmanjši znesek, ki ga bo kmet plačal za striženje (najem delovne sile + elektrika), če delovno silo izbira optimalno. Vsi strižci lahko delajo hkrati in vsak ima svoj aparat za striženje. Tisti, ki začne striči ovco, jo mora tudi dokončati. Na voljo bo vedno dovolj delavcev, da bodo lahko v danem času ostrigli vse ovce.

Primer: recimo, da imamo $n = 5$ ovac, $t = 12$ enot časa, ceno elektrike $c = 1$ in dva strižca: enega s $p_1 = 6$, $t_1 = 2$ (drag, a hiter) in enega s $p_2 = 2$, $t_2 = 5$ (počasen, a cenejši). Potem se izkaže, da je najmanjši skupni strošek striženja enak 38 (dosežemo ga, če prvi strižec ostriže tri ovce, drugi pa dve).

2. Spričevala

Da bi v prihodnje hitreje odkrili sumljiva (ponarejena) spričevala, so te v javni upravi prosili za pomoč. **Napiši program**, ki bo za vsa spričevala zaposlenih preveril to,

- ali je bil kateri ravnatelj v istem letu podpisan na spričevalih več šol
- ali sta se v istem letu pod katerikoli dve spričevali iste šole podpisali različni osebi,

in v obeh primerih opozoril, da bi bilo koristno preveriti podatke.

Za vsa spričevala so na voljo podatki o šoli, kjer je bilo spričevalo izdano, letu izdaje in podpisanem ravnatelju. Da bi ti bilo lažje, so ti že pripravili vhodne podatke, kjer so šole namesto s polnim imenom predstavljene s števili od 1 do s in ravnatelji s števili od 1 do r.

Tvoj program naj podatke bere s standardnega vhoda ali pa iz datoteke z imenom `spricevala.txt` (kar ti je lažje). V prvi vrstici so števila S (število šol), R (število ravnateljev) in n (število spričeval); veljalo bo $1 \leq S \leq 10^6$ in $1 \leq R \leq 10^6$. Sledi n vrstic, za vsako spričevalo po ena, v njej pa so zaporedoma zapisani leto, šifra šole in šifra ravnatelja. Te vrstice so urejene naraščajoče glede na leto izdaje spričevala.

Če npr. dobi naslednje vhodne podatke:

```
6 5 12
1995 5 2
1995 6 1
1995 5 2
1995 5 3
1998 5 4
2000 3 1
2000 2 5
2000 4 1
2001 4 2
2010 1 1
2010 2 3
2010 2 1
```

mora tvoj program izpisati:

```
Preveri spricevala sole st. 5 v letu 1995.
Preveri spricevala s podpisom ravnatelja st. 1 v letu 2000.
Preveri spricevala sole st. 2 v letu 2010.
Preveri spricevala s podpisom ravnatelja st. 1 v letu 2010.
```

Vrstni red izpisa ni pomemben, izogibaj se le podvojenim izpisom.

3. Razpolovišče lika

Nekoč je živel kralj, ki je umrl. Kraljestvo je v oporoki zapustil svojima sinovoma — vsakemu natanko polovico. Njegova želja je, da se kraljestvo razdeli na dva dela s popolnoma ravno mejo, ki poteka točno v smeri vzhod–zahod. Dvorni geometri imajo težavo določiti primerno lego meje. Pomagaj jim.

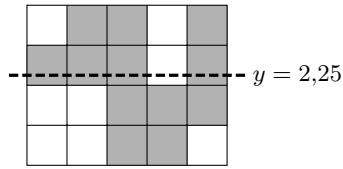
Zemljevid obstoječega kraljestva je znan. Vrisan je na karirasto mrežo širine w in višine h kvadratkov in orientiran tako, da kaže sever navpično navzgor. Na mreži je vsak kvadratek bodisi bel bodisi črn; črni kvadratki sestavljajo kraljestvo.

Napiši podprogram (funkcijo), ki pregleda zemljevid kraljestva in vrne y-koordinato tiste vodoravne črte (premice), za katero sta površini kraljestva pod in nad črto enaki. Če ob razrezu kraljestva „podkraljestvi“ razpadeta na več kosov, nas to ne moti. Koordinatni sistem postavimo tako, da ima izhodišče v spodnjem levem vogalu zemljevida, enota pa je enaka stranici enega kvadratka na karirasti mreži. Pozor — iščemo točno vrednost za y in ta ni nujno celoštevilska. (Predpostavi, da so podatkovni tipi, ki jih tvoj programski jezik uporablja za predstavitev realnih števil (npr. **float** ali **double**) dovolj natančni za potrebe te naloge, torej ti ni treba skrbeti zaradi morebitnih majhnih zaokrožitvenih napak.)

Nalogo lahko rešiš tudi tako, da izpišeš *celoštevilski* y, ki čim bolje (čeprav morda ne čisto natančno) razpolovi kraljestvo. Takšne rešitve bodo vredne največ 10 točk (od 20).

Pri pisanju podprograma privzemi, da imaš nekje že definirani spremenljivki `w` in `h`, ki hranita širino oz. višino zemljevida (v številu kvadratkov), ter funkcijo `Znotraj(x, y)`, ki vrne 1, če je kvadratek s koordinatama (x, y) na zemljevidu črn, sicer pa 0.

Primer:



Na sliki je $w = 5$, $h = 4$. Vrisana je tudi iskana rešitev — meja poteka na višini $y = 2{,}25$. V tem primeru je vrstica razrezana po četrtini in zato južni polovici kraljestva prispeva en kvadratek ploščine, severni polovici pa tri kvadratke. Tako imata severna in južna polovica res enako skupno ploščino, namreč 6 kvadratkov. Rešitev za 10 točk, ki vedno vrača celoštevilske y, pa bi morala pri primeru s slike vrniti $y = 2$.

4. Strukturirani podatki

Na vhodni datoteki imamo strukturirano besedilo v obliki, ki je podobna HTML ali XML, a s preprostejšim zapisom. Lahko si ga predstavljamo kot knjigo, ki vsebuje poglavja, ta vsebujejo podpoglavja, ta odstavke in tako naprej.

Vsaka vrstica vhodne datoteke lahko vsebuje začetno značko, končno značko ali pa neko poljubno vrstico besedila.

Začetna značka je sestavljena iz znaka „`+`“, ki mu sledi neko poljubno ime značke, npr:

```
+uvod
```

Končna značka je sestavljena iz znaka „`-`“, ki mu sledi ime značke, ki se na tem mestu zaključuje, npr:

```
-uvod
```

Predpostavimo lahko, da so vhodni podatki pravilni: začetne in končne značke so lahko gnezdene, pri tem se ime končne značke vedno ujema z imenom zadnje začetne značke, zato ga ni treba preverjati.

Vse ostale vrstice lahko vsebujejo poljubno besedilo. Te vrstice se ne morejo začeti z znakoma „`+`“ ali „`-`“.

Zaradi preglednosti lahko na začetku vsake vrstice stojijo presledki, ki pa jih ignoriramo.

Primer takega besedila na vhodni datoteki:

```
+ena
  +dve
    alfa
  -dve
  +tri
    beta
    gama
    +stiri
      delta
    -stiri
    epsilon
  -tri
  zeta
  +tri
  -tri
-ena
eta
```

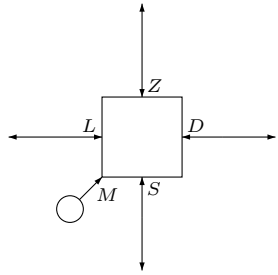
Napiši program, ki bo bral vhodne podatke v tej obliki in sproti izpisoval vse vrstice besedila, ki niso značke; pri tem pa naj pred vsako tako vrstico izpiše vse trenutno aktivne značke (v takem vrstnem redu, kot so gnezdene druga v drugi), med seboj ločene s pikami. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `vhod.txt` (kar ti je lažje). Bere naj vse do konca (EOF). Predpostavi, da je posamezna vrstica dolga največ 100 znakov in da značke niso gnezdene več kot 100 nivojev globoko. Imena značk so sestavljena le iz črk angleške abecede.

Takle naj bo rezultat obdelave zgornjega primera:

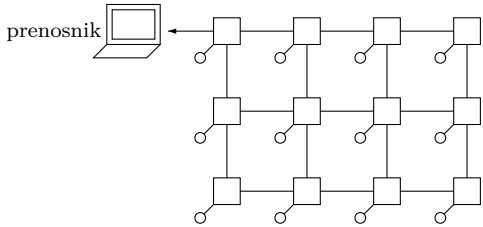
```
ena.dve alfa
ena.tri beta
ena.tri gama
ena.tri.stiri delta
ena.tri epsilon
ena zeta
 eta
```

5. Največji pretok

Imamo mrežo enostavnih merilnih računalnikov. Vsakega sestavljajo procesor, senzor in pet komunikacijskih kanalov. Štirje kanali so namenjeni povezavi med računalniki: L (levo), D (desno), Z (zgoraj) in S (spodaj). Peta povezava, M (merilnik) sprejema podatke od senzorja.



Računalniki so s kanali L, D, Z in S povezani v dvodimenzionalno mrežo.



Mrežo potopimo v reko, pri čemer merilniki na posameznih računalnikih merijo pretok vode v reki. Na zgornji levi računalnik priključimo prenosnik, ki bo zajemal podatke.

Na vseh računalnikih je pognan enak program, ki se izvaja v neskončni zanki. Program ima na voljo funkciji Beri in Pisi.

```
function Beri(Kanal: char): integer;                    { v pascalu }
procedure Pisi(Kanal: char; Vrednost: integer);

int Beri(char kanal);                                   /* v C/C++ */
void Pisi(char kanal, int vrednost);

public static int Beri(char kanal);                     // v javi/C#
public static void Pisi(char kanal, int vrednost);

def Beri(kanal): ...# vrne int                          # v pythonu
def Pisi(kanal, vrednost): ...
```

Funkcija Beri prebere podatek iz komunikacijskega kanala (če je na voljo) in vrne njegovo vrednost. Če je kanal prazen ali pa ni nikamor povezan (večina kanalov na robu mreže), funkcija vrne -1.

Podprogram Pisi zapiše podatek na podani kanal. Predpostaviš lahko, da je v kanalu vedno dovolj prostora za zapis podatka. Če ta kanal ni nikamor povezan, podprogram ne naredi ničesar.

S prenosnikom, priključenim na zgornji levi računalnik, bi radi prebrali največji izmerjeni pretok reke v celem omrežju. **Napiši program**, ki se bo izvajal na vseh računalnikih v omrežju in bo poskrbel za to, da bo zgornji levi računalnik na kanal L pošiljal največji doslej zaznani pretok (največji od začetka delovanja celotne mreže). Predpostavi, da so procesorji hitri, komunikacija med njimi tudi, sprejemljivo pa je, če pride podatek o največjem pretoku do zgornjega levega računalnika z majhno zakasnitvijo.

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 imenik `U:\_Osebno`, v katerem lahko kreiraš svoje datoteke. 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 1.7 in s prevajalnikom za C# iz Visual Studia 2008. Za delo lahko uporabiš `FP` oz. `ppc386` (Free Pascal), `GCC`/`G++` (GNU C/C++ — command line compiler), `javac` (za javo 1.7), Visual Studio 2010 in druga orodja.

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 ali pa porabil več kot 200 MB pomnilnika, 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.

Preden oddaš kak program, ga najprej prevedi in testiraj na svojem računalniku, oddaj pa ga šele potem, ko se ti bo zdelo, da utegne pravilno rešiti vsaj kakšen testni primer.

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. (Izjema je tretja naloga, kjer je testnih primerov 20 in za pravilen odgovor pri posameznem testnem primeru dobiš 5 točk.) Nato se 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. Liste z nalogami lahko po tekmovanju obdržiš.

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 %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. de-FFT permutacija (`defft.in`, `defft.out`)

Obveščevalna agencija SOVA se je odločila, da za šifriranje svojih sporočil ne bo zaupala ameriškim standardom, kot sta SHA-2 in 3DES. Raje so razvili lastno, novo kodirno funkcijo po imenu FFT (Frišna Funkcija proti Terorju). FFT zakodira podani niz znakov $s_1 s_2 s_3 s_4 \ldots$ tako, da znake premeša po pravilu

$$\text{FFT}(s_1 s_2 s_3 s_4 \ldots) = \text{FFT}(s_1 s_3 s_5 \ldots) + \text{FFT}(s_2 s_4 s_6 \ldots),$$

pri čemer + pomeni navadno stikanje nizov. Zgornja rekurzivna definicija velja za vse vhodne nize dolžine 2 ali več; za vhodne nize dolžine 1 pa nimamo česa premešati in definiramo kar $\text{FFT}(s_1) = s_1$.

Primer 1:

$$\begin{aligned} \text{FFT}(\texttt{sadje}) &= \text{FFT}(\texttt{sde}) + \text{FFT}(\texttt{aj}) \\ &= (\text{FFT}(\texttt{se}) + \text{FFT}(\texttt{d})) + (\text{FFT}(\texttt{a}) + \text{FFT}(\texttt{j})) \\ &= ((\text{FFT}(\texttt{s}) + \text{FFT}(\texttt{e})) + \texttt{d}) + (\texttt{a} + \texttt{j}) \\ &= \texttt{sedaj} \end{aligned}$$

Primer 2 (izpeljavo preskočimo): $\text{FFT}(\texttt{kokain\_crv}) = \texttt{krik\_ovnac}$

Dokaži Sovi, da FFT ni prav dobra šifra. **Napiši program**, ki dešifrira poljuben niz, zašifriran s FFT.

Vhodna datoteka: v prvi vrstici je število n ($1 \leq n \leq 100$), število šifriranih sporočil. Vsaka od naslednjih n vrstic vsebuje po eno šifrirano sporočilo y_i, sestavljeno izključno iz malih črk angleške abecede ter podčrtajev. Sporočila ne bodo daljša od 1024 znakov. V 30 % testnih primerov bodo vse dolžine sporočil potence števila 2.

Izhodna datoteka: vanjo po vrsti izpiši vseh n izvornih sporočil x_i, vsako v svoji vrstici. To, da je x_i „izvorno sporočilo“, pomeni, da velja zveza $\text{FFT}(x_i) = y_i$.

Primer vhodne datoteke:

```
9
sedaj
krik_ovnac
tutla_orask
avion_desno
desna_avtor
enak_nitro
lesk_oktav
uhan_serje
star_mivka
```

Pripadajoča izhodna datoteka:

```
sadje
kokain_crv
tolsta_kura
adonis_oven
danost_reva
enkrat_oni
lokast_vek
usnjar_ehe
smrkav_ati
```

2. Potovanje (`potovanje.in`, `potovanje.out`)

S prijatelji se odpravljate na dolgo potovanje. Pot, ki vas bo vodila od zahoda proti vzhodu, ste že izbrali. Prav tako ste označili vse bencinske črpalke ob poti in količino goriva, ki jo lahko kupite na posamezni črpalki. Izkaže pa se, da je dolžina poti, ki jo boste lahko prevozili, odvisna od začetne lokacije vašega potovanja. Ko vam zmanjka goriva, se bo vaša pot namreč končala. Avto porabi 1 enoto goriva na kilometer in ima neomejeno prostornino tanka za gorivo. Prišlo je do manjšega prepira, saj se nikakor ne morete dogovoriti, kje začeti. Da bi bila odločitev lažja, **napiši program**, ki bo za vsako črpalko izračunal, kakšno razdaljo lahko prepotujete, če svojo pot začnete na tej črpalki.

Vhodna datoteka: v prvi vrstici je število bencinskih črpalk n (velja $n \leq 10^6$). V naslednjih n vrsticah sledijo opisi črpalk; i-ta od teh vrstic vsebuje dve celi števili, x_i in g_i, ločeni s presledkom, pri čemer je x_i oddaljenost i-te črpalke od zahodnega konca poti v kilometrih, g_i pa je količina goriva, ki je na voljo na tej črpalki. Obe števili sta med vključno 1 in 10^9. Črpalke so podane v vrstnem redu, kot si sledijo od zahoda proti vzhodu. V 40 % testnih primerov bo veljalo $n \leq 10^4$.

Izhodna datoteka: za vsako črpalko (po vrsti od zahoda proti vzhodu) izpiši po eno vrstico, ki naj vsebuje največjo možno prepotovano razdaljo v kilometrih, če začnete svojo pot na tej črpalki.

(*Nasvet:* skupna količina razpoložljivega goriva je lahko pri nekaterih testnih primerih večja od 2^{32}, zato je pri tej nalogi koristno za nekatere količine uporabiti kakšnega od 64-bitnih celoštevilskih podatkovnih tipov, na primer `int64` v pascalu, **long long** v C/C++ in **long** v javi/C#.)

Primer vhodne datoteke:

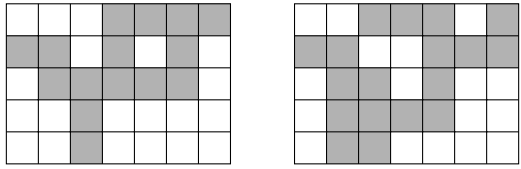
```
5
1 10
4 5
9 4
19 1
22 11
```

Pripadajoča izhodna datoteka:

```
20
9
4
1
11
```

3. Leteči pujsi (`pujsi.in`, `pujsi.out`)

Ptice so tako dolgo najedale pujsom, da so se odločili preiti v protinapad in tokrat napasti ptičjo trdnjavo. Ta je predstavljena s karirasto mrežo, pri čemer črna polja predstavljajo opeke, bela polja pa so prosta. Rekli bomo, da je opeka *stabilna*, če bodisi leži v najbolj spodnji vrstici mreže ali pa na svojem desnem, levem ali spodnjem robu meji na neko drugo stabilno opeko. Na primer, v spodnjih dveh mrežah so vse opeke stabilne:



Leteči pujs si lahko izbere eno od opek (katero koli) in se zaleti vanjo; izbrana opeka pri tem postane nestabilna, zaradi tega pa lahko postanejo nestabilne tudi nekatere druge opeke. **Napiši program**, ki za vsako polje mreže ugotovi, koliko stabilnih opek ostane na mreži, če se pujs zaleti v tisto polje. (Če se zaleti v polje, na katerem sploh ni bilo opeke, se v mreži nič ne spremeni.)

Vhodna datoteka: v prvi vrstici sta dve celi števili, ločeni s presledkom: najprej h (višina mreže) in nato w (širina mreže). Sledi h vrstic, ki opisujejo začetno stanje mreže. Vsaka od njih vsebuje w znakov, ki opisujejo tisto vrstico, pri čemer pika „`.`“ pomeni prazno polje, znak „`#`“ pa opeko. Začetno stanje mreže je pri vseh testnih primerih takšno, da je na mreži prisotna vsaj ena opeka in da so vse opeke stabilne. Veljalo bo $1 \leq w \leq 300$ in $1 \leq h \leq 300$. Pri 50 % testnih primerov bo veljalo tudi $1 \leq w \leq 100$ in $1 \leq h \leq 100$.

Izhodna datoteka: vanjo izpiši h vrstic, v vsako od teh pa w števil, ločenih s po enim presledkom. Vsako od teh števil naj pove, koliko stabilnih opek ostane na mreži, če se pujs zaleti v pripadajoče polje mreže.

Primer vhodne datoteke:

```
5 7
..###.#
##..###
.##.#..
.####..
.##....
```

Pripadajoča izhodna datoteka:

```
18 18 17 16 15 18 17
17 16 18 18 11 15 16
18 15 17 18 10 18 18
18 17 7 8 9 18 18
18 17 17 18 18 18 18
```

4. Nakup parcele (`parcela.in`, `parcela.out`)

V lasti imaš zemljišče, ki je razdeljeno na karirasto mrežo $h \times w$ parcel (h vrstic, w stolpcev; parcele so kvadratne oblike in so vse enako velike). Za vsako parcelo poznamo njeno vrednost, ki je celo število med vključno 1 in 10^9. Za nakup zemlje se zanima k kupcev, ki želijo kupiti različno velika zemljišča: i-ti kupec želi kupiti pravokotno zemljišče, visoko n_i parcel in široko m_i parcel. V navadi je, da kupec za zemljišče plača toliko, kot je vredna najcenejša parcela na njem. Cena, ki jo bo kupec plačal za zemljišče, je torej v splošnem lahko odvisna od tega, kje na naši mreži si bo to zemljišče izbral — v mreži velikosti $h \times w$ si lahko izberemo pravokotnik $n_i \times m_i$ na $(h - n_i + 1) \cdot (w - m_i + 1)$ načinov. Ker mi poznamo le velikost zemljišča, ki ga hoče ta kupec kupiti, ne pa tudi njegovega položaja, bi radi izračunali vsoto cene tako velikega zemljišča (torej $n_i \times m_i$ parcel) po vseh možnih položajih takega zemljišča v naši mreži. **Napiši program**, ki za vsakega kupca posebej izračuna to vsoto cene zemljišča.

Vhodna datoteka: prva vrstica vsebuje višino h in širino w tvojega zemljišča, ločeni s presledkom; veljalo bo $2 \leq h \leq 2000$ in $2 \leq w \leq 2000$. Naslednjih h vrstic vsebuje po w celih števil, ki predstavljajo vrednosti posameznih parcel. Temu sledi vrstica s številom kupcev k (velja $1 \leq k \leq 10$), nato pa k vrstic, ki opisujejo velikosti zemljišč, ki bi jih radi ti kupci kupili; nakup i-tega kupca je opisan z višino n_i in širino m_i (ločeni sta s presledkom), pri čemer velja $2 \leq n_i \leq h$ in $2 \leq m_i \leq w$.

Izhodna datoteka: izpiši k vrstic, ki za vsakega kupca vsebujejo vsoto cene zemljišča danih dimenzij po vseh možnih položajih takega zemljišča.

(*Nasvet:* vsota bo pri nekaterih testnih primerih večja od 2^{32}, zato je pametno zanjo uporabiti kakšnega od 64-bitnih celoštevilskih podatkovnih tipov, na primer `int64` v pascalu, **long long** v C/C++ in **long** v javi/C#.)

Primer vhodne datoteke:

```
3 5
8 5 5 8 1
6 4 8 3 8
8 2 8 7 7
2
2 3
3 3
```

Pripadajoča izhodna datoteka:

```
15
5
```

Razlaga primera: prvi kupec bi lahko kupil 6 različnih zemljišč. Njihove vrednosti od zgoraj navzdol in od leve proti desni so: 4, 3, 1, 2, 2 in 3. Vsota teh vrednosti je 15.

5. Rotacija (`rotacija.in`, `rotacija.out`)

Igralnica je kupila novo igro, ki se imenuje Kolo sreče. Sestavljena je iz velikega kolesa, ki ima na obodu napisane števke od 1 do 9. Igralec kolo zavrti in ko se kolo ustavi, se v smeri urinega kazalca prebere število, ki ga sestavljajo zaporedne števke na obodu. To število predstavlja dobitek igralca. **Napiši program**, ki iz opisa kolesa sreče izračuna največji možni dobitek.

Vhodna datoteka: v prvi vrstici je celo število n ($1 \leq n \leq 10^6$), ki pove, koliko je števk na kolesu. V drugi vrstici je niz n števk, kot si sledijo na kolesu sreče v smeri urinega kazalca. V 40 % testnih primerov bo veljalo $n \leq 10^4$.

Izhodna datoteka: izpiši največji možni dobitek, ki ga lahko zadenemo na opisanem kolesu sreče.

Primer vhodne datoteke:

```
6
425747
```

Pripadajoča izhodna datoteka:

```
747425
```

NALOGE ZA ŠOLSKO TEKMOVANJE

27. januarja 2012

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve, poleg tega, da so pravilne, tudi učinkovite (bolj učinkovite rešitve dobijo več točk). Pri vsaki nalogi lahko dobiš od 0 do 20 točk.

1. Kamen, papir, škarje

Dve osebi, A in B, igrata igro *kamen, papir, škarje*. Kamen ponazarja stisnjena pest, papir je iztegnjena dlan, škarje pa so predstavljene s kazalcem in sredincem, ki tvorita črko V.

Igra poteka tako, da pri vsaki potezi oba igralca ob določenem znaku hkrati pokažeta vsak svojo izbiro ene od treh dovoljenih vrednosti. Točko dobi tisti, katerega izbrana vrednost premaga izbrano vrednost nasprotnika. Če oba izbereta isto vrednost, nihče ne dobi točke.

Pri tem veljajo naslednja pravila:

- kamen premaga škarje,
- papir premaga kamen,
- škarje premagajo papir.

Potek igre imamo zapisan z nizom znakov. V tem nizu so z znaki „`K`“ (kamen), „`P`“ (papir) in „`S`“ (škarje) zapovrstjo zapisane vse poteze. Za vsako potezo je najprej izpisana vrednost, ki jo je v tej potezi pokazal prvi igralec A in nato še vrednost, ki jo je v tej isti potezi pokazal drugi igralec B.

Primer: niz `PSKPSP` predstavlja igro s tremi potezami; v prvi potezi je A pokazal papir, B pa škarje (torej je dobil točko B); v drugi potezi je A pokazal kamen, B pa papir (torej je dobil točko B); v tretji potezi je A pokazal škarje, B pa papir (torej je dobil točko A). Zmagovalec igre je torej B, ki ima dve točki, A pa le eno.

Napiši program, ki bo prebral niz znakov in na koncu izpisal zmagovalca (`A` ali `B`) oziroma morebitni neodločen rezultat. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `zapis.txt`, karkoli ti bolj ustreza.

2. Papajščina

Med otroki ali določenimi skupinami ljudi so se razvili nekateri načini spreminjanja govorjenih ali pisanih besed z namenom, da so tako predelane besede nerazumljive ali težko razumljive ostalim. Takih „skrivnih jezikov“ ali „jezikovnih igric“ je polno, na Wikipediji jih je naštetih skoraj sto (`http://en.wikipedia.org/wiki/Secret_language`).

Med slovenskimi otroki je popularna papajščina (ki uporablja enaka pravila kot španska jeringonza). Pravilo je preprosto: za vsakim samoglasnikom (`a`, `e`, `i`, `o`, `u`) vrinemo črko „`p`“ in za njo ponovimo isti samoglasnik. Tako se na primer beseda „`zdravo`“ predela v „`zdrapavopo`“.

Napiši program, ki bo bral besede (predpostavi, da je vsaka beseda v svoji vrstici in da so besede sestavljene le iz malih črk angleške abecede) ter vsako od njih izpisal kot besedo v papajščini. Program lahko bere s standardnega vhoda ali pa iz datoteke `besede.txt`, karkoli ti bolj ustreza. Obdela naj vse besede do konca vhodnih podatkov (EOF).

3. Obfuskator

Pri branju besedila vešči bralec ne prebira besed po črkah, ampak zajame s pogledom večje dele. Manjše tipkarske napake, na primer medsebojne zamenjave črk znotraj besede, branja ne otežujejo močno. Če prva in zadnja črka besede ostaneta na svojem mestu, vmesne med njima pa premešamo, je besedilo še vedno za silo čitljivo:

> Pri brjnau beeslida vešči barlec ne prbriea bseed po črkah, aampk zamjae s pgodloem večje dele. Manjše tpirkakse naakpe, na pmrier mdsebonjee zmanevjae črk ztnoarj besede, bnarja ne otežujejo mčono. Če pvra in zdnjaa črka beesde oatetnsa na sojvem mtesu, vmnese med nmija pa premešamo, je bedesilo še vedno za slio čitljivo:

Več o pojavu in o urbani legendi, povezani s tem, je zapisano na `http://www.mrc-cbu.cam.ac.uk/people/matt.davis/Cmabrigde/`.

Napiši program, ki bo v besedah naključno premešal črke in pri tem ohranil prvo in zadnjo črko na svojem mestu. Besede naj prebira z datoteke `besede.txt` ali pa iz standardnega vhoda, karkoli ti je lažje. Predpostavi, da je vsaka beseda zapisana v svoji vrstici in vsebuje le male črke angleške abecede, ločil in drugih znakov v datoteki ni. Preoblikovane besede naj tvoj program sproti izpisuje, vsako v svoji vrstici. Vsaka beseda je dolga kvečjemu 20 znakov.

Če ne veš, kako se v tvojem programskem jeziku generira naključna števila, lahko predpostaviš, da obstaja neka funkcija `Nakljucno(n)`, ki vrne naključno celo število od 0 do $n-1$ (in ima vsako od teh števil enako verjetnost, da ga bo funkcija vrnila).

4. Zarota

Dano je celo število k (večje od 0) in še neko zaporedje n celih števil: $a_1, a_2, \ldots, a_{n-1}, a_n$. **Opiši postopek**, ki ugotovi, koliko je v danem zaporedju takih strnjenih podzaporedij, ki imajo vsoto deljivo s k.[1] Tvoj postopek naj bo čim bolj učinkovit, tako da bo deloval hitro tudi pri velikih vrednostih n (bolj učinkovite rešitve dobijo več točk; predpostaviš lahko, da je k majhen v primerjavi z n).

Primer: recimo, da imamo $k = 10$ in naslednje zaporedje osmih števil:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
1	8	9	6	7	7	3	6

[1] Zanimivo, vendar težjo različico naloge dobimo, če poleg strnjenih dovolimo tudi nestrnjena podzaporedja. Na primer, $(5, 8, 7)$ se pojavlja kot nestrnjeno podzaporedje v zaporedju $1, 5, 9, 2, 8, 7, 6$.

Podzaporedja, ki imajo vsoto deljivo s $k = 10$, so v tem primeru štiri:

- od a_2 do a_5 (vsota je $8 + 9 + 6 + 7 = 30$);
- od a_4 do a_6 (vsota je $6 + 7 + 7 = 20$);
- od a_6 do a_7 (vsota je $7 + 3 = 10$);
- od a_2 do a_7 (vsota je $8 + 9 + 6 + 7 + 7 + 3 = 40$).

5. Sveče

Imamo n sveč, ki niso nujno vse enako visoke; na začetku so vse ugasnjene, podane pa so njihove začetne višine. Poleg tega imamo podatke o tem, kdaj smo sveče prižigali in ugašali. **Napiši program**, ki te podatke prebere in ugotovi, katera sveča je gorela najdlje. (Če je več enako dobrih, je vseeno, katero od njih izpiše.) Podatke lahko tvoj program bere s standardnega vhoda ali pa iz datoteke `svece.txt`, karkoli ti je lažje.

Vhodni podatki so naslednje oblike: v prvi vrstici je celo število n, ki pove, koliko je sveč (vsaj 1 in največ 10); v drugi vrstici je n celih števil, ločenih s presledki, ki povedo začetne višine sveč; vsaka od naslednjih vrstic pa vsebuje dve celi števili, t in x, ločeni s presledkom. Če je $x = -1$, ta vrstica pomeni, da ob času t vse sveče ugasnemo; sicer pa je x število od 1 do n in nam pove, da ob času t prižgemo x-to svečo (če še ne gori in če še ni povsem dogorela). Podatki so urejeni naraščajoče po času, zadnja vrstica pa ima zagotovo $x = -1$, tako da so sveče na koncu ugasnjene. Višine sveč so podane v centimetrih, časi v urah, hitrost gorenja sveč pa je pri vseh svečah enaka in znaša 1 cm na uro.

Primer vhodne datoteke:

```
3
10 12 5
2 3
3 2
4 -1
8 1
9 -1
```

Pri tem primeru imamo tri sveče z začetnimi višinami 10, 12 in 5; ob času $t = 2$ prižgemo tretjo svečo, ob času $t = 3$ prižgemo drugo svečo in ob času $t = 4$ obe ugasnemo; ob času $t = 8$ prižgemo prvo svečo in jo ob času $t = 9$ ugasnemo; najdlje je torej gorela tretja sveča, tako da je pravilni odgovor pri tem primeru 3.

NEUPORABLJENE NALOGE IZ LETA 2010

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 5. tekmovanjem ACM v znanju računalništva (leta 2010), 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. 75–128) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Povprečne temperature

Janez je dobil nalogo, da za mednarodno skupino vsak dan računa popvrečno temperaturo Slovenije. Njegov problem je v tem, da nekatere vremenske postaje pokrivajo področje, ki ni v Sloveniji, ampak v kateri od sosednjih držav. Te vremenske postaje sme v povprečni temperaturi upoštevati le toliko, kolikor odstotkov pokritega ozemlja te postaje je slovenskega.

Za to dobi podatke o postajah: za vsako postajo je ena vrstica, v njej pa sta dve števili; prvo od teh števil je izmerjena temperatura (v stopinjah Celzija), drugo pa je pa je celo število, ki pove, koliko odstotkov ozemlja, ki ga postaja pokriva, je v Sloveniji.

Napiši program, ki bo prebral te podatke in izpisal izračunano povprečje temperature. Pri tem predpostavi, da vsaka postaja pokriva enako veliko ozemlje, da se ozemlja različnih postaj med seboj ne prekrivajo, in da postaje v vhodnih datotekah skupaj pokrivajo celotno ozemlje Slovenije. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `meritve.txt` (kar ti je lažje).

2. Skakačev obhod

Skakačev obhod je matematično/programerski problem s skakačem na standardni šahovnici velikosti 8×8 polj. Na poljubno začetno polje postavimo skakača in z njim v 63 premikih obiščemo vsa ostala polja na šahovnici. Pri tem na vsako polje skočimo natanko enkrat. Skakač se premika po običajnih šahovskih pravilih — v vsakem premiku se premakne za dve polji v eno smer (levo, desno, gor ali dol) ter za eno polje pravokotno na originalno smer. Premika se torej v obliki velike črke `L`.

Napiši program, ki bo za neko zaporedje skakačevih potez preveril, ali gre res za pravi obhod. Preveriti mora, ali so vse poteze regularne in ali z njimi skakač obišče vsa polja na šahovski plošči.

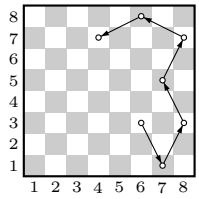
Program naj bere poteze iz standardnega vhoda. V prvi vrstici bosta zapisani koordinati začetnega polja (dve števili od 1 do 8), nato pa bo sledilo 63 premikov. Vsak premik je opisan z dvema številoma — prvo poda število polj, za katero se premakne skakač v eni smeri, drugo pa v drugi smeri. Premiki so lahko pozitivni ali negativni.

Če je zaporedje potez regularno in predstavlja skakačev obhod, naj program izpiše `OK`, sicer pa naj ne izpiše ničesar.

Primer začetka datoteke s podatki:

```
6 3
1 -2
1  2
-1 2
1 2
-2 1
-2 -1
...
```

Slika tega zaporedja premikov:



Predpostaviš lahko, da je na vhodu vedno prava količina podatkov ter da so začetni položaj ter premiki podani s celimi števili. Ni pa nujno, da vsi podatki predstavljajo pravilne koordinate ter dovoljene premike.

3. Pravokotni meseci

Sekta pravokotničarjev je nastala leta 2010. Tega leta je namreč skupina filozofov ugotovila, da obstajajo meseci, ki se začnejo na ponedeljek in končajo na nedeljo. Te mesece so poimenovali *pravokotni meseci*. Zaradi tega izrednega pojava tak mesec na koledarju izgleda najlepše. Ker so bili prepričani, da je zaradi tega pojava vsako leto s pravokotnim mesecem še posebej dobro, so želeli izračunati vsa leta s takim mesecem, a je bilo njihovo matematično znanje žal preveč omejeno. Zato se obračajo na tebe, izkušenega in uglednega programerja, da jim pomagaš pri tem problemu.

(*a*) **Napiši** jim **podprogram** PravokotniMeseci(letoOd, letoDo), ki izpiše vsa tista leta na območju od vključno letoOd do vključno letoDo, v katerih se pojavi pravokotni mesec. Podprogram naj leta izpiše po naraščajočem vrstnem redu, vsako v svojo vrstico.

V pomoč: prestopna leta so tista, ki so deljiva s 400, in tista, ki so deljiva s 4 in ne s 100.

(*b*) Zanimivo in malo težjo različico naloge dobimo, če namesto izpisa let s pravokotnim mesecem zahtevamo le izračun tega, koliko je takšnih let v nekem opazovanem obdobju, pri čemer pa je to obdobje lahko zelo dolgo (in bi radi učinkovit postopek).

(*c*) Še ena malo težja različica naloge je, da nas namesto pravokotnih mesecev nas zanimajo *dobri meseci*. Ti so definirani takole: mesec je dober natanko tedaj, če je skupno število sobot in nedelj v njem večje od 8.

Lažja različica naloge: predpostavi, da je na voljo funkcija funkcija DanVTednu(dan, mesec, leto), ki ti pove, na kateri dan v tednu pade dani datum. Funkcija vrne število od 1 (ponedeljek) do 7 (nedelja).

4. DNSx20

Ozadje naloge (katerega razumevanje ni nujno potrebno pri reševanju naloge): pri preslikavi internetnih imen domen v naslove IP (npr. domeni `rtk.ijs.si` pripada naslov 95.87.154.232) uporabljamo strežnike DNS. Poizvedbo z imenom domene pošljemo strežniku (skupaj z neko identifikacijsko številko), v odgovoru pa prejmemo pripadajoči naslov, skupaj s kopijo vprašanja. Kopija vprašanja v odgovoru je namenjena preverjanju pravilnosti delovanja in odkrivanju zlonamernih ponarejenih

odgovorov — odgovoru verjamemo le, če je kopija vprašanja v njem povsem enaka naši poizvedbi.

Glede na dokaj majhen nabor identifikacijskih številk in možnost ugibanja imen domen v poizvedbi ima morebitni ponarejevalec odgovorov precej možnosti, da naključno ugane naše vprašanje, ne da bi pri tem moral tudi prestreči poizvedbo. Radi bi torej zmanjšali možnosti napadalcu, da ugane vprašanje — ne da bi pri tem spremenili standardni protokol DNS za poizvedbe. Pri tem si lahko pomagamo z dejstvom, da so velike in male črke v imenu domene med seboj enakovredne, tako lahko `rtk.ijs.si` zapišemo tudi kot `RTK.ijs.si`, `rtk.IJS.si`, `Rtk.Ijs.Si` ali pa `rTk.ijS.SI`, in v vsakem primeru dobimo v odgovoru isti naslov IP (skupaj z nespremenjeno kopijo imena domene, po kateri smo poizvedovali).

Napiši program, ki bo prebral s standardnega vhoda eno vrstico, ki predstavlja ime domene. To je lahko sestavljeno iz velikih in malih črk angleške abecede, številk, in nekaterih posebnih znakov. Program naj predela prebrani niz tako, da bo v njem pri vsaki črki vrgel kovanec (poklical funkcijo za generiranje naključnih števil), in če bo padel grb (funkcija je vrnila vrednost 1) predelal malo črko v veliko, oziroma veliko v malo — če pa bo padla glava kovanca (vrednost 0), naj ostane črka nespremenjena. Znaki, ki niso črke, na ostanejo nespremenjeni. Po predelavi naj program predelani niz izpiše.

Na razpolago je funkcija za generiranje naključnih števil, ki ob vsakem klicu vrne vrednost 0 ali 1, pri tem je izbira med njima naključna in obe vrednosti sta enako verjetni:

```
function Kovanec: boolean; external;
```

5. CamelCase

Pri programiranju si včasih želimo kakšno spremenljivko ali funkcijo poimenovati z imenom, ki je sestavljeno iz več besed. Ker pa presledki v imenih spremenljivk in funkcij niso dovoljeni, moramo besede v takšnem večbesednem imenu ločiti nekako drugače. Dva priljubljena načina sta:

- ločevanje s podčrtaji: na primer: `dots_per_inch`, `print_stack_trace`
- camel case: besede zapišemo brez presledkov, vendar prvo črko vsake besede (razen prve) spremenimo v veliko; na primer: `dotsPerInch`, `printStackTrace`.

Napiši program, ki prebere besedilo s standardnega vhoda, predela vse besede iz camel case-a v besede, ločene s podčrtaji, in tako predelano besedilo sproti izpiše na standardni izhod.

Primer:

vhod: `Stopil je mimoVrste in naročil jagodniSladoled.`
izhod: `Stopil je mimo_vrste in naročil jagodni_sladoled.`

Različica naloge: **napiši program**, ki pretvarja v obratno smer, torej iz besed, ločenih s podčrtaji, v camel case.

6. Galci in Rimljani

Dan je seznam imen vojščakov (Galci in Rimljani). Razdeliti jih je potrebno v dve skoraj enako veliki legiji ($|A - B| \leq 1$, kjer je A velikost prve, B pa velikost druge):

(a) „skoraj sami Galci“ in

(b) „skoraj sami Rimljani“.

V idealnem primeru bi vsi Galci (imena na *-ix*, npr. Asterix, Obelix, Panoramix, ...) bili v skupini (a), ostali (torej, Rimljani) pa v skupini (b). Če je vojakov na *-ix* preveč, potem je lahko kateri od njih tudi v skupini (b). Če jih je premalo, potem skupino dopolnimo z rimskimi vojaki. (To je zaradi tega, ker homogena skupina boljše funkcionira kot mešana.) **Napiši program**, ki prebere seznam imen in ga izpiše tako, da bodo najprej vsi iz skupine (a), nato vrstica z znakom `#` in nato vsi iz skupine (b). Predpostaviš lahko, da je imen največ 1000 in da je posamezno ime dolgo največ 10 znakov.

7. Vagoni

Dva vlaka stojita v nasprotnih si smereh na dveh vzporednih tirih tako, da sta začetka njunih lokomotiv poravnana. Vsako od obeh kompozicij vlakov sestavljajo visoka lokomotiva, ki ji mešano sledijo visoki in nizki vagoni. Lokomotiva in vsi vagoni so enako dolgi. Oba vlaka imata enako število vagonov.

Na mestu, kjer sta začetka lokomotiv poravnana, stoji opazovalec, ki gleda pravokotno na tire. Opazovalec lahko vidi čez nizke vagone, čez visoke ne more.

Vlaka začneta voziti z enakima hitrostma drug mimo drugega.

Napiši podprogram `Vagoni(a, b)`, ki ugotovi, kolikokrat bosta med vožnjo vlakov mimo opazovalca pred opazovalcem hkrati nizka vagona v obeh kompozicijah, tako da bo lahko opazovalec videl čez njiju na drugo stran obeh vlakov. Kot vhod dobi dva niza, `a` in `b`, ki po vrsti opisujeta višine vagonov v vlakih, od leve proti desni (gledano z zornega kota opazovalca). Pri tem črka `v` pomeni visok vagon (ali lokomotivo), `n` pa nizkega.

8. Koren besed

V slovenščini sklanjamo samostalnike, npr. lipa, lipe, lipi, lipo, ... *Koren* samostalnika je tisti začetni del besede, ki se pri sklanjanju ne spreminja, npr. koren besede „lipa“ je „lip“.

Napiši podprogram `Koren(t)`, ki sprejme tabelo besed `t` in izpiše njihov koren. Na primer, če je `t` tabela

$$[\texttt{"lipam"}, \texttt{"lipah"}, \texttt{"lipama"}, \texttt{"lipami"}],$$

naj podprogram izpiše niz `"lipa"`.

9. Zemljevid

Nov slovenski iskalni portal `poišči.si` bo uporabnikom ponudil dostop do svetovnih zemljevidov. Napravili so ogromno sliko, sestavljeno iz satelitskih posnetkov, tvoja naloga pa je, da sprogramiraš vmesnik, ki glede na zahteve uporabnika pove, kateri del slike je potrebno izrisati na zaslon. Uporabnik se lahko premika v dveh smereh, gor-dol, levo-desno, ali pa poveča ali pomanjša sliko. Koordinate na zemljevidu pomenijo število kilometrov vzhodno (x) in število kilometrov južno (y) od središča pomembnega kraja Zelena vas.

Napiši podprograme, ki jih kliče uporabniški vmesnik:

```
void Init(int sirinaZaslona, int visinaZaslona, double x1, double y1, double kmNaPiksel);
void Premik(int deltaX, int deltaY);
void Povecava(double faktor);
```

`Init` se kliče na začetku, ko uporabnik odpre stran z zemljevidom. Navedejo se dimenzije zaslona v pikslih (npr. 640×400), pozicija levo zgoraj na zemljevidu (npr. 17,2 km vzhodno in 13 km južno od Zelene vasi), ter ločljivost (npr. 0,5 km na piksel). `Premik` se kliče, ko se uporabnik premakne po zemljevidu in pove, za koliko pikslov se je premaknil po zaslonu, ločljivost zemljevida pa ostane enaka. `Povecava` pa se kliče, ko uporabnik klikne na ustrezen gumb — zamenja se ločljivost zemljevida tako, da se na sredini zaslona kaže isti položaj na zemljevidu. Vsak od zgornjih podprogramov mora na koncu poklicati zunanjo funkcijo, ki na zaslon nariše izsek zemljevida:

```
void Narisi(double x1, double y1, double x2, double y2, double kmNaPiksel);
```

Pri tem sta `x1`, `y1` koordinati na zemljevidu, ki ustrezata zgornjemu levemu kotu slike, `x2`, `y2` pa koordinati na zemljevidu za spodnji desni kot slike.

10. Semaforji

Ko se je Kopitarjev Miha (Michael Schumacher) upokojil, je z avtom veliko dirkal po domači vasici. Ker pa so policisti zelo strogi, se je moral držati prometnih pravil: omejitev hitrosti, brez vožnje skozi rdeče luči in podobno.

Vas ima zelo enostavno cestno omrežje: ena sama cesta, ki vodi iz vzhodnega do zahodnega konca vasi, na njej pa je n semaforjev. Na semaforju i (kjer je i med 1 in n) se prvič prižge zelena luč t_i sekund po Mihovem startu in sveti d_i sekund. Cikel se ponovi vsakih p_i sekund.

Na primer, če se semafor 3 prvič prižge 15 sekund po startu ($t_3 = 15$), zelena luč sveti 10 sekund ($d_3 = 10$), in se cikel ponovi na 20 sekund ($p_3 = 20$), potem se rdeča luč prižge 25 sekund po startu ($t_3 + d_3$). Zelena luč se spet prižge 35 sekund po startu ($t_3 + p_3$), pa spet 55 sekund po startu ($t_3 + 2 \cdot p_3$) in tako naprej.

Podani so tudi časi r_i, ki povedo, koliko časa Miha porabi, da se prepelje od semaforja $i - 1$ do semaforja i. Ker je Miha zelo spreten voznik, lahko predpostaviš, da ima neskončne pospeške (od 0 do omejitve hitrosti v 0 sekundah!), tako da r_i ni odvisen od tega, ali je Miha pri semaforju i moral čakati na zeleno luč.

Napiši podprogram, ki izračuna, koliko časa Miha potrebuje za vožnjo do semaforja n. Njegova vožnja se začne nekje vzhodno od semaforja 1 (do katerega bo prišel po r_1 sekundah vožnje).

Predpostaviš lahko, da so vsi številski podatki cela števila.

11. Doroteja

Doroteja, lev, strašilo in drvar so se znašli v čudežni hiši. Hiša ima n sob in m vrat; vsaka vrata povezujejo po dve sobi. Sobe so pobarvane, ravno tako tudi vrata; barve vseh sob in vrat so znane. V vsakem koraku se lahko eden izmed naših štirih junakov premakne skozi vrata iz svoje trenutne sobe v sosednjo sobo, če vsaj dva izmed ostalih stojita v sobi iste barve, kot so vrata, skozi katera se bi rad premaknil.

Opiši postopek, ki ugotovi, ali se lahko vsi štirje srečajo v skupni sobi. Kot vhodni podatek dobiš njihov začetni položaj (v kateri sobi se kdo nahaja).

12. Zanimivi datumi

Zanimajo nas datumi, ki jih sestavlja čim več enakih števk.

Napiši program, ki bo v datumih od 1. 1. 0001 do 31. 12. 9999 izpisal vse datume, ki vsebujejo vsaj 6 enakih števk. Pri tem dan in mesec pišemo brez vodilnih ničel, leto pa vedno štirimestno z vodilnimi ničlami.

13. Računovodstvo

V podjetju izvajalca izdelujejo izdelke, ki jih prodajajo le enemu kupcu. Kupec ima veliko izpostav. Ko izdelovalec pošlje izdelke v neko izpostavo kupca, pošlje zraven tudi račun za poslane izdelke. Izpostava kupca vsake toliko časa pošlje račune v centralo kupca. Centrala zbira račune iz svojih izpostav in vsake toliko časa nakaže izvajalcu določen znesek.

V računovodstvu izvajalca imajo seznam parov (*datum*, *znesek*) za vse račune, ki so jih posredovali kupcu, in seznam parov (*datum*, *znesek*) za vsa plačila, ki so jih prejeli od kupca.

V podjetju izvajalca ne zapirajo sproti odprtih postavk izdanih računov, temveč le ob koncu poslovnega leta. Zapiranje postavk pomeni, da morajo za vsak izdan račun navesti, s katerim plačilom kupca je bil plačan. Oziroma, morajo za vsako plačilo kupca najti vse svoje račune, ki so v tem plačilu zajeti.

Ker je računov in plačil veliko, zneski pa so zelo različni, imajo pri zapiranju težave, zato si želijo pomagati z računalnikom.

Napiši podprogram, ki bo za vsako plačilo v danem seznamu plačil izračunal, kateri računi iz danega seznama računov so bili s tem plačilom plačani. Pri tem velja pravilo, da račun ne more biti plačan prej, preden je bil račun izdan.

Za pokrivanje računov s plačili uporabimo naslednji postopek, ki ga po vrsti izvajamo na vsakem plačilu iz seznama plačil: vzamemo plačilo in zapiramo po vrsti vse izdane račune, dokler ne „porabimo“ vsega zneska iz plačila, oziroma dokler ne ostane v plačilu znesek, ki je manjši od najmanjšega še nezaprtega računa. Morebitni ostanek prištejemo k naslednjemu plačilu kupca. Postopek nato ponovimo z naslednjim plačilom in tako naprej, dokler ne obdelamo vseh plačil kupca. Stanje na koncu je lahko izravnano, lahko imamo primanjkljaj (kupec je premalo plačal) ali pa imamo višek (kupec je preveč plačal).

14. Drevo

Napiši funkcijo `Drevo(int n)`, ki izpiše polno binarno drevo z 2^n listi. Izpis naj bo sestavljen iz znakov „`#`“ in „`.`“, kot kaže spodnji primer (za $n = 4$):

```
#.#.#.#.#.#.#.#
###.###.###.###
.#...#...#...#.
.#####...#####.
...#.......#...
...#########...
.......#.......
.......#.......
```

15. Podniz

Prijatelju želiš poslati skrivno sporočilo. Da bi zagotovil varno dostavo sporočila, ga boš poslal v dveh delih. Najprej mu boš poslal nek članek iz časopisa, za tem pa še naluknjan papir. Ko bo tvoj prijatelj oboje prejel, bo lahko članek prekril s papirjem in skozi luknje v papirju prebral sporočilo. Članek si že izbral, vendar je zelo dolg. Želiš ga čim bolj skrajšati s tem, da spredaj in zadaj odrežeš del besedila. Najmanj koliko znakov mora imeti skrajšan članek, da bo še vedno vseboval skrivno sporočilo?

Primer:
— članek: `bacbbacabacacba`
— sporočilo: `abc`
— rešitev: 4 (`bacbbac`<u>`abac`</u>`acba`)

16. Kamere

V televizijskem studiu snemajo intervju s pomembnim politikom in so v ta namen na različna mesta postavili kamere. Vse kamere so obrnjene v isto smer in imajo enak zorni kot (smer in zorni kot sta podana). Če gledamo s stropa studia, vsaka kamera pokriva del studia v obliki enakokrakega trikotnika. Vsaka kamera bo opisana s koordinatama (x, y) in dolžino območja, ki ga pokriva (višina trikotnika). Politik bi se rad postavil na mesto, kjer ga bodo snemale vse kamere. **Opiši postopek**, ki izračuna, kako veliko površino studia ima na voljo.

17. Produkt

Učiteljica matematike sestavlja test iz množenja. Test bo vseboval eno samo nalogo, ki se bo glasila „izračunaj produkt vseh podanih števil“. Pri sestavljanju testa je na papir zapisala n števil, izmed katerih jih bo k vključila v nalogo. Števila želi izbrati tako, da bo njihov produkt čim manj „okrogel“ — produkt se mora končati s čim manj ničlami, da ne bodo učenci rezultata kar uganili. **Opiši postopek**, ki izračuna, s koliko ničlami se bo končala pravilna rešitev naloge, če učiteljica za test optimalno izbere k števil iz množice danih n kandidatov.

Primer: $n = 4, k = 3$, števila 4, 15, 12, 75.
Rezultat: 1 ($4 \cdot 15 \cdot 12 = 720$). Če bi tri števila od teh štirih izbrali na kakršen koli drug način, bi imel zmnožek na koncu dve ničli, ne le eno.

18. Špekulacije

Menjalniške tečaje med valutami predstavimo z matriko (dvojno tabelo) m, pri čemer nam $m[i][j]$ pove, koliko enot valute j dobimo za eno enoto valute i. Indeksa i in j sta zaporedni številki valut (vse valute so oštevilčene). Denimo, če imata evro in ameriški dolar zaporedni številki 42 in 23 in je $m[42][23] = 1.45$, to pomeni, da za 1 evro dobimo 1.45 ameriškega dolarja.

Špekulanti na borzi iščejo *profitne cikle*, to so taka zaporedja menjav iz ene valute v drugo, ki se začnejo in končajo z isto valuto in ustvarijo dobiček. Na primer, če je m matrika

$$\begin{bmatrix} 1{,}0 & 2{,}0 & 0{,}4 & 5{,}0 \\ 0{,}5 & 1{,}0 & 0{,}002 & 2{,}5 \\ 2{,}5 & 500 & 1{,}0 & 10{,}0 \\ 0{,}2 & 0{,}4 & 0{,}1 & 1{,}0 \end{bmatrix}$$

imamo profitni cikel $3, 2, 0, 3$ dolžine tri, ker je

$$m[3][2] \cdot m[2][0] \cdot m[0][3] = 0{,}1 \cdot 2{,}5 \cdot 5{,}0 = 1{,}25.$$

Se pravi, če začnemo z eno enoto valute 3, ki jo po vrsti zamenjamo v valute 2, 0 in 3, dobimo 1,25 enote valute 3.

(*a*) *Lahka različica naloge:* **opiši postopek** `Spekulant(m)`, ki sprejme matriko menjalniških tečajev in izpiše na zaslon *profitni trikotnik*, to je profitni cikel, v katerem nastopajo tri različne valute. Cikel naj se izpiše kot zaporedje številk valut, vsaka v svojo vrsto. Če je možnih več profitnih trikotnikov, naj se izpiše eden od njih, če pa takega trikotnika ni, naj postopek izpiše „`ni rešitve`“.

(*b*) *Težja različica:* **opiši postopek** `Spekulant(m, k)`, ki sprejme matriko menjalniških tečajev `m` in pozitivno število `k` ter izpiše na zaslon profitni cikel dolžine k. Cikel naj se izpiše kot zaporedje številk valut, vsaka v svojo vrsto. Če je možnih več profitnih ciklov dolžine `k`, naj se izpiše eden od njih, če pa takega cikla ni, naj postopek izpiše „`ni rešitve`“.

(*b′*) Kaj pa, če ne zahtevamo, da so valute v ciklu različne?

(*c*) *Še težja različica:* **opiši postopek** `Spekulant(m)`, ki sprejme matriko menjalniških tečajev in izpiše na zaslon profitni cikel. Cikel naj se izpiše kot zaporedje številk valut, vsaka v svojo vrsto. Če je možnih več profitnih ciklov, naj se izpiše eden od njih, če pa takega cikla ni, naj postopek izpiše „`ni rešitve`“.

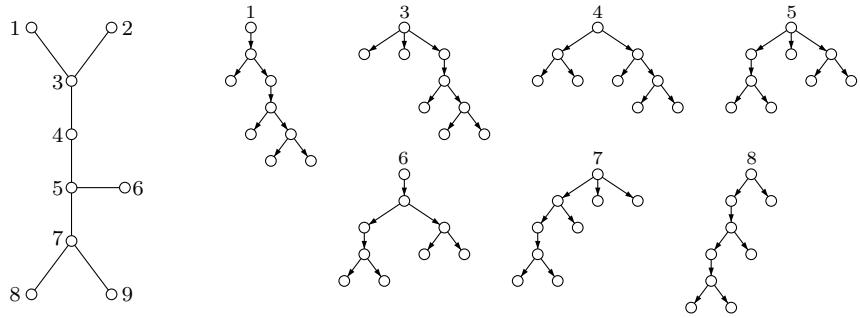
19. AVL-drevo

Dano je drevo s korenom. Rekli bomo, da je drevo *uravnoteženo*, če so uravnotežena vsa vozlišča v njem; vozlišče pa je uravnoteženo, če je višina njegovega najglobljega poddrevesa največ za 1 večja od višine njegovega najplitvejšega poddrevesa.

Opiši postopek, ki za dano drevo ugotovi, ali je uravnoteženo ali ne. Drevo je podano kot seznam parov (u, v), ki povedo, da je v otrok vozlišča u. Vozlišč je n in so oštevilčena s celimi števili od 1 do n.

Težja različica naloge: drevo je podano brez korena, torej kot povezan neusmerjen acikličen graf. **Opiši postopek**, ki ugotovi, na koliko načinov lahko v tem drevesu izberemo koren tako, da nastane uravnoteženo drevo. Naivna rešitev ima časovno zahtevnost $O(n^2)$, obstaja pa tudi boljša rešitev v $O(n \log n)$.

Spodnja slika kaže primer drevesa brez korena (na levi) in nekaterih dreves s korenom, ki jih lahko dobimo iz njega. Če za koren izberemo vozlišče 4 ali 6, dobimo uravnoteženo drevo, sicer pa ne.



20. Birokrati

Imamo n birokratov (oštevilčeni so od 1 do n), ki prekladajo papirje. Ko birokrat u opravi z nekim papirjem, ga lahko odloži na mizo enemu od naslednjih n_u birokratov; papir torej lahko v enam koraku pride od u do v natanko tedaj, če velja $u < v \leq u + n_u$. **Opiši postopek**, ki ugotovi, kakšno je najmanjše število korakov, v katerih bi se dalo papir spraviti od birokrata 1 do birokrata n. Birokratov je lahko do 10^6, zato naj bo postopek učinkovit.

21. Konjska vprega

V neki deželi je n vasi in m cest med njimi; vsaka cesta je dvosmerna in neposredno povezuje dve vasi (vmes se ne ustavi v nobeni tretji vasi). Kmet gre z vozom na pot iz vasi s in bi rad prišel v vas t. Ceste so blatne in se kolesa vdirajo, zato potrebujemo za cesto med vasema u in v vsaj d_{uv} konjev, da dovolj močno povlečejo voz. **Opiši postopek**, ki ugotovi, kakšno je najmanjše število konj, s katem lahko pridemo iz vasi s v vas t.

22. Mutacije

Dar Vin je odkril načelo, po katerem se vrste v naravi razvijajo z naravnim izborom. Iz podobnosti med vrstama je trdil, da se je človek razvil iz opice. Če poznamo genoma dveh različnih vrst, lahko s preučevanjem mutacij ugotovimo, najmanj koliko mutacij je potrebnih za spremembo genoma pripadnika ene vrste v genom pripadnika druge vrste.

Denimo, da vsebuje genom opice podzaporedje:

```
CAGTTC
```

in genom človeka podzaporedje

```
CAGAAC
```

pri tem pa so možne mutacije

```
AGT  →  AGA
AGAT →  AGTT
CC   →  CGC
CAG  →  GG
```

Tedaj vidimo, da sta bili uporabljeni zaporedoma prva in druga mutacija.

Opiši postopek, ki izpiše najmanjše število mutacij, potrebnih za transformacijo začetnega zaporedja v končno.

Pri tem upoštevaj:

- genom in mutacije so zapisani z znaki „`A`“, „`C`“, „`G`“ in „`T`“;
- mutacija se lahko izvede ničkrat, enkrat ali večkrat;
- mutacije se izvajajo od leve proti desni in mutacija ne more delovati nad delom niza, ki je nastal kot rezultat kakšne zgodnejše mutacije.

23. Potapljanje ladjic

Potapljanje ladjic je igra, ki se odvija na karirasti mreži. Naš nasprotnik razporedi po njej 5 ladjic; ena od njih ima pet polj, ena štiri polja, dve imata po tri polja, ena pa ima dve polji. Vsaka ladja je ali vodoravna ali navpična. Ladje se ne smejo prekrivati, lahko pa se dotikajo. Med igro ugibamo, na katerih poljih so nasprotnikove ladje; vsaka poteza je sestavljena iz tega, da povemo koordinati nekega polja (x_i, y_i), nasprotnik pa nam pove izid (zgrešil, zadetek, potopljena). Ladja je potopljena, ko so zadeta vsa njena polja. Ker sumimo, da nasprotnik goljufa, nam **opiši postopek**, ki za dano zaporedje potez (in izidov) ugotovi, ali sploh obstaja tak začetni položaj ladjic, ki je konsistenten s temi potezami.

24. Bubble sort

Recimo, da imamo v tabeli $a[0..n-1]$ shranjeno neko permutacijo števil od 1 do n. Na njej poženemo naslednji postopek za urejanje:

```
izpiši tabelo a;
za i = 1, 2, ..., n − 1:
  za j = 1, 2, ..., n − i:
    če je a[j] > a[j + 1], potem zamenjaj ta
      dva elementa v tabeli a in po zamenjavi tabelo a izpiši;
```

Ob vsakem izpisu se tabela izpiše v novo vrstico, posamezni elementi pa so ločeni s po enim presledkom. Po izpisu je nekdo odprl tako nastalo datoteko in iz nje izbrisal nič ali več vrstic, pri čemer pa vemo, da je prvo vrstico (ki vsebuje začetno stanje tabele) gotovo pustil pri miru.

Napiši program, ki iz datoteke prebere n, k in neko zaporedje k vrstic ter ugotovi, ali bi to zaporedje lahko nastalo z delovanjem gornjega postopka (vključno s tem, da se lahko nekatere vrstice (razen prve) iz izpisa pobriše).

25. Lonci

Ravnokar si se vrnil iz nakupovalnega centra z novim kompletom n kuhinjskih loncev, vendar si pred vhodnimi vrati naletel na manjšo težavo. Od avtomobilskega prtljažnika do kuhinje te namreč loči precejšnje število stopnic, v eni roki pa lahko hkrati neseš samo en lonec, saj drugo roko potrebuješ za odpiranje vrat. Da ti kljub temu ne bo treba prevečkrat po stopnicah, si se odločil lonce zložiti enega v drugega. Lonce razvrstiš na tla in jih postopoma zlagaš — izbereš nek lonec (ki morda vsebuje že druge lonce) in ga postaviš v večji prazen lonec. Lonec ima obliko valja in bo opisan s svojim polmerom in višino. Lonec je večji od nekega drugega lonca, če je višji in ima hkrati tudi večji polmer. **Opiši postopek**, ki ugotovi, najmanj kolikokrat boš moral po stopnicah, da znosiš vse lonce v kuhinjo.

Primer:
— seznam loncev (višina, polmer): $(5, 6)$, $(4, 4)$, $(3, 2)$, $(1, 3)$, $(2, 4)$
— rešitev: 2; enkrat nesemo $(1, 3)$ in $(2, 4)$, drugič pa $(3, 2)$, $(4, 4)$ in $(5, 6)$.

Težjo različico naloge dobimo, če namesto loncev prenašamo škatle. Vsaka škatla ima obliko kvadra in jo opišemo s širino, višino in globino. Eno škatlo lahko položimo v drugo le, če je po vseh treh dimenzijah manjša od druge.

26. Ropar na podzemni

Glavno mesto Vilanije ima zelo dobro razvito podzemno železnico. Vsi vlaki se na postajah ustavljajo hkrati v razmiku 5 minut in na postaji čakajo 2 minuti na nove potnike, kar je dovolj, da se potniki v miru prestopijo na drugi vlak.

Ob 17.35 so na neki postaji opazili roparja največje mestne banke NMB, ko se je vkrcaval na vlak. Ob 17.40 je na podzemno železnico na različne postaje prispelo nekaj (k) agentov varnostne službe, z namenom, da roparja ulove. Agenti so se vozili po podzemni železnici v upanju, da bodo na kateri od postaj zasačili roparja, vendar brez uspeha. Čez nekaj časa nihče ni več vedel, kje bi lahko tičal ropar, vedeli so le, da nobena od varnostnih kamer ni opazila roparja, da bi zapustil podzemno. Tvoja naloga je, da napišeš program, ki izpiše seznam postaj, na katerih se lahko nahaja ropar, če veš naslednje:

- ropar v vsem času ni zapustil podzemne
- agent bi ujel roparja, če bi se znašla ob istem času na isti postaji
- agent ne ujame roparja, če se vozita ob istem času po isti povezavi v nasprotnih smereh
- poznaš postajo, na kateri je ropar začel
- poznaš poti vsakega od agentov
- ropar je, da bi zmedel agente, vsakih pet minut potoval s podzemno (nikoli se ni za pet minut ustavil na postaji)

Vhodna datoteka: v prvi vrstici so navedena števila n, m, k, p in s (ločena s po enim presledkom; velja $1 \leq n \leq 10^4$, $0 \leq m \leq 10^5$, $1 \leq k \leq 10^4$, $1 \leq p \leq 100$,

$1 \leq s \leq n$), kjer je n število postaj, m število povezav med postajami, k število agentov, p število postaj, ki so jih prepotovali agenti, s pa je številka postaje, na kateri so roparja prvič opazili. Vsaka od naslednjih m vrstic vsebuje dve števili a in b (ločeni s presledkom), ki povesta številki postaj, med katerima poteka neposredna povezava z vlakom v obe smeri. (Povezave so oštevilčene s celimi števili od 1 do n.) Sledi še k vrstic, za vsakega agenta po ena, ki opisuje pot tega agenta (vsebuje številke postaj v takem vrstnem redu, v kakršnem jih je obiskoval; to je p števil, ločena so s po enim presledkom). Nobena postaja nima več kot 100 možnih povezav.

Izhodna datoteka: izpiši eno samo celo število, ki pove, na koliko postajah se lahko po p petminutnih intervalih nahaja ropar.

27. Prenos podatkov

Neko podjetje izdeluje program, ki zna prenašati podatke z enega na drug prazen disk. V prvem koraku je potrebno iz seznama datotek, ki jih želimo prenesti, na drugem disku ustvariti direktorijsko strukturo. V ta namen imamo na voljo ukaza `mkdir` in `chdir`, kjer prvi ustvari direktorij, `chdir` pa se premika iz direktorija v drug direktorij. Oba ukaza sta nekoliko nespretno napisana, tako da `chdir` kot parameter sprejema samo absolutne poti, `mkdir` pa samo ime direktorija. Še več, `mkdir` celo povzroči okvaro računalnika, če bi moral imenik kreirati v imeniku, ki ne obstaja.

Napiši program, ki prebere seznam datotek, ki jih želimo kreirati, ter izpiše zaporedje ukazov `chdir` in `mkdir`, ki zapišeta direktorijsko strukturo na drugi disk.

Primer vhodne datoteke:

```
3
C:\Files\Dokumenti\dokument.doc
C:\Files\Slike\Morje\slika1.gif
C:\Temp\nekaj.txt
```

Pripadajoča izhodna datoteka:

```
mkdir Files
chdir C:\Files
mkdir Dokumenti
chdir C:\Files
mkdir Slike
chdir C:\Files\Slike
mkdir Morje
chdir C:\
mkdir Temp
```

28. Kendallov koeficient

Na nekem tekmovanju je nastopilo n tekmovalcev, njihove nastope pa je ocenjevalo z sodnikov. Radi bi primerjali ocene sodnikov, da bi videli, če kateri izrazito odstopa od ostalih. Recimo, da primerjamo sodnika x in y, pri čemer označimo z x_i oz. y_i oceni, ki sta ju tadva sodnika dala tekmovalcu i. Rekli bomo, da tekmovalca i in j tvorita *neskladen par*, če je eden od sodnikov ocenil i-ja višje kot j-ja, drugi pa nižje (torej z drugimi besedami, če je $x_i > x_j \wedge y_i < y_j$ ali pa $x_i < x_j \wedge y_i > y_j$); če pa sta oba sodnika ocenila i višje ali pa oba nižje kot j, rečemo, da i in j tvorita *skladen par* (torej z drugimi besedami, če je $x_i > x_j \wedge y_i > y_j$ ali pa $x_i < x_j \wedge y_i < y_j$). Število skladnih parov označimo z $n_c(x, y)$, neskladnih pa $n_d(x, y)$. Število vseh možnih parov je seveda $n_0 = n(n-1)/2$. Za mero podobnosti med ocenama obeh sodnikov lahko zdaj uporabimo $\tau(x, y) = (n_c(x, y) - n_d(x, y))/n_0$; to nam dá neko število na območju od -1 do 1 (večji τ pomeni, da se ocene bolj ujemajo). **Opiši postopek**, ki ugotovi, kateri sodnik ima najmanjšo povprečno vrednost τ do vseh ostalih sodnikov.

Predpostavi, da je z majhen (na primer $z \leq 10$), število tekmovalcev pa je lahko veliko (npr. $n \leq 10^6$).

Malo lažjo različico naloge dobimo, če predpostavimo, da posamezen sodnik ne sme dati enake ocene dvema ali več različnim tekmovalcem. Še malo lažjo različico dobimo, če dodamo še omejitev, da sodniki za ocene uporabljajo le cela števila od 1 do n.

29. Kidanje

Na neki ulici je $n+1$ zasneženih dvorišč; pri tem je „dvorišče“ $n+1$ velik kup snega, kamor morajo stanovalci zmetati ves sneg s prvih n dvorišč. Človek, ki kida dvorišče i, je močan ravno toliko, da sneg vrže največ d_i dvorišč daleč (torej na dvorišče $i+1$ ali na $i+2$ ali ... ali $i+d_i$). Vsako jutro se najprej zbudi človek i_1, skida (in za tisti dan konča), potem se zbudi človek i_2, skida (in za tisti dan konča), itd.; nazadnje se zbudi in skida človek i_n. Vsak človek je zmožen v enem dnevu skidati neomejene količine snega (in to še preden se zbudi naslednji). **Opiši postopek**, ki ugotovi, v koliko dneh lahko počistijo ulico, če vsi mečejo sneg optimalno daleč (to ni nujno maksimalno daleč). Veljalo bo $n \leq 10^6$, $1 \leq d_i \leq n$.

REŠITVE NALOG ZA PRVO SKUPINO

1. Prepletene besede

Nalogo lahko rešimo tako, da naredimo dva prehoda po vhodnem nizu `s`; v prvem kopiramo v `t` velike črke, v drugem pa male črke. Podrobnosti so načeloma precej odvisne od tega, kako se dela z nizi v našem izbranem programskem jeziku. Spodnja rešitev je v C-ju in se s kazalcem `p` premika naprej po vhodnem nizu; vsakič, ko naleti na primerno črko (veliko v prvem prehodu, malo v drugem), pa jo izpiše v izhodni niz `t` in se premakne naprej tudi po njem (poveča kazalec `t`).

```
void Prepletene(const char *s, char *t)
{
  const char *p;
  for (p = s; *p; p++) /* skopirajmo velike črke */
    if ('A' <= *p && *p <= 'Z') *t++ = *p;
  for (p = s; *p; p++) /* skopirajmo male črke */
    if (! ('A' <= *p && *p <= 'Z')) *t++ = *p;
  *t = 0; /* dodajmo še znak za konec izhodnega niza */
}
```

Pri preverjanju, katere črke so male in katere velike, bi si lahko pomagali tudi s funkcijami iz standardne knjižnice našega programskega jezika; pri C-ju sta to `isupper` in `islower`.

2. Manjkajoča števila

Spodnja rešitev si v spremenljivki `prejsnje` hrani prejšnje prebrano število. To je tudi zadnje število, ki smo ga doslej izpisali. Ko preberemo naslednje število, recimo `n`, gremo z notranjo zanko od `prejsnje` + 1 do `n` − 1 in ta števila izpisujemo v oklepajih. Ob koncu te zanke je `prejsnje` enak `n`, nato pa moramo `n` le še izpisati (brez oklepajev) in smo že pripravljeni na branje naslednjega vhodnega števila.

Poseben primer nastopi pri prvem vhodnem številu: pred njim ne smemo izpisati nobenega manjkajočega števila. Ker vnaprej ne vemo, kakšno bo prvo vhodno število, postavimo `prejsnje` na začetku na −1 in na to možnost posebej pazimo pred izpisom manjkajočih števil.

```
#include <stdio.h>

int main()
{
  int prejsnje = -1, n;
  while (1 == scanf("%d", &n))
  {
    if (prejsnje < 0) prejsnje = n;
    else while (++prejsnje < n) printf("(%d)\n", prejsnje);
    printf("%d\n", n);
  }
  return 0;
}
```

3. Kazenski stavek

Recimo, da je vhodni niz dolg n znakov, kazenski stavek pa d znakov. Spodnja rešitev gre z zanko po naraščajočih d, pri vsakem preveri, če se dá vhodni niz razumeti kot zaporedje kazenskih stavkov dolžine d (ločenih s po enim presledkom), in vrne prvi (torej najmanjši) tak d, ki ustreza temu pogoju.

Če je vhodni niz sestavljen iz p kopij kazenskega stavka dolžine d, mora biti $n = p \cdot d + (p-1)$, saj naloga pravi, da so posamezne kopije kazenskega stavka med seboj ločene s po enim presledkom. Ta pogoj lahko zapišemo tudi kot $(n+1) = p \cdot (d+1)$; z drugimi besedami, v poštev pridejo le takšne dolžine d, pri katerih je $n+1$ večkratnik števila $d+1$. Tako lahko večino d-jev takoj zavržemo.

Naslednji pogoj, ki ga je koristno preveriti, je, če je znak s[d] presledek — če imamo opravka s kazenskim stavkom dolžine d, mora prva pojavitev tega stavka pokrivati znake s[0], ..., s[d − 1], nato pa mora biti v s[d] presledek, ki ločuje prvo pojavitev kazenskega stavka od druge.

Nato moramo le še preveriti, če se isti kazenski stavek ponavlja tudi v preostanku niza; znak s[d + 1] mora biti enak s[0], znak s[d + 2] mora biti enak s[1] in tako naprej. To preverimo v notranji zanki; če ta uspešno pride do konca niza s, ne da bi našla kakšno neujemanje, potem vemo, da smo našli primerno dolžino kazenskega stavka. Število pojavitev (to je namreč tisto, po čemer sprašuje naloga) pa je potem, kot smo videli pri zgornji formuli, $(n+1)/(d+1)$.

```
#include <stdlib.h>

int KazenskiStavek(char *s)
{
  int n = strlen(s), d, i;
  for (d = 1; d < n; d++)
  {
    /* Preverimo, ali je n + 1 večkratnik d + 1 in ali je na pravem mestu presledek. */
    if ((n + 1) % (d + 1) != 0) continue;
    if (s[d] != ' ') continue;
    /* Preverimo, ali se prvih d + 1 znakov pojavlja po celem nizu. */
    for (i = d + 1; i < n; i++)
      if (s[i] != s[i % (d + 1)]) break;
    if (i == n) break; /* Čim najdemo primeren d, končajmo. */
  }
  /* Če glavna zanka ni našla ničesar pametnega,
     imamo zdaj d = n in bomo vrnili 1. */
  return (n + 1) / (d + 1);
}
```

Rešitev lahko potencialno izboljšamo s še nekaj dodatnimi pogoji, ki jih je poceni preverjati in nam lahko pomagajo hitreje odkriti morebitna neujemanja.

(1) Na primer, zadnji znak niza s mora biti enak zadnjemu znaku (domnevnega) kazenskega stavka, torej $s[n-1] = s[d-1]$. Če ta enakost ne drži, lahko nad tem d takoj obupamo in se lotimo naslednjega.

(2) Označimo s $p = (n+1)/(d+1)$ število pojavitev kazenskega stavka v nizu s (če je kazenski stavek res dolžine d). Recimo, da se neka črka c pojavi f_c-krat v kazenskem stavku; potemtakem se c pojavi $(p \cdot f_c)$-krat v celem nizu s. Če se torej za neko črko c zgodi, da število pojavitev te črke v s ni večkratnik števila p, lahko

takoj zaključimo, da trenutni d ne more biti dolžina kazenskega stavka. Ta pogoj je najlažje preverjati tako, da pred glavno zanko (tisto po d) preštejemo pojavitve vseh znakov[2] v nizu s in izračunamo največji skupni delitelj vseh teh števil pojavitev. Tako bomo morali kasneje preverjati le, ali p deli tega največjega skupnega delitelja.

Namesto s posameznimi črkami bi lahko ta razmislek ponovili tudi z daljšimi podnizi, npr. pari črk ali celimi besedami, važno je le, da ne vsebujejo presledkov (ker bi se v tem primeru lahko zgodilo, da podniz ne leži v celoti znotraj ene pojavitve kazenskega stavka, torej ni več lepe povezave med številom njegovih pojavitev v kazenskem stavku in v celem nizu s).

(3) Dodajmo nizu s v mislih še presledek na koncu, kot znak $s[n]$. Recimo, da zdaj razmišljamo o kazenskem stavku dolžine d in da smo že preverili, da je $s[d]$ presledek. V mislih razdelimo naš podaljšani s na $p = (n+1)/(d+1)$ kosov dolžine $d+1$ in jih zapišimo: $s = w_1 w_2 \ldots w_p$. Da preverimo, ali je naš kazenski stavek res dolžine d, moramo pravzaprav preveriti, ali so ti kosi vsi enaki, torej ali je $w_1 = w_2 = \ldots = w_p$. Ta pogoj pa je enakovreden pogoju $w_1 w_2 \ldots w_{p-1} = w_2 w_3 \ldots w_p$.

Izberimo si neko primerno razprševalno funkcijo, torej funkcijo, ki vsakemu nizu x priredi neko celo število $h(x)$ (ki mu pravimo tudi razprševalna koda niza x). Preden primerjamo niza $w_1 w_2 \ldots w_{p-1}$ in $w_2 w_3 \ldots w_p$, je koristno primerjati njuni razprševalni kodi; če tidve nista enaki, potem tudi niza ne moreta biti enaka (in se lahko izognemo primerjanju znak po znak).

Vprašanje je le, kako lahko dovolj učinkovito računamo takšne razprševalne kode. Koristno je uporabiti razprševalno funkcijo iz Rabin-Karpovega algoritma; izberimo si pozitivni celi števili a in M ter za niz $x[0, \ldots, k-1]$ definirajmo $h(x) := (\sum_{i=0}^{k-1} a^i x[i]) \bmod M$. Tako definirana razprševalna funkcija ima nekaj lepih lastnosti, zaradi katerih jo je lažje računati. Če niz x podaljšamo z znakom c, velja $h(xc) = (h(x) + a^k c) \bmod M$; če c vrinemo na začetek niza, pa velja $h(cx) = (c + a \cdot h(x)) \bmod M$. O obojem se zlahka prepričamo iz definicije funkcije h. S pomočjo teh zakonitosti lahko v $O(n)$ časa izračunamo razprševalne kode za vse možne začetke (prefikse) in konce (sufikse) niza s; moramo jih le shraniti v neko tabelo, pa bodo prišli prav kasneje pri preverjanju pogoja $h(w_1 w_2 \ldots w_{p-1}) = h(w_2 w_3 \ldots w_p)$. Niza v tem pogoju sta namreč tudi prefiks in sufiks s-ja.

Če se ta test izide, bi si lahko pred podrobnim primerjanjem znak po znak privoščili še en test z razprševalnimi kodami: preverili bi lahko, ali so tudi razprševalne kode vseh w_i enake, torej ali je $h(w_1) = h(w_2) = \ldots = h(w_p)$. Če bi si hoteli vnaprej izračunati razprševalne kode vseh možnih podnizov s-ja in si jih shraniti, bi to porabilo preveč časa in pomnilnika, zato si lahko pomagamo z naslednjo zakonitostjo (ki tudi sledi neposredno iz definicije naše razprševalne funkcije h): če imamo niz x dolžine k in še nek niz y, potem velja $h(xy) = (h(x) + a^k h(y)) \bmod M$. Če si a in M izberemo tako, da obstaja multiplikativni inverz a^{-1} po modulu M, lahko $h(y)$ izračunamo preprosto kot $h(y) = (a^{-1})^k (h(xy) - h(x)) \bmod M$.[3] Tako bomo

[2] Razen presledkov; če bi hoteli v ta razmislek vključiti še presledke, bi morali njihovo število v nizu s povečati za 1, kajti če se presledek pojavi f-krat v kazenskem stavku, ta stavek pa p-krat v nizu s in so pojavitve kazenskega stavka v s ločene s po enim presledkom (kot zahteva naloga), potem je število presledkov v nizu s enako le $p \cdot f + (p-1)$. Če pa ga povečamo za 1, res dobimo večkratnik števila p, namreč $p \cdot (f+1)$.

[3] Kako poiščemo multiplikativni inverz števila a po modulu M? Označimo ta inverz z u. Enačbo $a \cdot u \equiv 1 \pmod{M}$ predelajmo v $au + Mv = 1$; to je linearna diofantska enačba in jo lahko rešujemo na primer z razširjenim Evklidovim algoritmom. Med pari (u, v), ki rešijo to

lahko za vsak w_i poceni izračunali kodo $h(w_i)$ s pomočjo kod za $h(w_1w_2\ldots w_{i-1})$ in $h(w_1w_2\ldots w_{i-1}w_i)$.

4. Mase

Rekli bomo, da neka vrsta izdelkov z maso m *pokrije* meritev x natanko tedaj, ko bi lahko meritev x nastala pri tehtanju izdelka te vrste; z drugimi besedami je to takrat, ko je $m - a \le x \le m + a$. Naloga tako pravzaprav zahteva, da vse dane meritve pokrijemo s čim manj izdelki.

Oglejmo si najmanjšo meritev; recimo ji x_1. Pokril bi jo na primer izdelek z maso $x_1 + a$; tak izdelek pokrije celoten interval $[x_1, x_1 + 2a]$. Ali je smiselno razmišljati o rešitvah, pri katerih je najlažji izdelek kaj lažji od $x_1 + a$? Ne, kajti če maso izdelka zmanjšamo pod $x_1 + a$, se interval mas, ki ga pokrije ta izdelek, pomakne navzdol; na spodnjem robu s tem ničesar ne pridobimo (saj meritev, manjših od x_1, sploh nimamo), na zgornjem robu pa mogoče kaj izgubimo (novi izdelek z manjšo maso mogoče ne pokrije nekaterih meritev, ki jih je izdelek $x_1 + a$ pokril). Tako torej vidimo, da ni nobene koristi od tega, da bi bil najlažji izdelek kaj lažji od $x_1 + a$. Po drugi strani pa najlažji izdelek ne sme biti težji od $x_1 + a$, saj bi drugače x_1 ostal nepokrit. Vzemimo torej izdelek z maso točno $x_1 + a$, iz zaporedja meritev pobrišimo tiste, ki jih je ta izdelek pokril (vse do vključno $x_1 + 2a$) in nadaljujmo po enakem postopku, dokler ni pokrito celotno zaporedje.

Zapišimo ta postopek še s psevdokodo:

naj bodo $x_1, \ldots, x_n$ vhodne meritve (urejene naraščajoče);
$i := 0$; $k := 0$; (* *i je števec po meritvah, k je število izdelkov* *)
while $i \le n$:
  $k := k + 1$; $m_k := x_i + a$; (* *nova vrsta izdelka z maso* m_k *)
  (* *Preskočimo meritve, ki jih pokrije novi izdelek.* *)
  while $i < n$ **and** $x_i \le m_k + a$ **do** $i := i + 1$;
return k;

5. V Afganistan!

Naloga pravi, da je vseeno, če poleg gajstnih vojakov pošljemo v Afganistan še kakšnega ne-gajstnega. Potemtakem ni nobene koristi od tega, da bi poslali ukaz vojaku x, če ima le-ta tudi nadrejenega, saj bi lahko ukaz poslali raje temu nadrejenemu in bi tako spravili v Afganistan tudi vojaka x (in vse njegove posredno in neposredno podrejene). Torej je koristno, če se od vsakega gajstnega vojaka sprehodimo gor po hierarhiji, dokler ne pridemo do vrha; vse tako obiskane vojake označimo (v spodnjem programu imamo zato tabelo poslji). Na koncu tega postopka imamo tako v tej tabeli za vsakega vojaka logično vrednost, ki nam pove, če mora on ali kdo od njegovih (lahko posredno) podrejenih v Afganistan. Ukaze pošljimo tistim vojakom, pri katerih je ta pogoj izpolnjen in ki nimajo svojega nadrejenega.

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

enačbo, moramo vzeti tistega, ki ima u na območju od 0 do $M - 1$. Da bo enačba rešljiva, si morata biti a in M tuja.

```
int main()
{
  int i, v, stUkazov;

  /* V tabeli poslji bomo označili gajstne vojake in vse njihove
     nadrejene (posredne in neposredne). */
  bool *poslji = (bool *) malloc(sizeof(bool) * n);
  for (i = 0; i < n; i++) poslji[i] = false;

  for (i = 0; i < n; i++)
    /* Če je i gajsten in ga v tabeli poslji se nismo oznacili,
       moramo zdaj označiti njega in njegove nadrejene.
       Če pa smo ga ze kdaj prej oznacili (npr. ker ima kakšnega
       gajstnega podrejenega), smo takrat označili tudi njegove
       nadrejene in se nam zdaj z njim ni treba še enkrat ukvarjati. */
    if (Gajsten(i) && ! poslji[i])
      for (v = i; v >= 0; v = Nadrejeni(v)) poslji[v] = true;

  /* Ukaze moramo poslati tistim, ki so označeni v tabeli poslji
     in ki nimajo svojega nadrejenega. */
  for (i = 0, stUkazov = 0; i < n; i++)
    if (poslji[i] && Nadrejeni(i) < 0) stUkazov++;

  printf("%d\n", stUkazov); free(poslji); return 0;
}
```

Označevanje v tabeli poslji ima še eno korist: ko se premikamo navzgor po hierarhiji, se lahko ustavimo, čim naletimo na nekega že označenega nadrejenega, saj takrat vemo, da smo njega in vse njegove nadrejene že označili nekoč prej, ko smo se ukvarjali z nekim prejšnjim gajstnim vojakom. To pomeni, da se bo notranja zanka (po v) iz nekega vojaka v njegovega nadrejenega premaknila največ enkrat (namreč takrat, ko bo tega nadrejenega prvič označila). Skupno število iteracij notranje zanke je torej le $O(n)$, zato je časovna zahtevnost našega postopka le $O(n)$, ne glede na to, kakšne oblike je hierarhija (npr. kako globoka je). Brez te izboljšave, torej če bi se pri vsakem gajstnem vojaku sprehodili vse do vrha njegove hierarhije, bi lahko naš postopek pri neugodno oblikovani hierarhiji porabil do $O(n^2)$ časa (npr. če je hierarhija izrojena v seznam, torej ima n nivojev in vsak vojak ima največ enega neposredno podrejenega).

REŠITVE NALOG ZA DRUGO SKUPINO

1. Ovce

Če damo strižcu i postriči eno ovco, je skupni strošek dela in elektrike enak $p_i + c \cdot t_i$; recimo temu q_i. Ni se težko prepričati, da je koristno dati največ dela tistim strižcem, pri katerih je ta skupni strošek q_i najmanjši. Recimo namreč, da imamo dva strižca, i in j, pri čemer je $q_i < q_j$; potem, če damo j-ju ostriči eno ovco manj in i-ju eno ovco več, bo skupni strošek manjši za $q_j - q_i$. Pri tem moramo paziti le na to, da posamezni strižec ne sme dobiti več ovac, kot jih bo lahko v t časa ostrigel, torej $\lfloor t/t_i \rfloor$. Zapišimo postopek še s psevdokodo:

za vsakega strižca izračunajmo $q_i = p_i + c \cdot t_i$
$c := 0$; (* *skupna cena striženja* *)
pregledujmo strižce po naraščajočem vrstnem redu q_i:
  naj bo i trenutni strižec;
  (* *Spremenljivka n na tem mestu vsebuje število ovac, ki jih še nismo dodelili nobenemu strižcu.* *)
  $n_i := \min\{n, \lfloor t/t_i \rfloor\}$; (* *ta strižec naj postriže n_i ovac* *)
  $c := c + n_i \cdot q_i$; $n := n - n_i$;

Če pade med izvajanjem te zanke n na 0, to pomeni, da smo razdelili že vse ovce in lahko takoj končamo. Če pa bi se zgodilo, da bi prišli do konca strižcev, n pa bi bil še vedno večji od 0, bi pomenilo, da vseh ovac v času t sploh ne bomo mogli ostriči; vendar naloga zagotavlja, da do tega ne bo prišlo.

2. Spričevala

Koristno si je v neki tabeli za vsakega ravnatelja zapisovati, na kateri šoli dela; v spodnjem programu uporabljamo za to tabelo rs. Ko prvič opazimo spričevalo s podpisom nekega ravnatelja, vpišemo številko šole v ustrezno celico tabele rs; če nato kasneje pri nekem drugem spričevalu opazimo podatek, da ta ravnatelj dela na neki drugi šoli, kot piše v tisti tabeli, pa vemo, da moramo izpisati opozorilo.

Razmisliti pa moramo še o dveh podrobnostih. Ena je, da moramo opisano tabelo po vsakem letu pobrisati (kajti če isti ravnatelj dela na različnih šolah v različnih letih, to še ni razlog za izpisovanje opozoril). Temu se lahko izognemo tako, da imamo še eno tabelo, v kateri za vsakega ravnatelja piše, v katerem letu smo ga nazadnje videli; v spodnjem programu je to tabela rl. Potem vemo, da moramo podatek za tega ravnatelja v tabeli rs ignorirati, če se ne nanaša na trenutno leto.

Druga podrobnost je, da nočemo enakega opozorila izpisovati po večkrat. Moramo si torej nekako zapomniti, da smo za nekega ravnatelja v trenutnem letu že izpisali opozorilo. Spodnji program si to zapomni tako, da v ustrezno celico tabele rs zapiše -1.

Doslej smo razmišljali o opozorilih za ravnatelje, enak razmislek pa bi lahko uporabili tudi za šole (da opozorimo, če ima ista šola v istem letu več ravnateljev), le vloge šol in ravnateljev so obrnjene. Spodnji program ima v ta namen še dve tabeli, sr in sl.

```
#include <stdio.h>

#define MaxS 1000000
#define MaxR 1000000

int rl[MaxR], rs[MaxR], sl[MaxS], sr[MaxS];

int main()
{
  int nSol, nRavnateljev, n, s, r, leto;
  scanf("%d %d %d", &nSol, &nRavnateljev, &n);
  /* Inicializirajmo tabele. */
  for (r = 0; r < nRavnateljev; r++) rl[r] = -1, rs[r] = -1;
  for (s = 0; s < nSol; s++) sl[s] = -1, sr[s] = -1;
  while (n-- > 0) /* Obdelajmo podatke o spričevalih. */
  {
    scanf("%d %d %d", &leto, &s, &r);
    if (leto != sl[s]) sl[s] = leto, sr[s] = r;
    else if (sr[s] >= 0 && r != sr[s]) {
      sr[s] = -1; printf("Preveri spricevala sole st. %d v letu %d.\n", s, leto); }
    if (leto != rl[r]) rl[r] = leto, rs[r] = s;
    else if (rs[r] >= 0 && s != rs[r]) {
      rs[r] = -1; printf("Preveri spricevala s podpisom ravnatelja "
                         "st. %d v letu %d.\n", r, leto); }
  }
  return 0;
}
```

3. Razpolovišče lika

Najprej preglejmo celotno mrežo in preštejmo, koliko je črnih polj; tako dobimo površino kraljestva. Nato preglejmo mrežo še enkrat, po vrsticah. Recimo, da je celotna površina kraljestva p, v dosedanjih vrsticah (pred trenutno) smo našteli d črnih polj, v trenutni vrstici pa v črnih polj. Mejo moramo potegniti v tisti vrstici, v kateri $d+v$ prvič doseže ali preseže $p/2$. Recimo, da pokriva trenutna vrstica na y-osi interval $[y, y+1]$ in da potegnemo mejo na višini $y+t$ (za nek $t \in [0,1]$). Južno od meje torej ostane $d+t\cdot v$ enot površine; to mora biti enako $p/2$, če hočemo površino res razpoloviti. Tako imamo $d+tv = p/2$ oz. $t = (p/2-d)/v$ oz. $t = (p-2d)/(2v)$.

Spodnji podprogram hrani d v spremenljivki pDoslej, vrednost v pa hrani v spremenljivki pVrstice.

```
double Razpolovisce()
{
  int povrsina = 0, pDoslej, pVrstice, x, y;
  /* V prvem prehodu izračunajmo površino kraljestva. */
  for (y = 0; y < h; y++) for (x = 0; x < w; x++)
    if (Znotraj(x, y)) povrsina++;
  /* V drugem prehodu določimo položaj meje. */
  for (y = 0, pDoslej = 0; y < h; y++)
  {
    for (x = 0, pVrstice = 0; x < w; x++)
      if (Znotraj(x, y)) pVrstice++;
```

```
      /* Ali smo (skupaj s trenutno vrstico) že dosegli polovico površine? */
      if (2 * (pDoslej + pVrstice) >= povrsina)
        /* Izračunajmo točno višino razmejitvene črte. */
        return y + (povrsina − 2 * pDoslej) / (2.0 * pVrstice);
      pDoslej += pVrstice;
    }
}
```

4. Strukturirani podatki

Trenutno odprte značke si je koristno shranjevati na nekakšnem skladu (ki je lahko predstavljen v tabeli ali pa kot seznam, povezan s kazalci). Vhodno datoteko beremo po vrsticah; ko naletimo na začetno značko, jo dodamo na sklad; ko naletimo na končno značko, jo pobrišemo z vrha sklada; ko pa naletimo na običajno vrstico besedila, se sprehodimo po skladu (od dna proti vrhu) in izpišemo imena značk s sklada (ločena s pikami), na koncu pa še trenutno vrstico besedila.

Spodnja rešitev ima sklad predstavljen kot seznam, povezan s kazalci, čisto dobra rešitev pa bi bila tudi s tabelo 100 elementov (saj naloga pravi, da značke ne bodo gnezdene več kot 100 nivojev globoko).

```
#include <stdio.h>
#include <stdlib.h>
#define MaxDolz 100

typedef struct Znacka_ {
  char *znacka;
  struct Znacka_ *nasl, *prej;
} Znacka;

int main()
{
  char buf[MaxDolz + 2], *p, n;
  Znacka *sklad = 0, *vrh = 0, *z;
  while (fgets(buf, sizeof(buf), stdin))
  {
    /* Preberimo naslednjo vrstico, preskočimo presledke na začetku in
       odrežimo znak za konec vrstice na koncu. */
    p = &buf[0]; while (*p == ' ') p++;
    n = strlen(p); if (n > 0 && p[n − 1] == '\n') p[−−n] = 0;

    /* Če imamo začetno značko, jo dodajmo (brez znaka +) na vrh sklada. */
    if (*p == '+') {
      z = (Znacka *) malloc(sizeof(Znacka));
      z−>znacka = (char *) malloc(n);
      strcpy(z−>znacka, p + 1);
      z−>prej = vrh; z−>nasl = 0;
      if (vrh) vrh−>nasl = z; else sklad = z;
      vrh = z; }

    /* Če imamo končno značko, pobrišimo zadnji zapis z vrha sklada. */
    else if (*p == '-') {
      free(vrh−>znacka); z = vrh−>prej;
      free(vrh); vrh = z; if (! vrh) sklad = 0; else vrh−>nasl = 0; }

    /* Sicer pa se sprehodimo po skladu, izpišimo trenutno gnezdene značke
       in za njimi še besedilo iz trenutne vrstice. */
```

```
      else {
        for (z = sklad; z; z = z->nasl)
          printf("%s%s", z == sklad ? "" : ".", z->znacka);
        printf(" %s\n", p); }
    }
    /* Če je dokument sintaktično pravilen, mora biti sklad zdaj prazen. */
    return 0;
}
```

5. Največji pretok

Program teče v neskončni zanki in v vsaki iteraciji poskuša prebrati meritev (iz kanala M) in maksimuma, ki mu ju pošiljata njegov desni in spodnji sosed (prek kanalov D in S). Največjo doslej prebrano vrednost hrani v spremenljivki max in jo v vsaki iteraciji pošlje svojemu zgornjemu in levemu sosedu. Tako pri vsakem računalniku max hrani največjo doslej prebrano vrednost v celotnem delu mreže, ki leži spodaj in desno od njega. Računalnik v zgornjem levem kotu mreže tako sčasoma dobi največjo meritev iz cele mreže in jo izpiše na svoj levi kanal, kjer jo bomo lahko prebrali.

```
int main()
{
    int x, max = 0;
    for ( ; ; )
    {
        if ((x = Beri('M')) > max) max = x;
        if ((x = Beri('D')) > max) max = x;
        if ((x = Beri('S')) > max) max = x;
        Pisi('Z', max);
        Pisi('L', max);
    }
}
```

Primera, ko kakšno branje ne uspe, nam ni treba preverjati posebej, saj naloga pravi, da Beri takrat vrne -1, to pa na naš maksimum ne bo vplivalo (saj je max že od vsega začetka ≥ 0).

REŠITVE NALOG ZA TRETJO SKUPINO

1. de-FFT permutacija

Naloge se lahko lotimo z rekurzijo. Recimo, da imamo kot šifrirano sporočilo nek niz t, dolg 11 znakov. Ker se ob šifriranju s funkcijo FFT dolžina sporočila ne spreminja, je moralo biti tudi prvotno sporočilo (recimo s) dolgo 11 znakov. Iz definicije funkcije FFT vidimo, da bi ta iz niza 11 znakov najprej naredila dva niza, enega dolžine 6 (namreč $s_1 s_3 \dots s_{11}$) in enega dolžine 5 (namreč $s_2 s_4 \dots s_{10}$), zakodirala vsakega od njiju in tako dobljena šifrirana niza spet staknila. Torej tvori prvih 6 znakov niza t ravno $\text{FFT}(s_1 s_3 \dots s_{11})$, preostalih 5 znakov niza t pa tvori niz $\text{FFT}(s_2 s_4 \dots s_{10})$. Torej lahko z rekurzivnim klicem najprej dešifriramo $t_1 \dots t_6$, da dobimo $s_1 s_3 \dots s_{11}$, in dešifriramo $t_7 \dots t_{11}$, da dobimo $s_2 s_4 \dots s_{10}$; potem pa moramo njune znake le še preplesti med seboj, da dobimo prvotni niz $s_1 s_2 \dots s_{10} s_{11}$. Enak razmislek seveda deluje pri vsaki dolžini, ne le 11; če imamo niz dolžine d, ga moramo razbiti na prvi del dolžine $\lceil d/2 \rceil$ in drugi del dolžine $\lfloor d/2 \rfloor$. Robni primer rekurzije pa nastopi pri $d \leq 2$, saj pri tako kratkih nizih vemo, da sta šifrirano in prvotno sporočilo enaki.

Zelo elegantno pa lahko nalogo rešimo tudi z naslednjim razmislekom. Za potrebe tega razmisleka bo lažje, če bomo položaje znakov v nizu šteli od 0 naprej namesto od 1 naprej. Recimo zdaj, da imamo nek niz s dolžine d, vendar ga pred šifriranjem še toliko podaljšajmo (z nekaj posebnimi znaki, ki se v njem sicer ne pojavljajo; recimo `#`), da bo njegova dolžina potenca števila 2 (recimo 2^b za $b = \lceil \log_2 d \rceil$). Tako podaljšani niz (recimo mu $\hat{s}$) zašifrirajmo in dobimo $\hat{t} = \text{FFT}(\hat{s})$. Kje v nizu $\hat{t}$ je pristal posamezni znak niza $\hat{s}$, recimo s_i? Izkaže se, da če i zapišemo kot b-bitno število v dvojiškem zapisu in potem te bite preberemo od desne proti levi, dobimo ravno indeks, na katerem je v $\hat{t}$ po šifriranju pristal znak s_i.

Primer: vzemimo $b = 3$, torej je niz $\hat{s}$ dolg $2^b = 8$ znakov in ga lahko pišemo kot $s_0 s_1 \dots s_7$. Potem je

$$\begin{aligned} \hat{t} = \text{FFT}(\hat{s}) &= \text{FFT}(s_0 s_2 s_4 s_6)\,\text{FFT}(s_1 s_3 s_5 s_7) \\ &= \text{FFT}(s_0 s_4)\,\text{FFT}(s_2 s_6)\,\text{FFT}(s_1 s_5)\,\text{FFT}(s_3 s_7) \\ &= s_0 s_4 s_2 s_6 s_1 s_5 s_3 s_7. \end{aligned}$$

Če zdaj za vsak znak niza $\hat{s}$ pogledamo, na katerem mestu v nizu $\hat{t}$ je pristal, vidimo:

Indeks znaka pred šifriranjem:	0	1	2	3	4	5	6	7
oz. dvojiško:	000	001	010	011	100	101	110	111
Indeks po šifriranju:	0	4	2	6	1	5	3	7
oz. dvojiško:	000	100	010	110	001	101	011	111

Vidimo torej, da indeks znaka po šifriranju res vsakič dobimo tako, da indeks pred šifriranjem zapišemo kot b-mestno dvojiško število in nato vrstni red bitov obrnemo. Z indukcijo po b bi se lahko prepričali, da velja ta zakonitost pri vsakem b, ne le pri $b = 3$; podrobnosti tega dokaza prepuščamo bralcu za vajo.

Niz $\hat{t}$ je torej zelo enostavno dešifrirati v $\hat{s}$: vsak indeks i od 0 do $2^b - 1$ zapišemo kot b-bitno število, mu bite obrnemo in dobimo novi indeks, na katerega moramo v $\hat{s}$ zapisati t_i.

Toda naš program kot vhodni niz seveda ne dobi $\hat{t}$, pač pa $t = \text{FFT}(s)$, torej šifro prvotnega nepodaljšanega niza s. Na srečo se izkaže (tudi o tem se lahko prepričamo

z indukcijo po b), da če iz $\hat{t}$ pobrišemo vse znake `#`, dobimo ravno t. Torej si lahko $\hat{t}$ sestavimo „v mislih“: za vsak indeks i od 0 do 2^b-1 si z obračanjem bitov izračunamo, iz katerega indeksa v nizu $\hat{s}$ bi prišel i-ti znak niza $\hat{t}$; če je ta indeks $\geq d$, vemo, da ima $\hat{t}$ tukaj znak `#`, sicer pa ima tukaj naslednji znak niza t.

```
#include <stdio.h>

#define MaxDolz 1024

int main()
{
  FILE *f = fopen("defft.in", "rt"), *g = fopen("defft.out", "wt");
  char s[MaxDolz + 1], t[MaxDolz + 3];
  int n, d, b, i, is, it, bb;
  fscanf(f, "%d\n", &n);
  while (n-- > 0)
  {
    fgets(t, sizeof(t) - 1, f);
    /* Naj bo d dolžina niza t; poiščimo tudi prvo potenco števila 2, ki je ≥ d. */
    d = 0; while (t[d] == '_' || ('a' <= t[d] && t[d] <= 'z')) d++;
    b = 0; while (d > (1 << b)) b++;
    /* Dešifrirajmo niz t v niz s. */
    s[d] = 0;
    for (i = 0, it = 0; i < (1 << b); i++)
    {
      /* Spremenljivka i označuje naš položaj v hipotetičnem nizu t̂, ki je dolg 2^b znakov
         in iz katerega dobimo t tako, da v njem pobrišemo vse znake #. Če pogledamo i
         kot b-bitno število in obrnemo vrstni red bitov v njem, dobimo indeks is, ki pove,
         iz katerega znaka niza ŝ je nastal ta znak niza t̂. */
      for (bb = 0, is = 0; bb < b; bb++)
        if (i & (1 << bb)) is |= 1 << (b - 1 - bb);
      /* Če je is ≥ d, to pomeni, da je v nizu ŝ tam znak #; ta je torej tudi
         na indeksu i v nizu t̂, v nizih s in t pa tega znaka ni. */
      if (is >= d) continue;
      /* Sicer pa imamo opravka z znakom, ki se pojavlja tudi v nizu t, in sicer na
         indeksu it. Prepišimo ga na pravo mesto (indeks is) niza s in se premaknimo
         naprej tudi po t (in ne le po t̂ — po njem se premaknemo v vsakem primeru). */
      s[is] = t[it++];
    }
    fprintf(g, "%s\n", s);
  }
  fclose(f); fclose(g); return 0;
}
```

2. Potovanje

Naj bo p_i najbolj vzhodna črpalka, ki jo lahko dosežemo, če začnemo vožnjo na črpalki i s praznim tankom; in naj bo h_i količina goriva, s katero dosežemo to črpalko (preden tankamo na njej). Z „najbolj vzhodna“ mislimo to, da tudi če potem na p_i načrpamo vseh g_i enot goriva, ki so tam na voljo, še vseeno ne bomo mogli doseči črpalke $p_i + 1$. Dolžina celotnega potovanja bo v tem primeru $x_{p_i} - x_i + g_i + h_i$. Vprašanje je le še, kako to učinkovito izračunati za vse i.

Izkaže se, da je vrednosti p_i in h_i koristno računati od vzhoda proti zahodu. Recimo, da začnemo pri i s praznim tankom; seveda lahko takoj natočimo g_i enot goriva. Če je $x_{i+1} - x_i > g_i$, ne bomo mogli doseči niti naslednje črpalke, torej je $p_i = i$, $h_i = 0$ in smo s tem i končali. Če pa lahko črpalko $i + 1$ dosežemo (in to z $g' := g_i - (x_{i+1} - x_i)$ goriva v tanku), si lahko v nadaljevanju pomagamo s prej izračunanimi rezultati: če bi začeli v $i + 1$ s praznim tankom, bi lahko prišli do p_{i+1} s h_{i+1} goriva; mi pa smo prišli do $i + 1$ z g' goriva, ne s praznim tankom, torej bomo prišli lahko do p_{i+1} s kar $g'' := g' + h_{i+1}$ goriva. Tam se bomo potem lahko naprej ukvarjali s tem, kako daleč se da priti, če začnemo v p_{i+1} z g'' goriva.

O časovni zahtevnosti tega postopka lahko razmislimo takole. Tabela p_i nam definira kup „skokov“ $i \to p_i$, torej predstavlja nekakšen graf. Na vsaki iteraciji zunanje zanke (tiste, ki pregleduje začetne postaje i od vzhoda proti zahodu) dodamo v graf eno povezavo, $i \to (i + 1)$, nato pa izračunamo p_i in celotno verigo povezav $i \to (i + 1) \to p_{i+1} \to \ldots \to p_i$ zamenjamo z enim samim skokom $i \to p_i$. V nadaljevanju namreč teh vmesnih povezav ne bomo nikoli več uporabili, saj bomo vedno, ko bomo prišli do i, od tam nemudoma skočili na p_i. V vsaki iteraciji dodamo le $O(1)$ povezav, torej je z dodajanjem povezav skupno le $O(n)$ dela; in ker gremo po vsaki povezavi največ enkrat (saj jo bomo takoj zatem pobrisali), je tudi s sprehajanjem po povezavah le $O(n)$ dela. Časovna zahtevnost tega postopka je torej le $O(n)$.

```
#include <stdio.h>

#define MaxX 1000000000
#define MaxG 1000000000
#define MaxN 1000000

/* Črpalka i je na položaju xi[i] in na njej je gi[i] goriva. */
int xi[MaxN], gi[MaxN];

/* Če začnemo na črpalki i s praznim tankom, je najbolj desna črpalka, ki jo še lahko
   dosežemo, črpalka najP[i], do nje pa pridemo z najG[i] enotami goriva v tanku. */
int najP[MaxN];
long long najG[MaxN];

int main()
{
  int i, p, n; long long g;

  FILE *f = fopen("potovanje.in", "rt");
  fscanf(f, "%d\n", &n);
  for (i = 0; i < n; i++) fscanf(f, "%d %d\n", &xi[i], &gi[i]);
  fclose(f);

  for (i = n - 1; i >= 0; i--)
  {
    najP[i] = i; najG[i] = 0;
    p = i; g = 0;
    while (p < n - 1)
    {
      /* Ali je znan skok od p do najP[p]? */
      if (najP[p] > p) { g += najG[p]; p = najP[p]; continue; }

      /* Če ne, ali se lahko vsaj pripeljemo od p do p + 1? */
      if (xi[p + 1] - xi[p] > g + gi[p]) break;
      g = g + gi[p] - (xi[p + 1] - xi[p]); p++;
```

```
    }
    najP[i] = p; najG[i] = g;
  }

  f = fopen("potovanje.out", "wt");
  for (i = 0; i < n; i++) fprintf(f, "%lld\n", najG[i] + xi[najP[i]] − xi[i] + gi[najP[i]]);
  fclose(f); return 0;
}
```

3. Leteči pujsi

Strnjeni skupini opek, ki ležijo v isti vrstici in jih z leve in desne omejujejo prazna polja ali robovi mreže, bomo rekli *prečka*. Če so vse opeke na prečki stabilne, bomo rekli, da je tudi cela prečka stabilna. V začetnem stanju mreže so bile torej stabilne vse prečke. Pri opekah pravi naloga, da lahko opeka dobi svojo stabilnost z leve, z desne ali pa od spodaj; prečka pa dobi svojo stabilnost le od spodaj, saj z leve in desne meji na prosta polja (ali na rob mreže). Prečka je torej stabilna le, če jo od spodaj podpira vsaj ena druga stabilna prečka (ali pa če leži v najnižji vrstici mreže).

Recimo, da se pujs zaleti v polje (x, y). Če je to polje prosto, se ne zgodi nič zanimivega in lahko takoj končamo. Recimo torej, da je to polje opeka; doslej je bila, tako kot vse opeke na mreži, stabilna, zdaj pa jo je pujs destabiliziral. Kako to vpliva na preostanek mreže?

Ker se stabilnost širi le gor, levo in desno, ne pa tudi dol, so opeke na nižjih vrsticah (pod y) še vedno stabilne. Opeka (x, y) pripada neki prečki, ki leži v vrstici y in pokriva recimo interval $p_l \ldots p_d$. Doslej je bila cela prečka stabilna, zdaj pa polje (x, y) ni več stabilno; kako to vpliva na stabilnost preostanka prečke? Oglejmo si na primer del desno od (x, y). Če je ta del prečke, od $x + 1$ do p_d, spodaj podprt s kakšnimi drugimi opekami, je še vedno stabilen; če pa ne, je doslej svojo stabilnost dobival prek opeke (x, y) in je zdaj ta del prečke v celoti nestabilen. Podobno lahko razmišljamo tudi za levi del prečke, od p_l do $x - 1$. Rezultat je torej ta, da je v vrstici y destabiliziran nek interval opek (ki se začne pri p_l ali x in se konča pri x ali p_d). Druge prečke v vrstici y pa so gotovo še vedno stabilne, saj so morale svojo stabilnost dobiti od spodaj, v spodnjih vrsticah pa se ni nič spremenilo.

Zdaj smo torej videli, da je v vrstici y destabiliziran nek interval, recimo $u \ldots v$. Kaj to pomeni za dogajanje eno vrstico višje, v $y+1$? V tej vrstici so tiste prečke, ki ležijo v celoti znotraj intervala $u \ldots v$, izgubile vso podporo od spodaj in so torej tudi same postale nestabilne. Levo od njih je mogoče prečka, ki leži le delno nad $u \ldots v$, njen levi konec pa sega levo od u; če ima ta prečka na tem levem delu še kakšno drugo podporo, je še vedno stabilna, drugače pa je tudi ta v celoti nestabilna. Podobno lahko razmišljamo tudi na desnem koncu intervala $u \ldots v$. Tako lahko določimo interval nestabilnosti v novi vrstici $y + 1$.

S tem postopkom lahko zdaj nadaljujemo gor po mreži in tako v vsaki vrstici določimo interval nestabilnosti. Da izračunamo krajišči intervala v naslednji vrstici iz krajišč intervala v prejšnji vrstici, je koristno, če imamo za poljubno polje pri roki podatek o krajiščih prečke, ki jima pripada; če pa je neko polje prosto, leži med dvema prečkama in je koristno imeti podatek o desnem koncu leve prečke in levem koncu desne prečke. Spodnji program ima v ta namen tabeli levo in desno. S temi podatki bomo lahko novi krajišči izračunali v $O(1)$ časa. Poleg tega je koristno imeti za vsako polje tudi podatek o tem, koliko je črnih polj levo od njega v njegovi

vrstici (tabela crne); potem moramo le odšteti dve taki vrednosti, pa dobimo število črnih polj na nekem intervalu. To naredimo za interval nestabilnosti v vsaki vrstici, rezultate seštevamo in na koncu dobimo skupno število nestabilnih opek. Tako za vsak možni začetni položaj pujsa porabimo le $O(h)$ časa, da ugotovimo, koliko opek je zdaj nestabilnih. Ker je možnih začetnih položajev $O(wh)$, ima naš postopek časovno zahtevnost $O(w \cdot h^2)$.

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

int h, w;
char t[302][302];

/* crne[i][j] = število črnih polj v t[i][1..j];
   levo[i][j] in desno[i][j] = če je polje (i, j) opeka, sta to krajišči njene prečke;
   sicer sta to koordinati najbližje opeke v vrstici i levo in desno od j. */
int levo[302][302], desno[302][302], crne[302][302];

int main()
{
  int i, j, k, nOpek, nNestabilnih, x1, x2, y1, y2;
  FILE *f = fopen("pujsi.in", "rt");

  /* Preberimo vhodne podatke. */
  fscanf(f, "%d %d", &h, &w);
  memset(t, '.', sizeof(t));
  for (j = 1; j <= w; j++) t[0][j] = '#';
  for (i = h; i >= 1; i--) fscanf(f, "%s", t[i] + 1);
  fclose(f);

  /* Pripravimo si tabele levo, desno in crne. */
  nOpek = 0;
  for (i = 1; i <= h; i++)
  {
    crne[i][0]=0;
    levo[i][0]=0;
    for (j = 1; j <= w; j++)
    {
      crne[i][j] = crne[i][j - 1];
      if (t[i][j] == '#') {
        nOpek++; crne[i][j]++;
        if (t[i][j - 1] == '#') levo[i][j] = levo[i][j - 1];
        else levo[i][j] = j; }
      else {
        if (j == 1 || t[i][j - 1] == '#') levo[i][j] = j - 1;
        else levo[i][j] = levo[i][j - 1]; }
    }
    desno[i][w + 1] = w + 1;
    for (j = w; j >= 1; j--)
    {
      if (t[i][j] == '#') {
        if (t[i][j + 1] == '#') desno[i][j] = desno[i][j + 1];
        else desno[i][j] = j; }
      else {
        if (j == w || t[i][j + 1] == '#') desno[i][j] = j + 1;
        else desno[i][j] = desno[i][j + 1]; }
    }
```

```
  }

  /* Izračunajmo rezultate in jih izpišimo. */
  f = fopen("pujsi.out", "wt");
  for (i = h; i >= 1; i--)
  {
    for (j = 1; j <= w; j++)
    {
      if (j != 1) fprintf(f, " ");
      if (t[i][j] == '.') fprintf(f, "%d", nOpek);
      else
      {
        int x1 = j, x2 = j;
        if (t[i - 1][j - 1] == '.' && levo[i - 1][j - 1] < levo[i][j]) x1 = levo[i][j];
        if (t[i - 1][j + 1] == '.' && desno[i][j] < desno[i - 1][j + 1]) x2 = desno[i][j];
        int nNestabilnih = x2 - x1 + 1;
        for (k = i + 1; k <= h; k++)
        {
          /* Interval nestabilnosti v vrstici k - 1 je x1 ... x2. Poglejmo zdaj, katere
             prečke ležijo v vrstici k vsaj delno na tem intervalu. Naj bo y1 levo
             krajišče najbolj leve izmed teh prečk, y2 pa desno krajišče najbolj desne. */
          if (t[k][x1] == '#') y1 = levo[k][x1]; else y1 = desno[k][x1];
          if (t[k][x2] == '#') y2 = desno[k][x2]; else y2 = levo[k][x2];

          /* Najbolj leva in najbolj desna od prečk na intervalu y1 ... y2 imata
             mogoče še kakšno podporo zunaj tega intervala; če je tako, ju iz
             intervala izvzamemo. */
          if (y1 < x1 && (t[k - 1][y1] == '#' || desno[k - 1][y1] < x1))
            y1 = desno[k][desno[k][y1] + 1];
          if (x2 < y2 && (t[k - 1][y2] == '#' || x2 < levo[k - 1][y2]))
            y2 = levo[k][levo[k][y2] - 1];

          /* Tako smo dobili pravi interval nestabilnosti v vrstici k. */
          x1 = y1; x2 = y2;
          if (x2 < x1) break;
          nNestabilnih += crne[k][x2] - crne[k][x1 - 1];
        }
        fprintf(f, "%d", nOpek - nNestabilnih);
      }
    }
    fprintf(f, "\n");
  }
  fclose(f); return 0;
}
```

4. Nakup parcele

Recimo, da nas zanimajo pravokotniki višine n in širine m. Razdelimo v mislih naše zemljišče na „celice“ velikosti $(n-1)\times(m-1)$. Za vsako parcelo (x, y) naj bo $zl(x, y)$ minimum vrednosti vseh parcel (x', y'), ki so isti celici kot (x, y) in ležijo zgoraj levo od nje, torej zanje velja $x' \le x$ in $y' \le y$. Podobno definirajmo še $zd(x, y)$ za območje zgoraj desno od (x, y) (torej $x' \ge x$, $y' \le y$), $sl(x, y)$ za območje spodaj levo ($x' \le x$, $y' \ge y$) in $sd(x, y)$ za območje spodaj desno ($x' \ge x$, $y' \ge y$). Z nekaj pazljivosti lahko vse te minimume izračunamo le v času $O((n-1)(m-1))$ za eno celico oz. $O(wh)$ za celotno zemljišče.

Opazimo lahko, da leži vsak pravokotnik velikosti $n \times m$ v natanko štirih celicah;

njegov presek z zgornjo levo od teh štirih celic je eno od območij, o kakršnih govori tabela *sd*; njegov presek z zgornjo desno od teh celic je eno ob območij, o kakršnih govori tabela *sl*; in tako naprej. Minimum po celem pravokotniku bomo torej dobili tako, da vzamemo minimum primernih štirih vrednosti iz tabel *sl*, *sd*, *zl* in *zd* (za vogale pravokotnika). Tako imamo torej $O(wh)$ dela s pripravo tabel in potem še $O(1)$ z vsakim pravokotnikom, teh pa je tudi $O(wh)$, tako da ima celotni postopek (za enega kupca) časovno zahtevnost $O(wh)$.

Namesto celic velikosti $(n-1) \times (m-1)$ bi lahko uporabili tudi celice velikosti $n \times m$ ali pa celo $(n+1) \times (m+1)$, vendar bi potem naš pravokotnik ne ležal nujno v natanko štirih celicah, pač pa lahko včasih tudi v manj kot štirih, kar bi rešitev malo zapletlo.

```
#include <stdio.h>

#define MaxW 2000
#define MaxH 2000

int a[MaxH][MaxW], zl[MaxH][MaxW], zd[MaxH][MaxW],
    sl[MaxH][MaxW], sd[MaxH][MaxW];

int Min(int a, int b) { return a < b ? a : b; }

int main()
{
   FILE *f = fopen("parcela.in", "rt");
   FILE *g = fopen("parcela.out", "wt");
   int w, h, x, y, k, wk, hk, r;
   int cxCell, cyCell, xCell, yCell, x0, y0, cw, ch;
   long long vsota;

   fscanf(f, "%d %d", &h, &w);

   for (y = 0; y < h; y++) for (x = 0; x < w; x++)
      fscanf(f, "%d", &a[y][x]);

   fscanf(f, "%d", &k);

   while (k-- > 0)
   {
      fscanf(f, "%d %d", &hk, &wk);
      cxCell = wk - 1; cyCell = hk - 1;
      for (yCell = 0; yCell * cyCell < h; yCell++)
      for (xCell = 0; xCell * cxCell < w; xCell++)
      {
         /* Določimo zgornji levi kot in velikost trenutne celice. */
         x0 = xCell * cxCell; y0 = yCell * cyCell;
         cw = Min(cxCell, w - x0);
         ch = Min(cyCell, h - y0);

         /* Izračunajmo zl in zd za to celico (od zgoraj navzdol). */
         for (y = y0; y < y0 + ch; y++) {
            for (x = x0; x < x0 + cw; x++) {
               r = a[y][x];
               if (x > x0) r = Min(r, zl[y][x - 1]);
               if (y > y0) r = Min(r, zl[y - 1][x]);
               zl[y][x] = r; }
            for (x = x0 + cw - 1; x >= x0; x--) {
```

```
            r = a[y][x];
            if (x < x0 + cw − 1) r = Min(r, zd[y][x + 1]);
            if (y > y0) r = Min(r, zd[y − 1][x]);
            zd[y][x] = r; }}

        /* Izračunajmo sl in sd za to celico (od spodaj navzdol). */
        for (y = y0 + ch − 1; y >= y0; y−−) {
          for (x = x0; x < x0 + cw; x++) {
            r = a[y][x];
            if (x > x0) r = Min(r, sl[y][x − 1]);
            if (y < y0 + ch − 1) r = Min(r, sl[y + 1][x]);
            sl[y][x] = r; }
          for (x = x0 + cw − 1; x >= x0; x−−) {
            r = a[y][x];
            if (x < x0 + cw − 1) r = Min(r, sd[y][x + 1]);
            if (y < y0 + ch − 1) r = Min(r, sd[y + 1][x]);
            sd[y][x] = r; }}
      }

      vsota = 0;
      /* Za vsak možni položaj okna izračunajmo minimum cene v oknu. */
      for (y = 0; y + hk <= h; y++) for (x = 0; x + wk <= w; x++)
      {
        /* Vogali okna zagotovo pripadajo različnim celicam. */
        r = Min(Min(sd[y][x], sl[y][x + wk − 1]),
                Min(zd[y + hk − 1][x], zl[y + hk − 1][x + wk − 1]));
        vsota += r;
      }
      fprintf(g, "%lld\n", vsota);
    }

    fclose(f); fclose(g); return 0;
}
```

5. Rotacija

Naj bo $s = s_0 \dots s_{n-1}$ vhodni niz, ki opisuje naše kolo. Za začetek poiščimo največjo številko v nizu; najboljša možna rotacija našega niza se mora očitno začeti pri eni od pojavitev te številke. Ker še ne vemo, pri kateri, si jih vse zapišimo v nek seznam. Nato za vsako od njih poglejmo, kaj bi dobili na drugem mestu, če bi našo rotacijo začeli pri tej številki; to je številka, ki pride v vhodnem nizu eno mesto za tisto, pri kateri smo začeli našo rotacijo; med temi številkami si zapomimo največjo in obdržimo v seznamu le tiste kandidatke, ki imajo na drugem mestu res to številko. Podobno nadaljujemo s številkami na tretjem mestu in tako naprej. Postopek se konča, če ostane le še ena kandidatka ali pa ko dosežemo dolžino n. S psevdokodo ga lahko zapišemo takole:

$m := \max\{s_0, s_1, \dots, s_{n-1}\}$;
$L := \{i : s_i = m\}$; $d := 1$;
while $|L| > 1$ **and** $d < n$:
  (* *L je množica indeksov, na katerih se v s začne leksikografsko največji podniz dolžine d.* *)
  $m := \max\{s_{i+d} : i \in L\}$;
  $L' := \{i \in L : s_{i+d} = m\}$;

$L := L'$; $d := d + 1$;

Pri tem si moramo predstavljati, da niz s naslavljamo ciklično: kjer se gornja psevdokoda sklicuje na s_i za $i \geq n$, moramo v resnici uporabiti $s_{i \bmod n}$.

Slabo pri tem postopku je, da ima lahko časovno zahtevnost $O(n^2)$; na primer, če so vse števke niza s enake, bomo šla glavna zanka vse do $d = n$ in v množici L bodo ves čas kar vsi indeksi od 0 do $n - 1$. Rešitev lahko izboljšamo z naslednjim razmislekom. Na začetku vsake iteracije glavne zanke velja naslednje: če poiščemo v s leksikografsko največji podniz dolžine d (pri tem si moramo ves čas predstavljati s kot cikličen niz) — recimo temu podnizu x — so v L vsi indeksi, na katerih se x pojavlja kot podniz v s. Po vsaki iteraciji glavne zanke nekatere pojavitve mogoče izpadejo iz L, tiste, ki ostanejo, pa se podaljšajo za en znak (ker se d poveča za 1). V nekem trenutku se mogoče zgodi, da se dve pojavitvi sprimeta skupaj: v L najdemo torej nek i in za njim še $i + d$. Recimo, da se je tako sprijelo k pojavitev podniza x; niz s lahko zdaj v mislih razdelimo na nekaj delov: $s = ux^kv$, pri čemer je u območje pred indeksom i, v pa območje za $i + kd$. Kakšne rotacije dobimo, če s zarotiramo za i, $i + d$, ..., $i + kd$ mest? Pri rotaciji za i dobimo $s^{(i)} = x^kvu$; pri rotaciji za $i + d$ dobimo $s^{(i+d)} = x^{k-1}vux$; in v splošnem pri rotaciji za $i + td$ dobimo $s^{(i+td)} = x^{k-t}vux^t$. Če primerjamo $s^{(i+td)}$ (za $1 \leq t \leq k$) in $s^{(i)}$, vidimo, da se oba začneta na $k - t$ pojavitev niza x, zatem pa ima $s^{(i)}$ še eno pojavitev niza x, niz $s^{(i+td)}$ pa ima tam v (oz. natančneje, tam ima prvih d znakov niza vux^t), kar je gotovo leksikografsko večje od x (saj bi bil drugače tudi indeks $i + (k + 1)d$ tudi še prisoten v L, mi pa smo predpostavili, da ni). Torej $s^{(i+td)}$ gotovo ne bo dal največjega dobitka, torej lahko indeks $i + td$ brez škode pobrišemo iz L.

S takšnim sprotnim brisanjem, čim se začnejo najdaljši podnizi dolžine d sprijemati, bomo tudi zagotovili, da se podnizi dolžine d z začetkom pri indeksih iz L nikoli ne bodo prekrivali. Pri dolžini d lahko torej L vsebuje največ n/d različnih indeksov. Skupna cena vseh podaljševanj vseh podnizov (oz. prenašanja indeksov iz L v L' pri posameznih iteracijah glavne zanke) je torej $\leq \sum_{d=1}^{n}(n/d) = n\sum_{d=1}^{n}(1/d) \approx n \log n$. Tako smo časovno zahtevnost našega postopka izboljšali z $O(n^2)$ na $O(n \log n)$.

Spodnja rešitev v C-ju hrani množico L kar v tabeli zacetki (število njenih elementov pa v nZacetkov).

```
#include <stdio.h>

#define MaxN 1000000
char s[2 * MaxN];
int zacetki[MaxN], zacetki2[MaxN];

int main()
{
  int n, nZacetkov, nZacetkov2, i, d;
  char cNaj;

  FILE *f = fopen("rotacija.in", "rt");
  fscanf(f, "%d\n", &n);
  fgets(s, MaxN + 3, f);
  fclose(f);

  nZacetkov = 0;
  for (i = 0, cNaj = 0; i < n; i++) {
    s[n + i] = s[i];
```

```
    if (s[i] > cNaj) cNaj = s[i], nZacetkov = 0;
    if (s[i] == cNaj) zacetki[nZacetkov++] = i; }

  d = 1;
  while (nZacetkov > 1)
  {
    /* Pobrišimo neobetavne začetke (torej tiste, pri katerih se je trenutna pojavitev
       podniza x sprijela s prejšnjo). Spotoma si v cNaj zapomnimo največjo številko,
       ki sledi kakšni pojavitvi dosedanjega x. */
    cNaj = 0;
    for (i = 0, nZacetkov2 = 0; i < nZacetkov; i++) {
      if (i > 0 && zacetki[i] == zacetki[i - 1] + d) continue;
      zacetki2[nZacetkov2++] = zacetki[i];
      if (s[zacetki[i] + d] > cNaj) cNaj = s[zacetki[i] + d]; }

    /* Pripravimo nov seznam začetkov; od dosedanjih začetkov obdržimo tiste,
       pri katerih za x-om pride številka cNaj in ne kakšna manjša. */
    for (i = 0, nZacetkov = 0; i < nZacetkov2; i++)
      if (s[zacetki2[i] + d] == cNaj)
        zacetki[nZacetkov++] = zacetki2[i];
    d += 1;
  }

  f = fopen("rotacija.out", "wt");
  fwrite(&s[zacetki[0]], n, sizeof(s[0]), f);
  fclose(f); return 0;
}
```

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Kamen, papir, škarje

Spodnja rešitev v zanki prebira vhodne podatke; v vsaki iteraciji prebere dva znaka in obdela potezo, ki jo opisujeta. V spremenljivki tocke hranimo razliko med številom točk tekmovalca A in številom točk tekmovalca B; torej, če je tocke > 0, je v vodstvu tekmovalec A, če je tocke < 0, je v vodstvu tekmovalec B, sicer pa je rezultat trenutno neodločen. Pri vsaki potezi moramo vrednost tocke primerno popraviti (če je potezo dobil A, povečamo tocke za 1; če jo je dobil B, pa zmanjšamo tocke za 1).

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

int main()
{
  int tocke = 0, a, b;
  while (true) {
    /* Preberimo potezi obeh tekmovalcev. */
    a = getchar(); b = getchar();
    /* Poglejmo, kdo je dobil to potezo (oz. ali je neodločena). */
    if (a == 'K') tocke += (b == 'S') ? 1 : (b == 'P') ? -1 : 0;
    else if (a == 'P') tocke += (b == 'K') ? 1 : (b == 'S') ? -1 : 0;
    else if (a == 'S') tocke += (b == 'P') ? 1 : (b == 'K') ? -1 : 0;
    else /* Do tega else pridemo na koncu vhodnih podatkov, ko je a == EOF. */
      break; }
  /* Izpišimo rezultat. */
  if (tocke == 0) printf("Neodloceno.\n");
  else printf("Zmagovalec je %s.\n", (tocke > 0) ? "A" : "B");
  return 0;
}
```

2. Papajščina

Vhodne podatke lahko beremo znak po znak; vsak znak takoj tudi izpišemo, nato pa preverimo, če je bil samoglasnik, in v tem primeru izpišemo še „p“ in nato še eno kopijo pravkar prebranega znaka.

```
#include <stdio.h>

int main()
{
  int c;
  /* Prebirajmo vhodne podatke po znakih. */
  while ((c = getchar()) != EOF)
  {
    /* Izpišimo pravkar prebrani znak. */
    putchar(c);
    /* Preverimo, ali je samoglasnik. */
    if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
      /* Izpišimo „p“ in še eno kopijo našega samoglasnika. */
      putchar('p'); putchar(c); }
  }
  return 0;
}
```

3. Obfuskator

Vhodne podatke berimo po vrsticah, pri vsaki vrstici primerno premešajmo znake in jo nato izpišimo. Recimo, da imamo besedo, dolgo n znakov, shranjeno v tabeli s od $s[0]$ do $s[n-1]$. Ker moramo prvi in zadnji znak pustiti pri miru, pridejo v poštev za mešanje le znaki od $s[1]$ do $s[n-2]$. Mešanje lahko izvedemo tako, da se z i sprehodimo od 1 do $n-2$; pri vsakem i si izberimo nek naključen indeks j z območja od 1 do i in nato znaka $s[i]$ in $s[j]$ zamenjamo. (Ker pri $i = 1$ tudi j ne more biti drugega kot 1, lahko zanko pravzaprav začnemo šele pri $i = 2$.) Pokazati je mogoče, da če funkcija Nakljucno(k) vrne vsako od vrednosti $0, \ldots, k-1$ z enako verjetnostjo (torej $1/k$), bo tudi ta postopek mešanja z enako verjetnostjo zgeneriral vsako od možnih premešanih različic vhodne besede.

```
#include <stdio.h>
#define MaxDolzina 20

int main()
{
  char s[MaxDolzina + 1], c; int i, j, n;
  /* Prebirajmo vhodne podatke po vrsticah. */
  while (gets(s)) {
    /* Naj bo n dolžina trenutne besede. */
    n = 0; while ('a' <= s[n] && s[n] <= 'z') n++;
    /* Premešajmo znake v njej, razen prvega in zadnjega. */
    for (i = 2; i < n - 1; i++) {
      j = Nakljucno(i) + 1; /* Naključno število od 1 do i. */
      /* Zamenjajmo s[i] in s[j]. */
      c = s[i]; s[i] = s[j]; s[j] = c; }
    /* Izpišimo premešano besedo. */
    puts(s); }
  return 0;
}
```

4. Zarota

Vsako strnjeno podzaporedje vhodnega zaporedja je določeno s svojim začetnim indeksom (recimo mu i) in s svojim končnim indeksom (recimo mu j). Zelo naivna rešitev bi torej bila, da z gnezdenimi zankami pregledamo vse možne i in j, pri vsakem izračunamo vsoto zaporedja in preverimo, ali je večkratnik k-ja:

```
x := 0;
for i := 1 to n:
  for j := i to n:
    s := 0;
    for t := i to k:
      s := s + a_t;
    (* Zdaj je v s vsota a_i + ... + a_j. *)
    if s mod k = 0 then x := x + 1;
return x;
```

Ta rešitev ima časovno zahtevnost kar $O(n^3)$, saj izgubi veliko časa s tem, da znova in znova računa vsote podzaporedij. Opazimo lahko, da ko se j poveča za 1, pridobi opazovano podzaporedje na koncu nov člen z vrednostjo a_j (za novo vrednost j); torej ni treba vsote računati na novo od začetka, ampak lahko popravimo dosedanjo vsoto $a_i + \ldots + a_{j-1}$ tako, da ji prištejemo a_j. Tako se znebimo najbolj notranje zanke in dobimo rešitev s časovno zahtevnostjo $O(n^2)$:

```
x := 0;
for i := 1 to n:
  s := 0;
  for j := i to n:
    s := s + a_j;
    (* Zdaj je v s vsota a_i + ... + a_j. *)
    if s mod k = 0 then x := x + 1;
return x;
```

Do še učinkovitejših rešitev pa lahko pridemo z naslednjim razmislekom. Naj bo s_i vsota prvih i členov našega zaporedja, torej $s_i = a_1 + \ldots + a_i$. S tako dobljenimi s_i lahko elegantno izrazimo vsoto poljubnega podzaporedja: če imamo podzaporedje z začetkom pri i in koncem pri j, je vsota $a_i + \ldots + a_j$ enaka $s_j - s_{i-1}$. Namesto da se sprašujemo, kolikokrat (pri koliko i, j) se zgodi, da je $a_i + \ldots + a_j$ večkratnik k, se lahko vprašamo, kolikokrat se zgodi, da je $s_j - s_{i-1}$ večkratnik k; slednje pa se zgodi natanko tedaj, ko imata s_j in s_{i-1} enak ostanek po deljenju s k. Namesto vsot s_i so torej še bolj zanimivi njihovi ostanki po deljenju s k; recimo jim $t_i := s_i \bmod k$.

Lahko se torej sprehodimo z zanko po j in se pri vsakem j vprašamo, koliko je bilo pred njim takih i, za katere je $t_i = t_j$. Ker so vsi t-ji nastali kot ostanki po deljenju s k, so njihove vrednosti le cela števila od 0 do $k-1$, zato jih lahko poskusimo hraniti v tabeli u s k elementi:

```
for r := 0 to k − 1 do u_r := 0;
x := 0; t := 0; u_t := u_t + 1;
for j := 1 to n:
  t := (t + a_j) mod k;
  (* Zdaj je t = t_j = (a_1 + ... + a_j) mod k. Poleg tega nam u_t pove, koliko
     je na območju 0, ..., j − 1 takšnih indeksov i, za katere velja t_i = t. *)
  x := x + u_t;
  u_t := u_t + 1;
return x;
```

Ta rešitev porabi $O(n+k)$ časa in $O(k)$ dodatnega pomnilnika (zaradi tabele u); ker naloga pravi, da je k majhen v primerjavi z n, je to precej bolje od dosedanjih rešitev. Za primere, ko bi bil k velik v primerjavi z n, bi jo bilo koristno predelati tako, da bi tabela u postala razpršena tabela (*hash table*); če predpostavimo, da je cena dostopa do razpršene tabele $O(1)$, bi imela ta rešitev časovno in prostorsko zahtevnost $O(n)$. Še ena možnost je, da vrednosti t_j zložimo v tabelo in jo uredimo, tako da pridejo enake vrednosti skupaj; nato naredimo podobno zanko kot v zadnji rešitvi zgoraj, le da si ni treba zapomniti celotne tabele u, pač pa le po eno vrednost naenkrat (kajti ko enkrat v urejeni tabeli t_j-jev naletimo na novo vrednost, vemo, da se stara ne bo

nikoli več pojavila, saj je tabela urejena naraščajoče). Ta rešitev bi imela časovno zahtevnost $O(n \log n)$ in prostorsko zahtevnost $O(n)$.

Razmislimo še o težji različici naloge, pri kateri so dovoljena tudi nestrnjena podzaporedja. V tej različici je naloga primerna za reševanje z dinamičnim programiranjem, pri čemer si je koristno zastaviti podprobleme takšne oblike: naj bo $f(r, m)$ število takih nestrnjenih podzaporedij zaporedja $a_1, \ldots, a_m$, pri katerih dá vsota podzaporedja po deljenju s k ostanek r. Rezultat, po katerem sprašuje naloga, je potem $f(0, n)$.

Naše podprobleme lahko rešujemo z rekurzivnim razmislekom: podzaporedja, ki se končajo najkasneje pri a_m, lahko ločimo na tista, ki se končajo že pred a_m (med takšnimi je za naš namen ugodnih $f(r, m-1)$ podzaporedij) in tista, ki vsebujejo a_m. Slednja lahko preštejemo takole: recimo, da vzamemo neko tako podzaporedje, ki vsebuje a_m in ima njegova vsota po deljenju s k ostanek r; če mu zadnji člen pobrišemo, dobimo podzaporedje, ki se konča najkasneje pri a_{m-1}, njegova vsota po deljenju s k pa ima ostanek $r - a_m$.[4] Takšnih zaporedij je torej $f(r - a_m, m-1)$. Tako smo dobili zvezo $f(r, m) = f(r, m-1) + f(r - a_m, m-1)$.

Funkcijo f lahko računamo sistematično po naraščajočem m in pri vsakem m po vseh r; da bo rešitev učinkovita, si moramo rezultate podproblemov zapomniti v neki tabeli, da jih bomo imeli kasneje pri roki za reševanje večjih podproblemov. Takšna rešitev ima časovno zahtevnost $O(km)$ in prostorsko zahtevnost $O(k)$.

5. Sveče

Za vsako svečo je koristno hraniti naslednje podatke: trenutna višina sveče; ali je trenutno prižgana; od kdaj gori (če je trenutno prižgana); koliko časa je doslej gorela. Spodnji program ima v ta namen tabele visina, gori, goriOd in casGorenja. Na začetku preberemo začetne višine, postavimo čase gorenja na 0 in označimo v tabeli gori vse sveče kot ugasnjene. Ko svečo prižgemo, moramo le postaviti njen gori na **true** in shraniti trenuten čas v njen goriOd. Ko svečo ugasnemo, pa postavimo njen gori na **false** in popravimo njen čas gorenja in višino. Slednji se načeloma poveča za razliko med trenutnim časom (ko smo svečo ugasnili) in časom goriOd te sveče (ko smo jo prižgali), paziti pa moramo še na možnost, da je sveča medtem že popolnoma dogorela in torej v resnici ni bila ves ta čas prižgana.

Ker naloga zagotavlja, da se bodo vhodni podatki končali z operacijo ugašanja sveč, smo lahko prepričani, da bomo imeli ob koncu glavne zanke v tabelah casGorenja in visina prave vrednosti; zdaj moramo le še poiskati, katera sveča ima največji čas gorenja.

```
#include <stdio.h>
#include <stdbool.h>
#define MaxN 10
#define HitrostGorenja 1

int main()
{
  int i, n, t, naj, cas, visina[MaxN], goriOd[MaxN], casGorenja[MaxN];
```

[4] Vrednost $r - a_m$ moramo seveda tudi gledati po modulu k, torej bi jo v praksi lahko računali kot $(r - (a_m \bmod k) + k) \bmod k$.

```
  bool gori[MaxN];

  /* Preberimo začetne višine in označimo vse sveče kot ugasnjene. */
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    scanf("%d", &visina[i]);
    casGorenja[i] = 0; gori[i] = false; }

  /* Obdelajmo podatke o prižiganjih in ugašanjih. */
  while (2 == scanf("%d %d", &t, &i))
    if (i > 0) { /* Prižgali smo svečo i. */
      i--;
      if (! gori[i]) gori[i] = true, goriOd[i] = t; }
    else /* Ugasnili smo vse sveče. */
      for (i = 0; i < n; i++) {
        if (! gori[i]) continue;
        /* Koliko časa je gorela, odkar smo jo nazadnje prižgali? */
        cas = t - goriOd[i];
        /* Mogoče je med tem že popolnoma dogorela. */
        if (cas * HitrostGorenja > visina[i])
          cas = visina[i] / HitrostGorenja;
        /* Izračunajmo novo višino in čas gorenja. */
        visina[i] -= cas * HitrostGorenja;
        casGorenja[i] += cas; gori[i] = false; }

  /* Poglejmo, katera sveča je gorela najdlje, in jo izpišimo. */
  for (i = 1, naj = 0; i < n; i++)
    if (casGorenja[i] > casGorenja[naj]) naj = i;
  printf("Najdlje je gorela sveča %d.\n", naj + 1);
  return 0;
}
```

Če bi bila hitrost gorenja različna od 1, bi bilo koristno za računanje s časi in višinami uporabiti tip **double** namesto **int**, saj bi bile drugače lahko zaokrožitvene napake prehude.

Naloge so sestavili: prepletene besede, ovce — Nino Bašić; največji pretok — Primož Gabrijelčič; leteči pujsi — Matija Grabnar; potovanje, nakup parcele, rotacija — Tomaž Hočevar; mase — Nace Hudobivnik; kazenski stavek — Jurij Kodre; kamen, papir, škarje — Mitja Lasič; manjkajoča števila, obfuskator, papajščina, strukturirani podatki — Mark Martinec; spričevala — Mojca Miklavec; sveče, zarota, v Afganistan!, razpolovišče lika, de-FFT permutacija — Mitja Trampuš.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2010

1. Povprečne temperature

Recimo, da imamo n postaj in da je i-ta postaja izmerila temperaturo t_i stopinj. Če bi ozemlje vseh postaj ležalo v celoti znotraj Slovenije, bi bilo dovolj že izračunati čisto navadno povprečje:

$$\frac{t_1 + t_2 + \ldots + t_n}{n}.$$

V našem primeru pa moramo meritev posamezne postaje upoštevati sorazmerno s tem, kolikšen delež njenega ozemlja leži v Sloveniji; recimo, da je pri i-ti postaji ta delež enak p_i odstotkov. Naša vsota v števcu se tako spremeni v $p_1 \cdot t_1 + p_2 \cdot t_2 + \ldots + p_n \cdot t_n$. Spremeniti pa moramo tudi imenovalec; tudi tu morajo imeti tiste postaje, katerih območje delno leži zunaj Slovenije, manjši vpliv.[5] Namesto preprostega števila postaj moramo vzeti vsoto njihovih uteži. Tako dobimo formulo za uteženo povprečje:

$$\frac{p_1 \cdot t_1 + p_2 \cdot t_2 + \ldots + p_n \cdot t_n}{p_1 + p_2 + \ldots + p_n}.$$

Naš program lahko obe vsoti računa sproti, med branjem vhodnih podatkov. V spodnjem programu se vsota iz števca počasi računa v spremenljivki vsotaTemperatur, tista iz imenovalca pa v spremenljivki vsotaUtezi. V vsaki iteraciji glavne zanke preberemo nov par (t_i, p_i) in ustrezno povečamo obe vsoti; na koncu moramo le deliti vsoto (uteženih) temperatur z vsoto uteži in izpisati dobljeni rezultat.

```
#include <stdio.h>

int main()
{
  double temperatura, vsotaTemperatur = 0; int utez, vsotaUtezi = 0;
  while (2 == scanf("%lf %d", &temperatura, &utez)) {    /* Preberimo meritev. */
    vsotaTemperatur += temperatura * utez;               /* Povečajmo obe vsoti. */
    vsotaUtezi += utez; }
  printf("%.2f\n", vsotaTemperatur / vsotaUtezi); /* Izračunajmo in izpišimo rezultat.*/
  return 0;
}
```

Pri branju vhodnih podatkov smo si pomagali s funkcijo scanf iz standardne knjižnice jezika C. Ta funkcija vrne število polj, ki jih je uspešno prebrala; ker smo od nje zahtevali, naj prebere dve polji (temperaturo in utež), torej pričakujemo, da vrne 2; če vrne kaj drugega, je to znak, da smo prišli do konca vhodnih podatkov (ali pa da je v njih napaka).

Kot zanimivost povejmo še, da je pri računanju uteženega povprečja pravzaprav vseeno, v kakšnih enotah so izražene uteži, samo da so vse izražene v istih enotah. Namesto v odstotkih (od 0 do 100) bi bile naše uteži lahko predstavljene kot deleži

[5] O tem se lahko prepričamo na primer z naslednjim razmislekom. Recimo, da bi imeli dve postaji, pri čemer ozemlje prve v celoti leži znotraj Slovenije ($p_1 = 100\,\%$), ozemlje druge pa leži pol znotraj in pol zunaj ($p_2 = 50\,\%$). Če izmerita obe enako temperaturo, recimo T (torej $t_1 = t_2 = T$), bi pričakovali, da bo tudi (uteženo) povprečje enako T. V števcu imamo $p_1 t_1 + p_2 t_2 = 150\,\% \cdot T$, torej mora biti imenovalec $150\,\%$, to pa je ravno $p_1 + p_2$.

(realna števila od 0 do 1) ali pa celo v kvadratnih kilometrih (koliko km^2 slovenskega ozemlja pokriva tista postaja). To je posledica dejstva, da če se v naši formuli za uteženo povprečje vse uteži p_i pomnožijo z nekim konstantnim faktorjem C, se bo ta C v številcu in imenovalcu okrajšal, tako da bo ostalo uteženo povprečje enako kot prej.

2. Skakačev obhod

Za začetni položaj moramo preveriti, da sta obe koordinati z območja $\{1, \ldots, 8\}$; za vsak premik moramo preveriti, da je ena od sprememb koordinat ($\triangle x$ ali $\triangle y$) enaka ± 1, druga pa ± 2 (pri tem pride prav funkcija `abs` iz standardne C-jeve knjižnice, s katero izračunamo absolutno vrednost števila); po vsakem premiku moramo izračunati novi položaj skakača in preveriti, ali je ta še vedno na šahovnici (torej ali sta obe koordinati še vedno na območju $\{1, \ldots, 8\}$).

Ostane nam še preverjanje, ali je vsako polje obiskano natanko enkrat. Pri tem si lahko pomagamo s tabelo logičnih vrednosti (v spodnjem programu je to `obiskano`), v kateri za vsako polje šahovnice označimo, ali smo ga že obiskali ali ne. Na začetku postavimo vse elemente te tabele na **false**, nato pa jih med branjem premikov počasi postavljamo na **true**. Tako lahko sproti zelo poceni preverjamo, ali nas nek premik vodi na polje, ki smo ga obiskali že nekoč prej (to namreč krši pravila naloge, saj moramo vsako polje obiskati natanko enkrat).

S tem smo preverili, da nismo nobenega polja obiskali več kot enkrat. To pa tudi pomeni, da je po 63 premikih obiskanih že 64 polj (začetno polje + po eno novo po vsakem premiku), to pa je ravno toliko, kolikor je polj na šahovnici. Zato nam na koncu ni treba še posebej preverjati, ali smo res obiskali vsa polja šahovnice; če pregledamo vseh 63 premikov, ne da bi našli kakšno napako, potem že vemo, da smo prebrali veljaven skakačev obhod.

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

int main()
{
  int x, y, dx, dy, i; bool obiskano[8][8];
  /* Na začetku so vsa polja neobiskana. */
  for (y = 0; y < 8; y++) for (x = 0; x < 8; x++) obiskano[y][x] = false;
  scanf("%d %d", &x, &y); /* Preberimo začetni položaj. */
  x--; y--;               /* Pretvorimo koordinate iz 1..8 v 0..7. */
  if (x < 0 || x > 7 || y < 0 || y > 7) return 1; /* Ali je začetni položaj veljaven? */
  obiskano[y][x] = true;                          /* Označimo to polje za obiskano. */
  /* Preberimo zdaj 63 premikov. */
  for (i = 1; i <= 63; i++) {
    scanf("%d %d", &dx, &dy);
    /* Ali je ta premik v obliki L? */
    if (! (abs(dx) == 1 && abs(dy) == 2) && ! (abs(dx) == 2 && abs(dy) == 1))
      return 1;
    /* Preverimo, ali smo po tem premiku še vedno na šahovnici. */
    x += dx; y += dy;
    if (x < 0 || x > 7 || y < 0 || y > 7) return 1;
```

```
    if (obiskano[y][x]) return 1; /* Na vsako polje smemo skočiti le enkrat. */
    obiskano[y][x] = true; }
  printf("OK\n"); return 0;
}
```

3. Pravokotni meseci

Od ponedeljka do nedelje je (vključno s tema dnevoma) 7 dni; do naslednje nedelje je še nadaljnjih 7 dni in tako naprej. Če se torej nek mesec začne s ponedeljkom in konča z nedeljo, mora biti skupno število dni v njem večkratnik števila 7. Možne dolžine mesecev so od 28 do 31 dni, torej so za pravokotne mesece primerni le tisti z 28 dnevi; torej le februar, pa še to le, če leto ni prestopno. Tako torej vidimo, da je vprašanje, ali neko leto vsebuje pravokotni mesec, pravzaprav vprašanje, ali leto ni prestopno in ali prvi februar tega leta pade na ponedeljek.

Zdaj se moramo le zapeljati v zanki po vseh letih in pri vsakem preveriti, če ni prestopno in če pade prvi februar na ponedeljek. Če si lahko pomagamo s funkcijo DanVTednu, je rešitev zelo preprosta:

```
#include <stdio.h>

void PravokotniMeseci(int letoOd, int letoDo)
{
  int leto;
  for (leto = letoOd; leto <= letoDo; leto++) {
    if (leto % 4 == 0 && (leto % 100 != 0 || leto % 400 == 0))
      continue; /* to leto je prestopno */
    if (DanVTednu(1, 2, leto) == 1) /* Ali je prvi februar na ponedeljek? */
      printf("%d\n", leto); }
}
```

Kaj pa, če funkcije DanVTednu nimamo? Ena možnost je, da si jo sami napišemo. Dani datum najprej pretvorimo v zaporedno številko dneva od nekega izbranega začetnega izhodišča. Spodnja funkcija se pretvarja, da je od leta 1 naprej veljal prav takšen gregorijanski koledar, kot ga poznamo dandanes, in vrne zaporedno številko dneva od začetka leta 1 naprej (prvi januar leta 1 dobi številko 1, drugi januar leta 1 dobi številko 2; 27. marec 2010 dobi številko 733 858 itd.).

```
int ZapStDneva(int dan, int mesec, int leto)
{
  static const int dolzina[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  int n, m, L = leto - 1;
  /* Preštejmo dneve v 400-letnih obdobjih (oblike 1..400, 401..800 itd.),
     ki v celoti ležijo pred danim letom. Vsako od teh ima 97 prestopnih let. */
  n = (L / 400) * (365 * 400 + 97); L %= 400;
  /* Prištejmo dneve v stoletjih, ki v trenutnem 400-letnem obdobju ležijo
     v celoti pred pred danim letom. Vsako od teh ima 24 prestopnih let. */
  n += (L / 100) * (365 * 100 + 24); L %= 100;
  /* Prištejmo dneve v letih, ki v celoti ležijo pred danim letom. */
  n += L * 365 + L / 4;
  /* Koliko je dni v tem letu pred danim mesecem? */
  for (m = 1; m < mesec; m++) n += dolzina[m - 1];
  if (mesec > 2 && leto % 4 == 0 && (leto % 100 != 0 || leto % 400 == 0)) n++;
  return n + dan;
}
```

Zdaj vemo, da če se zaporedni številki dveh dni razlikujeta za večkratnik števila 7, to pomeni, da padeta tadva dneva na isti dan v tednu. Če rešujemo to nalogo na tekmovanju leta 2010, vidimo, da je danes sobota, 27. marca 2010; 29. marec je torej ponedeljek; če vzamemo razliko med zaporedno številko dneva, ki nas zanima, in zaporedno številko 29. marca 2010, nam ostanek po deljenju te razlike s 7 pove, na kateri dan v tednu je padel datum, ki nas zanima (0 = ponedeljek, 1 = torek in tako naprej; ker hočemo rezultat od 1 do 7 namesto od 0 do 6, mu bomo na koncu prišteli 1).

```
void DanVTednu(int dan, int mesec, int leto)
{
  int d = ZapStDneva(dan, mesec, leto) − ZapStDneva(29, 3, 2010);
  d %= 7; if (d < 0) d += 7;
  return d + 1;
}
```

Nalogo lahko rešimo tudi brez funkcije, kot je DanVTednu. Recimo, da imamo pri roki koledar in vidimo, da je bil 1. februar 2010 na ponedeljek. Ker leto 2010 ni prestopno, mine do naslednjega 1. februarja (leta 2011) 365 dni, kar je 52 tednov in 1 dan ($365 = 7 \cdot 52 + 1$); torej je 1. februar 2011 torek. Iz podobnega razloga je 1. februar 2012 sreda; nato pa, ker je leto 2012 prestopno, mine do naslednjega 1. februarja 52 tednov in 2 dni, zato je 1. februar 2013 petek (in ne na primer četrtek). Podobno lahko razmišljamo tudi za nazaj: ker je bil 1. februar 2010 ponedeljek, je bila 1. februarja 2009 nedelja, 1. februarja 2008 pa petek (in ne sobota, kajti 2008 je bilo prestopno leto) in tako nazaj. Lahko se torej z zanko zapeljemo od leta 2010 naprej ali nazaj do leta letoOd in tako za to leto ugotovimo, kateri dan v tednu je bil tistega leta prvi februar; od tam pa se potem v zanki zapeljemo naprej do leta letoDo in tako pregledamo še vsa ostala leta, ki nas zanimajo.

Če se obdobje, ki nas zanima, začne zelo daleč od leta 2010, lahko prihranimo nekaj časa tudi z naslednjim opažanjem: vzorec prestopnih in navadnih let se ponavlja na vsakih 400 let; v takem obdobju je 97 prestopnih let; skupaj je torej takšno obdobje dolgo $400 \cdot 365 + 97 = 146\,097$ dni, kar je natanko 20 871 tednov. Če torej vzamemo nek datum in letnici prištejemo (ali odštejemo) nek večkratnik števila 400, se dan v tednu s tem nič ne spremeni. Ker je bil 1. februar 2010 ponedeljek, bo prišel 1. februar na ponedeljek tudi v letih 2410, 2810, 3210 in tako naprej. Tako pridemo do naslednje rešitve:

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

void PravokotniMeseci(int letoOd, int letoDo)
{
  enum { Leto0 = 2010, Dan0 = 1 };
  int leto, d; bool prestopno;

  /* Postavimo se na zadnje leto oblike 2010 − 400 * k, ki leži na ali pred letom letoOd. */
  d = letoOd − Leto0;
  if (d > 0) leto = Leto0 + (d / 400) * 400;
  else leto = Leto0 − ((−d + 399) / 400) * 400;

  /* Za vsako leto do letoDo poglejmo, ali vsebuje pravokotni mesec. */
  for (d = Dan0; leto <= letoDo; leto++) {
    prestopno = (leto % 4 == 0 && (leto % 100 != 0 || leto % 400 == 0));
```

```
    if (leto >= letoOd && d == 1 && ! prestopno)
      printf("%d\n", leto);
    /* Izračunajmo dan, na katerega bo padel 1. februar naslednje leto. */
    d = (d + (prestopno ? 2 : 1)) % 7; }
}
```

Ta razmislek nam nakazuje tudi pot do učinkovite rešitve podnaloge (b). Videli smo, da se vzorec tega, katera leta imajo pravokotni mesec in katera ne, ponavlja na vsakih 400 let. Recimo, da nas za neko zelo dolgo obdobje, na primer n let, zanima, koliko je v tem obdobju let s pravokotnim mesecem. Če delimo n s 400 in dobimo $n = 400 \cdot k + r$, lahko torej naše obdobje razdelimo na k štiristoletnih obdobij in nato še krajše obdobje zadnjih r let. Vsako od štiristoletnih obdobij ima enako število let s pravokotnim mesecem, zato je dovolj, če preštejemo pravokotne mesece le v prvem od teh obdobij in jih pomnožimo s k. Nato prištejemo še število pravokotnih mesecev v zadnjem r-letnem obdobju. Tudi pri zelo velikih n moramo torej pregledati le eno 400-letno in eno r-letno (za $r < 400$) obdobje — celo če bi šel n v milijarde let.

Zapišimo dobljeni postopek še v obliki podprograma. Pri tem smo predpostavili, da smo enega od doslej predstavljenih podprogramov PravokotniMeseci predelali tako, da let s pravokotnim mesecem ne izpisuje, pač pa nam vrne njihovo število. Učinkovit postopek za štetje pravokotnih mesecev v daljšem obdobju je torej tak:

```
int KolikoPravokotnihMesecev(int letoOd, int letoDo)
{
  int n = letoDo − letoOd + 1;
  int k = n / 400, r = n % 400;
  return PravokotniMeseci(letoOd, letoOd + 399) * k +
         PravokotniMeseci(letoDo − r + 1, letoDo);
}
```

Oglejmo si zdaj še podnalogo (c), pri kateri nas zanimajo dobri meseci, torej taki z več kot osmimi sobotami in nedeljami. Očitna rešitev seveda je, da gremo v zanki po vseh dnevih v mesecu, za vsakega ugotovimo, kateri dan v tednu je, in štejemo sobote in nedelje. Lepše pa bi bilo imeti učinkovitejšo rešitev, ki se ji ne bi bilo treba ukvarjati z vsakim dnem posebej.

Ne glede na to, s katerim dnem v tednu se začne naš mesec, se bo v prvih sedmih dneh meseca pojavil vsak dan v tednu po enkrat (med njimi sta ena sobota in ena nedelja); v drugih sedmih ravno tako; in tako naprej. Prvih 28 dni meseca torej prispeva štiri sobote in štiri nedelje, ne glede na to, s katerim dnem v tednu se je mesec začel. O tem, ali je mesec dober, torej odločajo le dnevi od 29. naprej (če je mesec sploh tako dolg) — če je kakšen od teh dni sobota ali nedelja, je mesec dober, sicer pa ne. Ker pade 29. dan meseca na isti dan v tednu kot 1. dan tega meseca, je torej mesec z 29 dnevi dober natanko tedaj, ko se začne na soboto ali nedeljo. Pri mesecu s 30 dnevi moramo upoštevati še, da 30. dan pade na isti dan v tednu kot 2. dan tega meseca, torej bo tak mesec dober tudi, če je 2. dan v njem sobota ali nedelja, to pa je takrat, ko je 1. dan meseca petek ali sobota. Podoben razmislek lahko naredimo tudi pri mesecih z 31 dnevi. Zaključek je torej takšen: mesec z 29 dnevi je dober, če se začne na soboto ali nedeljo; mesec s 30 dnevi je dober, če se začne na petek, soboto ali nedeljo; mesec z 31 dnevi je dober, če se začne na četrtek, petek, soboto ali nedeljo.

Pri (*a*) smo se spraševali o tem, v katerih letih je prisoten kakšen pravokoten mesec; če bi se zdaj podobno vprašali o dobrih mesecih, naloga ne bi bila preveč zanimiva, saj se izkaže, da je v prav vsakem letu prisotnih 4 ali 5 takih mesecev.[6] To, kateri meseci so dobri, pa je seveda odvisno od tega, na kateri dan v tednu se leto začne in ali je prestopno. Lahko si torej zamislimo podprogram, ki bo izpisal dobre mesece v danem obdobju:

```
void DobriMeseci(int letoOd, int letoDo)
{
  static const int dolzina[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  int leto, mesec, d, prvi;
  for (leto = letoOd; leto <= letoDo; leto++) for (mesec = 1; mesec <= 12; mesec++) {
    /* Določimo dolžino meseca in na kateri dan v tednu se začne. */
    d = dolzina[mesec - 1];
    if (mesec == 2 && leto % 4 == 0 && (leto % 100 != 0 || leto % 400 == 0)) d++;
    prvi = DanVTednu(1, mesec, leto); /* 1 = ponedeljek, ..., 6 = sobota, 7 = nedelja */
    /* Mesec je dober, če je dolg vsaj 29 dni; in kolikor dni je daljši od 29, toliko
       dni pred soboto je še primernih za prvi dan meseca (če naj bo mesec dober). */
    if (d >= 29 && prvi >= 6 - (d - 29))
      printf("%d. %d\n", d, prvi, mesec, leto); }
}
```

Če hočemo le prešteti dobre mesece v nekem daljšem obdobju, pa lahko razmišljamo enako kot prej pri pravokotnih mesecih (funkcija KolikoPravokotnihMesecev zgoraj); enako kot tam namreč tudi pri dobrih mesecih velja, da če letnici prištejemo 400, se število dobrih mesecev v letu nič ne spremeni.

4. DNSx20

Vhodne podatke lahko beremo znak za znakom; spodnji program za to uporablja standardno C-jevo funkcijo fgetc. Za vsak znak preverimo, če je črka (velika ali mala); če ni črka, ga pustimo nespremenjenega, če pa je črka, se s pomočjo funkcijo Kovanec odločimo, ali bi jo spremenili (iz velike v malo ali obratno) ali ne. Spodnja rešitev predpostavlja, da nas zanimajo le črke angleške abecede in da naš program uporablja nabor znakov ASCII (ali nekaj kompatibilnega). V njem imajo namreč velike črke angleške abecede kode od 65 do 90, male pa od 97 do 122, tako da se koda velike črke od kode pripadajoče male črke razlikuje le po tem, da je v prvi bit 5 (torej bit z vrednostjo 32) ugasnjen, v drugi pa prižgan; zato lahko črko c spremenimo iz velike v malo ali obratno preprosto tako, da obrnemo bit 5, torej izračunamo c ^ 32.

```
#include <stdio.h>
int main()
{
  int c; while ((c = fgetc(stdin)) != EOF && c != '\n')
    fputc(c ^ (((’A’ <= c && c <= ’Z’) || (’a’ <= c && c <= ’z’)) && Kovanec() ?
                                                                  32 : 0), stdout);
}
```

[6] Natančneje, če se leto začne na sredo ali nedeljo, ali pa se začne na soboto in je prestopno, ali pa se začne na četrtek in ni prestopno, potem bo imelo pet dobrih mesecev, sicer pa štiri.

Lahko pa si pomagamo tudi s funkcijami iz standardne knjižnice našega programskega jezika; tam bomo zelo verjetno našli funkcije za preverjanje, ali je nek znak velika ali mala črka (v C-ju sta to isupper in islower) in za pretvorbo velike črke v malo ali obratno (v C-ju sta to tolower in toupper). Tako pridemo na primer do naslednje rešitve v C-ju:

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

int main()
{
  int c;
  while ((c = fgetc(stdin)) != EOF && c != '\n') {
    if (islower(c))       { if (Kovanec()) c = toupper(c); }
    else if (isupper(c)) { if (Kovanec()) c = tolower(c); }
    fputc(c, stdout); }
}
```

5. CamelCase

Poleg trenutnega znaka (ki ga preberemo v spremenljivko c) si je koristno zapomniti še prejšnjega (v spremenljivki cp). Prebrane znake načeloma izpišemo brez sprememb, razen če opazimo, da je prejšnji znak črka, trenutni znak pa velika črka; tedaj moramo pri izpisu spremeniti trenutni znak v malo črko in pred njega (torej med prejšnjim in trenutnim znakom) izpisati še podčrtaj. Pri preverjanju, ali je nek znak (velika) črka, in spreminjanju velike črke v malo, pridejo prav funkcije iz standardne knjižnice; v C-ju so to isalpha, isupper in tolower.

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

int main()
{
  int c, cp = −1;
  while ((c = getc(stdin)) != EOF) {
    if (isalpha(cp) && isupper(c)) {          /* Če smo med črko in veliko črko... */
      putchar('_'); putchar(tolower(c)); }    /* ...potem vrinimo podčrtaj. */
    else putchar(c);
    cp = c; }
  return 0;
}
```

Podobno se lahko lotimo tudi obratne pretvorbe, torej iz podčrtajev v camel case. Tudi tu beremo vhodne podatke beremo znak za znakom, trenutni znak imamo v c, prejšnjega pa v cp. Prebrane znake sproti sproti izpisujemo, razen če naletimo na podčrtaj, pri čemer je bil prejšnji znak črka; tedaj najprej preberemo še naslednji znak in če je ta velika črka, ga pri izpisu spremenimo v malo, podčrtaja pa ne izpišemo. Če pa znak za podčrtajem ni velika črka, lahko oba izpišemo brez sprememb. Paziti moramo še na primer, če se podčrtaj pojavlja kot zadnji znak vhoda (in torej za njim pride EOF).

```
int main()
{
```

```
  int c, cp = −1;
  while ((c = getc(stdin)) != EOF) {
    if (c != '_' || ! isalpha(cp)) { putchar(c); cp = c; continue; }
    c = getc(stdin); cp = c; /* Smo pri podčrtaju; preberimo še naslednji znak. */
    /* Če je naslednji znak velika črka, jo izpišimo kot malo in podčrtaj izpustimo. */
    if (isalpha(c)) { putchar(toupper(c)); continue; }
    /* Če znak za podčrtajem ni bila črka, izpišimo oba znaka brez sprememb. */
    putchar('_');
    if (c == EOF) break; else putchar(c); }
  return 0;
}
```

Naloga sicer žal ne predpisuje preveč natančno, kako naj naš program ravna v raznih posebnih primerih. Na primer: ali naj iz `_ena_dve` naredimo `_enaDve` (kot naša rešitev) ali `EnaDve`? Ali naj iz `ena__dve` naredimo `ena_Dve` ali celo `enaDve` (naša rešitev ne naredi nič od tega)? Ali naj iz `1_dve` naredimo `1Dve` (naša rešitev tega ne naredi)? Če bi hoteli v teh primerih doseči drugačno obnašanje, bi morali našo rešitev še kaj spremeniti.

6. Galci in Rimljani

Naj bo n skupno število vojščakov. Naloga pravi, da jih moramo razdeliti na dve legiji s po A oz. B vojščaki, pri čemer mora biti $|A - B| \leq 1$. Recimo, da je n sodo število, $n = 2k$. Tedaj je mogoče omejitev $|A - B| \leq 1$ izpolniti le tako, da je $A = B = k$, kajti če bi bila ena od teh legij velika vsaj $k+1$ vojščakov, bi morala biti druga velika $k-1$ ali manj vojščakov, tako da bi se legiji po velikosti razlikovali vsaj za 2, ne le za 1. V tem primeru torej nimamo veliko izbire; vsaka legija bo morala imeti natanko $n/2$ vojščakov. Če vhodni seznam uredimo tako, da pridejo vsi Galci na začetek, za njimi pa vsi Rimljani, lahko zdaj vzamemo prvih $n/2$ vojščakov in iz njih naredimo eno legijo, iz ostalih $n/2$ pa drugo. Na ta način bo gotovo ena od legij popolnoma homogena (sestavljena iz vojščakov večinske narodnosti), v drugi pa bodo preostali vojščaki večinske narodnosti in vsi vojščaki manjšinske narodnosti. (Če je Galcev in Rimljanov obojih enako število, pa bosta obe legiji popolnoma homogeni.)

Razmislimo zdaj še o primeru, ko je n liho število, torej $n = 2k + 1$. V tem primeru nas pogoj $|A - B| \leq 1$ pripelje do tega, da bo morala imeti ena od legij $k+1$ vojščakov, druga pa k vojščakov. Vprašanje pa je, katera od obeh legij naj bo večja in katera manjša. Pri tem si lahko pomagamo z naslednjim razmislekom: recimo, da bi imeli k Galcev in $k + 1$ Rimljanov. V tem primeru je vsekakor smiselno, da je legija (a) (tista, ki je pretežno galska) velika k vojščakov, legija (b) (tista, ki je pretežno rimska) pa $k + 1$, saj bosta tako lahko obe popolnoma homogeni, če pa bi njuni velikosti zamenjali (da bi imela galska legija $k + 1$ vojščakov, rimska pa k), bi bila galska legija mešana (ker bi bil v njej en Rimljan). To opažanje lahko posplošimo v zaključek, da je najboljša delitev ta, pri kateri večjo legijo zapolnimo v celoti s predstavniki večinske narodnosti, manjšo legijo pa z vsemi ostalimi vojščaki.

Naša spodnja rešitev hrani imena kar v tabeli, saj naloga pravi, da jih je največ 1000 in da so dolga po največ 10 znakov. Poleg tega bomo imena povezali še v dva seznama, enega za Galce in enega za Rimljane; spremenljivki `galci` in `rimljani` kažeta na začetek vsakega od teh seznamov (torej prvega Galca in prvega Rimljana), tabela

nasl[i] pa nam pove številko vojščaka, ki je v seznamu ustrezne narodnosti naslednji za vojščakom i (število −1 označuje konec seznama). Ob branju vhodnih podatkov torej shranjujemo imela v tabelo in gradimo oba seznama, obenem pa tudi štejemo vojščake vsake narodnosti; tako na koncu vemo, da imamo nG Galcev in nR Rimljanov in ni težko določiti, koliko vojščakov mora imeti prva legija (torej A). Nato moramo le še izpisati najprej vse Galce in nato vse Rimljane, po prvih A izpisanih vojščakih pa izpišemo še #, ki označuje začetek druge legije.

```
#include <stdio.h>
#include <string.h>
#define MaxN 1000

int main()
{
  char imena[MaxN][11]; int nasl[MaxN], n, nG, nR, galci = -1, rimljani = -1, i, j, d, A;

  for (n = 0; gets(imena[n]); n++) {
    /* Poglejmo, ali je ime galsko ali rimsko, in ga dodajmo v ustrezni seznam. */
    d = strlen(imena[n]);
    if (d > 2 && imena[n][d - 1] == 'x' && imena[n][d - 2] == 'i')
      nasl[n] = galci, galci = n, nG++;
    else
      nasl[n] = rimljani, rimljani = n, nR++; }

  /* Zdaj imamo n vojščakov, od tega nG Galcev in nR Rimljanov.
     Kako velika mora biti prva (pretežno galska) legija? */
  if (n % 2 == 0) A = n / 2;
  else if (nG > nR) A = (n + 1) / 2;
  else A = n / 2;

  /* Izpišimo rezultate. Najprej izpišemo vse Galce, nato vse Rimljane;
     po prvih A izpisanih imenih pa moramo izpisati še #, ki označuje konec
     prve in začetek druge legije. Zato v n zdaj štejmo, koliko imen smo že izpisali. */
  for (i = 0, n = 0; i < 2; i++)
    for (j = (i == 0) ? galci : rimljani; j >= 0; j = nasl[j]) {
      printf("%s\n", imena[j]);
      if (++n == A) printf("#\n"); }

  return 0;
}
```

Kot zanimivost omenimo, da v matematiki obstaja dobro znan način merjenja homogenosti množic, namreč entropija. Če imamo množico primerov (v našem primeru vojščakov), opisanih z atributoma X in Y (v našem primeru sta to narodnost in legija), je pogojna entropija $H(Y|X)$ mera negotovosti, ki ostane glede vrednosti atributa Y, če za nek primer že poznamo vrednost atributa X. V našem primeru to pomeni, kolikšna je negotovost glede narodnosti vojščaka, če že vemo, v katero legijo spada; očitno je ta negotovost 0, če sta obe legiji popolnoma homogeni. V splošnem je pogojna entropija definirana kot $H(Y|X) = -\sum_x \sum_y P(X = x, Y = y) \ln \frac{P(X=x,Y=y)}{P(X=x)}$, pri čemer gre vsota po vseh možnih vrednostih atributov X in Y, funkcija $P(\cdot)$ pa označuje verjetnost, da glede vrednosti atributov velja pogoj v oklepajih. Izkaže se, da dosežemo najmanjšo možno pogojno entropijo $H(\textit{narodnost}\,|\,\textit{legija})$ ravno s takšno razdelitvijo vojščakov na dve legiji, kot je opisana v naši rešitvi.

7. Vagoni

Naloga pravi, da sta oba vlaka enako dolga, recimo n vagonov (mednje štejemo tudi lokomotivo). Vhodna niza sta torej oblike $a = a_1 a_2 \ldots a_n$ in $b = b_1 b_2 \ldots b_n$. Recimo, da je a tisti vlak, ki je trenutno levo od opazovalca (in se bo peljal v desno), b pa vlak, ki je trenutno desno od opazovalca (in se bo peljal v levo). Ko se začneta vlaka premikati, bosta pred opazovalcem torej najprej vagona (pravzaprav lokomotivi) a_n in b_1; nato vagona a_{n-1} in b_2; in tako naprej; nazadnje sta pred opazovalcem vagona a_1 in b_n, po tistem pa se vlaka dokončno premakneta eden mimo drugega. Naloga sprašuje po tem, pri koliko izmed teh parov vagonov sta oba vagona v paru nizka.

Hitro lahko tudi opazimo, da če bi vlogi obeh vlakov obrnili in bi bil torej a tisti vlak, ki je na začetku desno od opazovalca in se premika v levo, b pa vlak, ki je na začetku levo od opazovalca in se premika v desno, bi imeli še vedno opravka z istimi pari vagonov kot prej, le da v obratnem vrstnem redu. Za našo rešitev je torej vseeno, kateri vlak je kateri.

Oglejmo si zdaj primer implementacijerešitve v C-ju. Dolžino vhodnih nizov (torej n) lahko poiščemo s standardno funkcijo strlen, nato pa se z zanko po i sprehodimo po nizih in za vsak par vagonov preverimo, če sta oba vagona nizka; število takšnih parov vzdržujemo v spremenljivki r, ki tako na koncu dobi vrednost, po kateri sprašuje naloga.

```
#include <string.h>

int Vagoni(char *a, char *b)
{
  int n = strlen(a), i, r;
  for (i = 0, r = 0; i < n; i++)
    if (a[i] == 'n' && b[n - 1 - i] == 'n')
      r++;
  return r;
}
```

8. Koren besed

Recimo, da smo že pregledali prvih $k-1$ besed in ugotovili, da je njihov koren dolg n znakov; torej se vse te besede ujemajo v prvih n znakih, ne pa tudi v $(n+1)$-vem znaku. Torej se tudi prvih k besed ne ujema v $(n+1)$-vem znaku; torej koren prvih k besed gotovo ni daljši od n znakov. Zdaj moramo le še preveriti, v koliko prvih znakih (vendar največ n) se k-ta beseda ujema s prejšnjimi $k-1$ besedami (zanje že vemo, da se v teh znakih vse ujemajo med seboj), pa bomo dobili dolžino korena prvih k besed. Nato se lahko začnemo ukvarjati z naslednjo (torej $(k+1)$-vo) besedo in tako naprej. Tako dobimo glavno zanko po k (ki gre po vseh besedah), pri vsaki besedi pa imamo še notranjo zanko, ki primerja znake trenutne besede z znaki dosedanjih besed (na primer kar s prvo besedo tabele), dokler bodisi ne pregleda n znakov ali pa opazi neujemanje. Poseben primer nastopi pri prvi besedi, kajti ko imamo eno samo besedo, je ta kar sama sebi koren (torej bo n enak dolžini te besede).

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

void Koren(const char *besede[], int stBesed)
```

```
{
  int i, k, n = 0;

  for (k = 0; k < stBesed; k++) {
    if (k == 0) { n = strlen(besede[k]); continue; }
    /* Koren besed od 0 do k − 1 je dolg n znakov.
       Poglejmo, v koliko od teh znakov se tudi beseda k ujema s prejšnjimi. */
    i = 0; while (i < n && besede[k][i] == besede[0][i]) i++;
    n = i; }

  /* Zdaj vemo, da je koren vseh vhodnih besed dolg n znakov; izpišimo ga. */
  for (i = 0; i < n; i++) fputc(besede[0][i], stdout);
  fputc('\n', stdout);
}
```

Lahko pa vrstni red zank tudi obrnemo; namesto zunanje zanke po besedah in notranje po črkah imamo lahko zunanjo zanko po črkah (torej po n) in notranjo po besedah (torej po k). V vsaki iteraciji glavne zanke se sprašujemo, ali se vse besede ujemajo v znaku n; če se ne, se moramo ustaviti in vemo, da smo našli dolžino korena (najdaljšega skupnega prefiksa vseh danih besed). Za to preverjanje uporabimo notranjo zanko po besedah; ta primerja n-ti znak vsake besede z n-tim znakom prve besede in se ustavi, čim opazi kakšno neujemanje. Paziti moramo še na možnost, da kakšna beseda n-tega znaka sploh nima, ker je prekratka; tudi to moramo obravnavati kot neujemanje (sicer bi lahko prišli v težave v primerih, ko so vse vhodne besede popolnoma enake).

```
void Koren2(const char *besede[], int stBesed)
{
  int i, k, n;

  for (n = 0; ; n++) {
    /* Vse vhodne besede se ujemajo v prvih n znakih.
       Ali se ujemajo tudi v znaku n? */
    for (k = 0; k < stBesed; i++)
      if (! besede[k][n] || besede[k][n] != besede[0][n]) break;

    /* Če se je zanka po k končala, še preden je pregledala vse besede,
       potem vemo, da se ne ujemajo vse v znaku n. */
    if (k < stBesed) break; }

  /* Zdaj vemo, da je koren dolg n znakov, in ga lahko izpišemo. */
  for (i = 0; i < n; i++) fputc(besede[0][i], stdout);
  fputc('\n', stdout);
}
```

Lepo pri tej rešitvi je, da pri vsaki besedi pregleda največ $n + 1$ črk, če je n prava dolžina korena vseh vhodnih besed. Prejšnja rešitev pa lahko pri prvih nekaj besedah pregleda tudi več črk, dokler spremenljivka `n` ne pade blizu prave dolžine korena. Ta razlika bi prišla zelo do izraza v primeru, če bi bile vse besede dolgi nizi oblike `aaa...aa`, razen zadnje besede, ki bi bila `b`. Prva rešitev bi pregledala vse besede v celoti in šele pri tistem `b`-ju na koncu seznama ugotovila, da je koren kar prazen niz; druga rešitev pa bi že pri $n = 0$ opazila neujemanje in se z ostalimi znaki naših vhodnih besed sploh ne bi ukvarjala.

9. Zemljevid

Podatke o trenutnem stanju prikaza bomo hranili v nekaj globalnih spremenljivkah:

sirina in visina hranita velikost zaslona v pikslih, xc in yc sta koordinati (v kilometrih) točke zemljevida, ki jo trenutno vidimo na sredi zaslona, kmNaPx pa je trenutna ločljivost (kilometrov na piksel). Koordinate središča (namesto npr. zgornjega levega kota) je koristno hraniti zato, ker se ob povečavi ne spreminjajo.

```
int sirina, visina;
double xc, yc, kmNaPx;
```

Nato pripravimo pomožni podprogram, ki bo poklical funkcijo Narisi; slednja zahteva koordinate (v kilometrih) zgornjega levega in spodnjega desnega kota slike, torej jih moramo še izračunati iz koordinat središča (ki ju imamo v globalnih spremenljivkah xc in yc). Pri tem upoštevamo, da je slika velika visina × sirina pikslov in da vsak piksel predstavlja kmNaPx kilometrov:

```
void Osvezi() {
  Narisi(xc − 0.5 * sirina * kmNaPx, yc − 0.5 * visina * kmNaPx,
         xc + 0.5 * sirina * kmNaPx, yc + 0.5 * visina * kmNaPx,
         kmNaPx); }
```

Zdaj se lahko lotimo pisanja treh podprogramov, ki jih od nas zahteva naloga. Vsak od njih primerno popravi vrednosti naših globalnih spremenljivk Init shrani v globalne spremenljivke velikost zaslona in začetno ločljivost, poleg tega pa izračuna koordinati središča slike (kot parametra namreč dobi koordinati zgornjega levega kota):

```
void Init(int sirinaZaslona, int visinaZaslona, double x1, double y1, double kmNaPiksel) {
  sirina = sirinaZaslona; visina = visinaZaslona; kmNaPx = kmNaPiksel;
  xc = x1 + 0.5 * sirinaZaslona * kmNaPx;
  yc = x1 + 0.5 * visinaZaslona * kmNaPx;
  Osvezi(); }
```

Pri premiku se spremenita koordinati središča slike v globalnih spremenljivkah xc in yc. Tidve sta izraženi v kilometrih, funkcija Premik pa dobi kot vhod premik v pikslih, zato ga moramo najprej še pomnožiti s kmNaPx:

```
void Premik(int deltaX, int deltaY) {
  xc += deltaX * kmNaPx; yc += deltaY * kmNaPx;
  Osvezi(); }
```

Ostane nam še funkcija Povecava. Če hočemo, da bodo stvari na sliki videti dvakrat večje kot prej, mora po novem en piksel predstavljati pol manj kilometrov kot prej. V splošnem moramo torej kmNaPx deliti s faktorjem, ki smo ga dobili kot parameter:

```
void Povecava(double faktor) {
  kmNaPx /= faktor;
  Osvezi(); }
```

10. Semaforji

V zanki se sprehodimo po semaforjih in v spremenljivki cas računajmo čas potovanja. Pri vsakem semaforju najprej prištejemo čas vožnje od prejšnjega semaforja do njega, torej r[i]. Nato pogledamo, kam v ciklu tega semaforja pade naš prihod: semafor je zelen v časovnih intervalih oblike $[t_i + k \cdot p_i, t_i + d_i + k \cdot p_i]$. Naš čas prihoda T pade v tak interval natanko tedaj, ko pade $T - t_i$ v nek interval oblike $[k \cdot p_i, d_i + k \cdot p_i]$,

to pa je natanko tedaj, ko je $(T - t_i) \bmod p_i$ z intervala $[0, d_i]$. Spodnji podprogram izračuna to vrednost v spremenljivki casOdZadnjeZelene. Če pade naš prihod v čas, ko je semafor rdeč, moramo počakati toliko časa, kolikor traja preostanek trenutnega cikla (torej p_i − casOdZadnjeZelene).

```
int CasVoznje(int n, int t[], int d[], int p[], int r[])
{
  int i, cas = 0, casOdZadnjeZelene;
  for (i = 0; i < n; i++)
  {
    cas += r[i];                /* Dodajmo čas vožnje do semaforja i. */
    if (i == n - 1) break;      /* Če smo prišli na cilj, končajmo. */
    if (cas < t[i]) cas = t[i]; /* Če je treba, počakajmo na prvo zeleno luč. */
    casOdZadnjeZelene = (cas - t[i]) % p[i];
    if (casOdZadnjeZelene > d[i])    /* Če smo prišli do semaforja i ob rdeči luči... */
      cas += p[i] - casOdZadnjeZelene;      /* ...počakajmo na naslednjo zeleno. */
  }
  return cas;
}
```

Naloga ne pove natančno, ali moramo pri zadnjem semaforju čakati na zeleno ali ne; gornja rešitev se ustavi takoj, ko pride do zadnjega semaforja, ne glede na njegovo takratno stanje.

11. Doroteja

Hišo lahko predstavimo z grafom, pri čemer množica točk V predstavlja sobe, množica povezav E pa vrata. Barvo sobe u označimo z $b(u)$, barvo vrat med u in v (če taka vrata obstajajo) pa z $b(u, v)$.

Nalogo lahko rešimo s preiskovanjem prostora stanj. To je pravzaprav nov, večji graf, v katerem vsaka točka predstavlja po eno „stanje“ našega sistema; v našem primeru je stanje preprosto urejena četverica $(u_1, u_2, u_3, u_4) \in V^4$, ki pove, v katerih sobah se nahajajo Doroteja, lev, strašilo in drvar.

Za tako stanje ni težko preveriti, v katera stanja je mogoče iz njega priti po enem koraku. Izbrati si moramo, katera oseba bi se premaknila (recimo $i \in \{1, 2, 3, 4\}$) in kam (recimo v v, seveda če v prvotni obstaja povezava (u_i, v)); nato moramo le še preveriti, če vsaj dve od ostalih oseb stojita v sobah barve $b(u_i, v)$.

Prostor stanj lahko pregledujemo s kakšnim od znanih postopkov za pregledovanje grafa, na primer z iskanjem v širino. Pri tem postopku vzdržujemo vrsto Q, v katero dodajamo stanja, ki smo jih že odkrili, nismo pa še pregledali njihovih naslednikov. Na vsakem koraku vzamemo eno stanje iz vrste in dodamo vanjo njegove naslednike; pri tem pa si v neki množici T (v praksi bi jo predstavili s tabelo ali pa razpršeno tabelo) označujemo, katera stanja smo že videli, tako da ne bomo istega stanja obiskovali (ali dodajali v vrsto) po večkrat. Zapišimo dobljeni postopek s psevdokodo:

vhod: začetni položaj vseh štirih oseb, $z = (z_1, z_2, z_3, z_4)$;

$T := \{z\}$; $Q :=$ prazna vrsta; dodaj z v Q;
while Q ni prazna:
  naj bo $u = (u_1, u_2, u_3, u_4)$ neko stanje iz Q; pobriši u iz Q;
  (* *Preglejmo naslednike stanja u.* *)

for $i := 1$ **to** 4:
  za vsako sobo $v \in V$, za katero obstaja povezava $(u_i, v) \in E$:
    (* *Preverimo, ali se oseba i lahko premakne v v.* *)
    $k := 0$;
    for $j := 1$ **to** 4 **do if** $j \neq i$ **and** $b(u_j) = b(u_i, v)$ **then** $k := k + 1$;
    if $k < 2$ **then continue**;
    (* *Našli smo veljaven premik.* *)
    u' := stanje, ki ga dobimo, če v u spremenimo u_i v v;
    if so v u' vsa štiri stanja enaka **then return true**;
    if $u' \notin T$ **then** dodaj u' v množico T in vrsto Q;
return false;

S tem postopkom torej počasi odkrivamo vsa stanja, ki so dosegljiva iz začetnega stanja s. Čim naletimo na kakšno stanje, pri katerem so vse osebe v isti sobi, lahko vrnemo **true**; če pa se glavna zanka **while** izteče, ne da bi našli kakšno tako stanje, potem vemo, da se naši štirje liki ne morejo srečati v isti sobi, zato vrnemo **false**.

12. Zanimivi datumi

Za posamezni konkretni datum ni težko preveriti, ali je zanimiv ali ne. Spodnja rešitev si pomaga kar s funkcijo `sprintf` iz standardne knjižnice jezika C; ta nam posamezne komponente datuma (dan, mesec in leto) pretvori v niz (za pretvorbo letnice uporabimo `%04d` namesto `%d`, tako da dobimo štirimestno letnico z vodilnimi ničlami), v katerem torej zdaj vsak znak hrani po eno števko. Nato se v zanki zapeljemo po teh števkah in v tabeli `n` štejemo, kolikokrat se pojavi posamezna števka. Na koncu le preverimo, če se kakšna števka pojavi vsaj šestkrat.

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

bool JeZanimiv(int dan, int mesec, int leto)
{
  char s[9]; int n[10], i;

  /* Inicializirajmo število pojavitev vseh števk na 0. */
  for (i = 0; i < 10; i++) n[i] = 0;

  /* Zapišimo datum kot niz (z vodilnimi ničlami pri letu). */
  sprintf(s, "%d%d%04d", dan, mesec, leto);

  /* Preštejmo pojavitve vsake števke v tem nizu. */
  for (i = 0; s[i]; i++) n[s[i] - '0']++;

  /* Poglejmo, ali se kakšna števka pojavi vsaj šestkrat. */
  for (i = 0; i < 10; i++) if (n[i] >= 6) return true;
  return false;
}
```

Zdaj potrebujemo še glavni blok programa, ki z gnezdenimi zankami pregleda vse datume v opazovanem obdobju in izpiše tiste, ki so zanimivi:

```
int main()
{
  int dan, mesec, leto;
  const int dolzina[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

```
  for (leto = 1; leto <= 9999; leto++) for (mesec = 1; mesec <= 12; mesec++)
    for (dan = 1; dan <= dolzina[mesec − 1]; dan++)
      if (JeZanimiv(dan, mesec, leto)) printf("%d. %d. %04d\n", dan, mesec, leto);
  return 0;
}
```

Opazimo lahko, da je naš program predpostavil, da ima februar vedno 28 dni. O tem, da smemo prestopne dneve pri tej nalogi res ignorirati, se lahko prepričamo z naslednjim preprostim razmislekom: če k datumu 29. 2. dodamo še štirimestno letnico, bi lahko dobili šest enakih števk le, če bi bila ta letnica 2222; to pa ni prestopno leto, saj 2222 ni deljivo s 4. Zato noben prestopni dan v našem opazovanem obdobju ne pripelje do zanimivega datuma.

Namesto s sprintf bi lahko seveda števila razbili na števke tudi sami:

```
n[dan % 10]++; if (dan >= 10) n[dan / 10]++;
n[mesec % 10]++; if (mesec >= 10) n[mesec / 10]++;
for (i = 0; i < 4; i++) { n[leto % 10]++; leto /= 10; }
```

Doslej opisano rešitev lahko še malo izboljšamo. Velika večina datumov se izkaže za nezanimive; v opazovanem obdobju (torej v letih od 1 do 9999) je približno 3,65 milijona dni, od tega pa jih ima le 1231 zanimive datume. Lepo bi torej bilo, če bi lahko zanimive datume iskali kaj bolj učinkovito kot s pregledovanjem vseh možnih datumov.

V zanimivem datumu se neka števka pojavi vsaj šestkrat; v letnici so lahko največ štiri od teh pojavitev (saj ima letnica le štiri števke), torej morata biti vsaj dve pojavitvi te števke v dnevu in/ali mesecu. Recimo zdaj, da bi si izbrali nek konkreten dan in mesec in se vprašali, katere letnice nam pri tem dnevu in mesecu dajo zanimiv datum. Kot smo pravkar videli, je lahko datum zanimiv le, če se neka števka pojavi v dnevu in/ali mesecu vsaj dvakrat; če se pojavi k-krat (za $k \geq 2$), se mora potem v letnici ta števka pojaviti vsaj $(6 - k)$-krat, da bomo dobili zanimiv datum.

Letnice, ki vsebujejo vsaj $6 - k$ pojavitev neke dane števke, pa lahko elegantno generiramo z rekurzijo; v spodnjem programu počne to podprogram Izpisi. Paziti moramo le na naslednje: recimo, da je preostali del letnice (tisti, ki ga še nismo zgenerirali) dolg letoSe števk in da se mora v njem pojaviti še vsaj xSe pojavitev števke x; v tem primeru moramo za naslednjo števko (in sploh vse preostale) nujno uporabiti x, drugače pa si lahko naslednjo števko izberemo poljubno.

Glavni del programa gre torej lahko v zankah po vseh mesecih in dnevih ter za vsak par $(dan, mesec)$ preveri, katere števke se v njem pojavijo vsaj dvakrat. Za vsako tako števko potem zgenerira primerne letnice in jih izpiše.[7]

```
#include <stdio.h>

/* Spodnja funkcija v parametru leto dobi doslej zgenerirani del letnice, ki mu je treba dodati
   še letoSe števk, med temi pa mora biti vsaj xSe pojavitev števke x. Na koncu funkcija
```

[7] Pri tem nam tudi ni treba skrbeti, da bi kakšno letnico pri istem paru $(dan, mesec)$ izpisali po večkrat. To bi namreč pomenilo, da se v dnevu in mesecu vsaj dve števki pojavita po dvakrat, kar pa je mogoče le tako, da se pojavita natanko dvakrat (ker imata dan in mesec skupaj največ štiri števke), torej bi od rekurzije zahtevali, da se v letnici tista števka pojavi še vsaj štirikrat; in ker so naše letnice vse štirimestne, bi rekurzija pri vsaki od obeh števk izpisala eno samo letnico (oblike $xxxx$, če je x opazovana števka).

```
   tako dobljene letnice izpiše, skupaj z danim dnevom in mesecem. */
void Izpisi(int dan, int mesec, int x, int xSe, int leto, int letoSe)
{
  if (letoSe == 0) { /* Rekurzije je konec, izpišimo dobljeno letnico. */
    if (leto > 0) printf("%d. %d. %04d\n", dan, mesec, leto); }
  else if (xSe == letoSe) /* Nujno moramo dodati števko x. */
    Izpisi(dan, mesec, x, xSe - 1, leto * 10 + x, letoSe - 1);
  else for (int i = 0; i < 10; i++) /* Nadaljujemo lahko s poljubno števko. */
    Izpisi(dan, mesec, x, xSe - (i == x ? 1 : 0), leto * 10 + i, letoSe - 1);
}

int main()
{
  int dan, mesec, n[10], i; char s[5];
  const int dolzina[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  for (mesec = 1; mesec <= 12; mesec++)
    for (dan = 1; dan <= dolzina[mesec - 1]; dan++)
    {
      for (i = 0; i < 10; i++) n[i] = 0;
      n[dan % 10]++; if (dan >= 10) n[dan / 10]++;
      n[mesec % 10]++; if (mesec >= 10) n[mesec / 10]++;
      for (i = 0; i < 10; i++) if (n[i] >= 2) Izpisi(dan, mesec, i, 6 - n[i], 0, 4);
    }
}
```

Malo enostavnejšo, vendar še vedno precej učinkovito rešitev pa lahko dobimo z naslednjim razmislekom. V primerih, ko se neka števka x pojavi v dnevu in mesecu natanko dvakrat, smo že videli, da je primerna letnica le $xxxx$, torej rekurzije takrat niti ne potrebujemo. Primerov, ko pa se neka števka pojavi v dnevu in mesecu trikrat ali štirikrat, pa je zelo malo[8] in ne bi bilo prevelike škode, če bi za vsakega od teh parov $(dan, mesec)$ preprosto šli po vseh možnih letnicah (od 0001 do 9999) in preverjali, katere nas pripeljejo do zanimivega datuma.

13. Računovodstvo

Predpostavimo, da sta oba seznama (računov in plačil) že urejena naraščajoče po datumu. V spodnji rešitvi je datum predstavljen kar s celim številom (**int**) — predstavljajmo si, da je to zaporedna številka dneva znotraj našega poslovnega leta. V glavni zanki se s številcem ip sprehajamo po seznamu plačil, medtem pa nam ir kaže na prvi še odprti račun (torej prvi tak račun, ki doslej še ni bil v celoti pokrit). V spremenljivki ostanek vzdržujemo podatek o tem, koliko je še neuporabljenega denarja od dosedanjih plačil. Novo plačilo ip torej prištejemo k temu ostanku, nato pa lahko z notranjo zanko po ir ugotavljamo, če lahko zdaj zapremo še kakšen račun. Pri tem se ustavimo, ko denarja zmanjka ali pa se v zaporedju računov premaknemo naprej od datuma trenutnega plačila. Slednje je posledica tega, da naloga pravi, da račun ne more biti plačan prej, preden je bil izdan. (Ima pa to lahko nenavadno posledico: morebitni računi, ki imajo kasnejši datum kot zadnje plačilo, bodo na koncu leta zagotovo ostali nepokriti, četudi nam obenem mogoče ostane višek denarja.)

[8] Natančneje, takih parov $(dan, mesec)$ je sedemnajst: 11. 1., 22. 2., 11. 10, 1. 11, 10.–19. 11., 21. 11., 11. 12. in 22. 12.

```
#include <stdio.h>

typedef struct Par_ { int datum, znesek; } Par;

void Racunovodstvo(Par racuni[], int nRacunov, Par placila[], int nPlacil)
{
  int ir = 0, ip, ostanek = 0;
  for (ip = 0; ip < nPlacil; ip++)
  {
    ostanek += placila[ip].znesek;
    while (ir < nRacunov &&
      racuni[ir].datum <= placila[ip].datum &&
      racuni[ir].znesek <= ostanek) {
      printf("Račun %d je bil plačan s plačilom %d.\n", ir, ip);
      ostanek -= racuni[ir].znesek; ir++; }
  }
  if (ir < nRacunov) printf("Računi %d..%d niso bili plačani (vsaj ne v celoti).\n",
                                                              ir, nRacunov - 1);
  while (ir < nRacunov) ostanek -= racuni[ir++].znesek;
  printf("Končno stanje je %d.\n", ostanek); /* ostanek < 0 pomeni primanjkljaj */
}
```

14. Drevo

V prvi vrstici izpisa je 2^n listov drevesa, ločeni pa so s po enim presledkom (oz. piko), tako da je dolžina te vrstice $2^n + (2^n - 1) = 2^{n+1} - 1$. Enako dolge so tudi vse nadaljnje vrstice, saj je iz primera razvidno, da moramo na desni dodati toliko pik, da so vse vrstice enako dolge. Znakom vsake vrstice pripišimo od leve proti desni x-koordinate od 0 do 2^n.

Naše drevo ima $n + 1$ nivojev in vsak nivo obsega dve vrstici; v prvi so vozlišča tega nivoja, v drugi pa so vodoravne prečke, ki povezujejo po dve vozlišči prejšnje vrstice. Nivoje oštevilčimo od 0 (nivo, ki vsebuje 2^n listov) do n (nivo, ki vsebuje le koren). Nivo k ima torej 2^{n-k} vozlišč.

Označimo s z_k koordinato prvega vozlišča na nivoju k, z d_k pa razliko v x-koordinati dveh zaporednih vozlišč tega nivoja. Za $k = 0$ vidimo, da se prvi list pojavi že pri $z_0 = 0$ in da so listi ločeni samo s po enim presledkom, tako da je razlika v koordinati dveh zaporednih listov $d_0 = 2$.

Pri ostalih nivojih pa lahko razmišljamo takole: prvih nekaj vozlišč nivoja k je na koordinatah z_k, $z_k + d_k$, $z_k + 2d_k$, $z_k + 3d_k$ in tako naprej. Vsako vozlišče nivoja $k + 1$ je na pol poti med dvema sosednjima vozliščema nivoja k; prvo vozlišče nivoja k je torej na pol poti med z_k in $z_k + d_k$, torej na $z_k + d_k/2$; drugo vozlišče nivoja k pa je na pol poti med $z_k + 2d_k$ in $z_k + 3d_k$, torej na $z_k + 5d_k/2$. Tako torej vidimo, da je $z_{k+1} = z_k + d_k/2$ in $d_{k+1} = (z_k + 5d_k/2) - (z_k + d_k/2) = 2d_k$.

Iz $d_0 = 2$ in $d_{k+1} = 2d_k$ takoj vidimo, da v splošnem velja $d_k = 2^{k+1}$. Če to uporabimo v formuli za z_k, dobimo $z_k = z_{k-1} + d_{k-1}/2 = z_{k-1} + 2^{k-1} = z_{k-2} + 2^{k-2} + 2^{k-1} = \ldots = \sum_{t=0}^{k-1} 2^t = 2^k - 1$. Tako smo dobili lepi eksplicitni formuli za z_k in d_k.

Prve vrstice nivoja k ni težko izpisati: izpisati moramo 2^{n-k} vozlišč, razdalja med njimi je d_k, torej za vsakim # izpišemo $d_k - 1$ pik. Prvo vozlišče mora biti na x-koordinati z_k, zato pred prvim # izpišemo z_k pik. Ker je slika simetrična, moramo tudi za zadnjim # izpisati z_k pik (in ne $d_k - 1$).

Drugo vrstico nivoja k izpišemo na enak način, razlika je le v tem, da vsako drugo skupino pik spremenimo v enako dolgo skupino znakov `#` — tako dobimo vodoravne prečke, ki povezujejo po dve vozlišči tega nivoja.

Še en pogled na izpis druge vrstice je, da izpišemo 2^{n-k-1} skupin znakov `#`, ki so dolge po $d_k + 1$ znakov, ločene so s po $d_k - 1$ pikami, pred in za prvo skupino znakov `#` pa je po z_k pik. Slabost te ideje pa je, da moramo nivo n v tem primeru obravnavati posebej, saj tam druga vrstica ne vsebuje vodoravne prečke (ker je na tem nivoju vozlišče le eno, namreč koren drevesa).

```
/* Izpiše k pojavitev znaka c. */
void Izpisi(int c, int k) { while (k-- > 0) putchar(c); }

void Drevo(int n)
{
  int z = 0, d = 2, v, k, i, vrstica;
  for (k = 0; k <= n; k++) {
    v = 1 << (n - k); /* število vozlišč */
    for (vrstica = 1; vrstica <= 2; vrstica++) {
      Izpisi('.', z);
      for (i = 0; i < v; i++) {
        Izpisi('#', 1);
        if (i < v - 1) Izpisi(i % 2 == 1 || vrstica == 1 ? '.' : '#', d - 1); }
      Izpisi('.', z); putchar('\n'); }
    z += d / 2; d *= 2; }
}
```

Drevo lahko izpišemo tudi tako, da sestavimo pogoj, ki nam bo za vsak znak izpisa povedal, ali mora biti tam pika ali `#`. Recimo, da smo pri prvi vrstici nivoja k in nas zanima znak na položaju x; če je $x < z_k$, potem že vemo, da bo ta znak pika. Sicer pa si moramo ogledati $x - z_k$; če je to večkratnik števila d_k, potem bo naš znak `#`, sicer pa pika. Ta pogoj lahko elegantno preverjamo tudi z bitnimi operatorji, saj je $d_k = 2^{k+1}$; preveriti moramo torej le, če je spodnjih $k + 1$ bitov v dvojiškem zapisu števila $x - z_k$ ugasnjenih.

Podobno je pri drugi vrstici nivoja k; tu moramo `#` izpisati tudi v vseh tistih primerih, ko je bit $k + 1$ v dvojiškem zapisu števila $x - z_k$ ugasnjen (spodnjih $k + 1$ bitov pa ima lahko poljubno vrednost); tako nastane prečka od z_k do $z_k + d_k$, pa od $z_k + 2d_k$ do $z_k + 3d_k$ in tako naprej.

```
void Drevo2(int n)
{
  int k, x, b, z, vrstica, sirina = (1 << (n + 1)) - 1;
  for (k = 0; k <= n; k++) {
    b = 1 << (k + 1); z = (1 << k) - 1;
    for (vrstica = 1; vrstica <= 2; vrstica++) {
      for (x = 0; x < sirina; x++)
        if (x < z || x >= sirina - z) putchar('.');
        else if (((x - z) & (b - 1)) == 0) putchar('#');
        else if (vrstica == 2 && ((x - z) & b) == 0) putchar('#');
        else putchar('.');
      putchar('\n'); }}
}
```

Pogoj `x >= sirina - z` je koristen v drugi vrstici zadnjega nivoja (pri $k = n$), kjer bi brez tega pogoja neupravičeno nastala vodoravna prečka, ki bi segala od sredine vrstice do desnega roba.[9]

15. Podniz

Recimo, da je naš članek niz oblike $s = s_1 s_2 \ldots s_n$, iskani podniz pa je $p = p_1 p_2 \ldots p_k$. Naloga zahteva, da poiščemo takšno pojavitev p-ja v s-ju, v kateri sta si prvi in zadnji znak p-ja čim bližje skupaj.

Za začetek lahko torej poiščemo neko pojavitev znaka p_1 v nizu s; recimo, da je to $s_{i_1} = p_1$. Potem moramo poiskati prvo pojavitev znaka p_2 od tega mesta naprej, torej tak $i_2 > i_1$, za katerega je $s_{i_2} = p_2$; nato moramo poiskati prvo pojavitev znaka p_3 od indeksa i_2 naprej itd. Tako bomo dobili „najbolj strnjeno“ pojavitev p-ja z začetkom pri indeksu i_1. Ker vnaprej ne vemo, pri katerem i_1 začeti, bomo pač preizkusili vse indekse, na katerih se v s pojavlja znak p_1, in vrnili najboljšo med tako dobljenimi pojavitvami p-ja.

Zapišimo dobljeni postopek s psevdokodo:

```
r := ∞;
for i := 1 to n:
  if s_i ≠ p_1 then continue;
  j := i + 1; t := 2;
  while t ≤ k and j ≤ n:
    if s_j = p_t then t := t + 1;
    j := j + 1;
  if t > k: (* Ali smo našli pojavitev celega podniza p? *)
    if j − i < r: (* Če je to najboljša doslej znana rešitev, si jo zapomnimo. *)
      r := j − i;
return r;
```

Imamo torej zunanjo zanko, ki gre po i in išče v s-ju pojavitve znaka p_1; pri vsaki taki pojavitvi gremo z notranjo zanko (z j) naprej po nizu s in iščemo pojavitve ostalih znakov p-ja. Na koncu nam $j - i$ pove dolžino tako uporabljenega dela niza s; najmanjšo tako dolžino si zapomnimo v r, kjer bomo tako na koncu dobili rezultat, po katerem sprašuje naloga.

Časovna zahtevnost tega postopka je $O(n^2)$, saj gre v najslabšem primeru notranja zanka pri vsakem i vse do konca niza s (na primer, če bi bili v s vsi znaki enaki p_1, noben pa enak p_2).

Oglejmo si zdaj še primer učinkovitejše rešitve. Naj bo $f(j, t)$ največji tak indeks i, za katerega velja, da je $i \leq j$ in da se $p_1 p_2 \ldots p_t$ pojavlja kot podniz v $s_i s_{i+1} \ldots s_j$. Če bomo znali izračunati to za poljuben t, bomo na koncu v $f(j, n)$ dobili indeks, pri katerem se začne najkrajša taka pojavitev p-ja v s, ki se konča pri s_j (ali celo

[9] Mimogrede, pri zapisu izrazov z bitnimi operatorji, kot je `&`, je potrebne nekaj previdnosti, saj imajo v C-ju in sorodnih jezikih ti operatorji zelo nizko prioriteto; `&` veže šibkeje kot `==`, zato potrebujemo v izrazu oblike `(u & v) == w` oklepaje, saj bi ga brez njih prevajalnik razumel kot `u & (v == w)`. Operator `&` veže šibkeje tudi kot `-`, kar pomeni, da bi na primer `(u - v) & w` smeli pisati brez oklepajev, vendar v zgornjem podprogramu tega raje nismo počeli, da si ne bi kdo takega izraza brez oklepajev pomotoma razlagal kot `u - (v & w)`.

levo od njega, če $s_j \neq p_k$). Odgovor, po katerem sprašuje naloga, bomo torej lahko dobili preprosto tako, da bomo poiskali najmanjšo razliko $j - f(j,n) + 1$ (po vseh j).

Funkcije f ni težko računati po naraščajočih j. Recimo, da že poznamo njene vrednosti pri nekem konkretnem j, zdaj pa bi jih radi izračunali za $j+1$. Če je s_{j+1} različen od p_t, potem so pojavitve podniza $p_1 \dots p_t$ v $s_i \dots s_j s_{j+1}$ popolnoma iste kot v $s_i \dots s_j$, tako da je v tem primeru $f(j+1,t) = f(j,t)$. Če pa je $s_{j+1} = p_t$, lahko do pojavitve $p_1 \dots p_t$ v $s_i \dots s_{j+1}$ pridemo tudi tako, da začnemo s pojavitvijo $p_1 \dots p_{t-1}$ v $s_i \dots s_j$ in nato uporabimo še znak $s_{j+1} = p_t$. V tem primeru je torej $f(j+1,t) = \max\{f(j,t), f(j,t-1)\}$. Še več, ta maksimum je gotovo dosežen z drugo od teh dveh možnosti, saj se v vsaki pojavitvi podniza $p_1 \dots p_t$ v $s_i \dots s_j$ skriva tudi pojavitev $p_1 \dots p_{t-1}$ v $s_i \dots s_j$ (in je zato $f(j,t) \leq f(j,t-1)$).

Vrednosti $f(j+1,t)$ torej ni težko dobiti iz vrednosti $f(j,t)$, tako da lahko funkcijo f računamo sistematično po naraščajočih j, stare vrednosti (za $j-1, j-2$ itd.) pa sproti pozabljamo, saj jih ne bomo več potrebovali. Tako pridemo do naslednjega postopka:

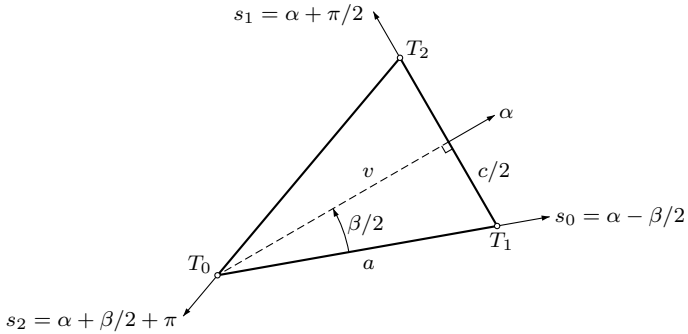
```
r := ∞;
for t := 1 to k do f[t] := −∞;
for j := 1 to k:
  (* Invarianta: za vsak t je v f[t] trenutno vrednost f(j − 1, t). *)
  for t := k downto 1:
    (* Invarianta: za vsak t' > t je v f[t'] trenutno vrednost f(j, t'),
       za vsak t' ≤ t pa je v f[t'] trenutno vrednost f(j − 1, t'). *)
    if s_j = p_t:
      if t = 1 then f[t] := j              (* f(j,1) = j, če je s_j = p_1 *)
               else f[t] := f[t − 1];      (* f(j,t) = f(j − 1, t − 1), če je s_j = p_t *)
  (* Zdaj je za vsak t v f[t] vrednost f(j,t).
     Če hočemo, da se pojavitev p-ja konča pri s_j, se mora začeti najkasneje
     na indeksu f[k]. Ali je to najboljša doslej znana rešitev? *)
  if j − f[k] + 1 < r then r := j − f[k] + 1;
return r;
```

Časovna zahtevnost je zdaj le še $O(nk)$, cena za to pa je nekaj večja poraba prostora (namreč $O(k)$ pomnilnika za tabelo f).

16. Kamere

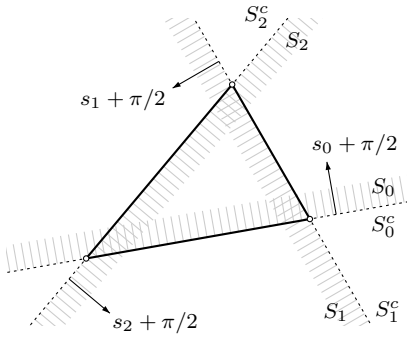
Naloga pravi, da gledajo vse kamere v isto smer in imajo enako širok zorni kot; recimo, da gledajo v smer α in imajo širino zornega kota β. Spodnja slika kaže primer trikotnega območja, ki ga pokriva ena kamera. Vrh trikotnika označimo s T_0, višina trikotnika je v, dolžina krakov je a, osnovnice pa c.[10] Kamera stoji v T_0; notranji kót pri oglišču T_0 je ravno širina zornega kota kamere, torej β. Kamera gleda v smer α; če se torej postavimo v T_0, vidimo kraka trikotnika v smereh $\alpha - \beta/2$ in $\alpha + \beta/2$, višino trikotnika pa v smeri α.

[10] Dolžine stranic pri naši nalogi sicer niso eksplicitno podane, vendar pa jih lahko izračunamo iz višine v in kota β. Če naš enakokraki trikotnik pri višini razpolovimo, dobimo pravokotni trikotnik, ki ima pri T_0 zdaj notranji kót $\beta/2$, njegove stranice pa so $c/2$, v in a. Velja torej $\cos(\beta/2) = v/a$ in $\operatorname{tg}(\beta/2) = (c/2)/v$, tako da lahko a in c računamo s formulama $a = v/\cos(\beta/2)$ in $c = 2v \operatorname{tg}(\beta/2)$.



Če začnemo v T_0 in se po enakokrakem trikotniku sprehodimo v pozitivni smeri (torej nasproti smeri urinega kazalca), vidimo najprej krak T_0T_1 (dolžine a) v smeri $s_0 := \alpha - \beta/2$; nato imamo osnovnico T_1T_2 (dolžine c), ki je pravokotna na višino; ker je višina v smeri α, ima torej T_1T_2 smer $s_1 := \alpha + \pi/2$; nato pa imamo še krak T_2T_0 (spet dolžine a); kot smo videli zgoraj, ima T_0T_2 smer $\alpha + \beta/2$, torej ima T_2T_0 ravno nasprotno smer: $s_2 := \alpha + \beta/2 + \pi$.

Pri takem sprehodu po trikotniku v pozitivni smeri je bil trikotnik ves čas na naši levi strani. Stranica s_k torej pravzaprav razdeli ravnino na dve polravnini, levo (recimo ji S_k; k njej štejmo tudi nosilko stranice s_k) in desno (to je komplement prve, torej S_k^c). Naš trikotnik pravzaprav ni nič drugega kot presek vseh treh levih polravnin: $\triangle = S_0 \cap S_1 \cap S_2$.



Doslej smo ves čas gledali le eno kamero in en trikotnik. V resnici imamo n kamer in vsaka ima svoj trikotnik, kar bomo označili z indeksom v oklepajih. Za i-to kamero imamo torej trikotnik $\triangle^{(i)}$, ki je presek polravnin $S_0^{(i)} \cap S_1^{(i)} \cap S_2^{(i)}$. Naša naloga pa sprašuje po preseku vseh trikotnikov, $Q := \cap_i \triangle^{(i)}$; to je zdaj presek vseh $3n$ polravnin oblike $S_k^{(i)}$ (za $k = 0, 1, 2$). Ker je presek komutativna operacija, ga lahko zapišemo kot $(\cap_i S_0^{(i)}) \cap (\cap_i S_1^{(i)}) \cap (\cap_i S_2^{(i)})$.

Presek polravnin $\cap_i S_k^{(i)}$ (za nek konkreten k) je enostavno računati, ker so stranice $s_k^{(i)}$, ki omejujejo te polravnine, vse vzporedne. Če pogledamo na primer polravnini $S_k^{(i)}$ in $S_k^{(j)}$, je gotovo ena vsebovana v drugi ali pa obratno, odvisno od tega, katera od premic $s_k^{(i)}$ in $s_k^{(j)}$ leži „višje“ (če gledamo pravokotno na te premice v smeri, v

kateri leži naš trikotnik glede na posamezno premico). Poiskati moramo torej tisto $s_k^{(i)}$, ki leži najvišje; njena $S_k^{(i)}$ je ravno enaka preseku $\cap_i S_k^{(i)}$. Pri tem si lahko pomagamo s skalarnim produktom: vzemimo vektor v smeri pravokotno na s_k in levo od nje (to je torej smer $s_k + \pi/2$); skalarni produkt neke točke s tem vektorjem je tem večji, čim višje v tisti smeri leži ta točka.

Tako na koncu dobimo tri polravnine, katerih presek je trikotno območje Q, ki ga iščemo. Za vsako od teh polravnin poznamo premico, ki jo omejuje; če izračunamo presek po dveh zaporednih premic, dobimo oglišča trikotnika Q. Če si ta oglišča (recimo jim T_0, T_1 in T_2) ne sledijo v pozitivni smeri, pač pa v negativni, je to znak, da je presek naših polravnin v resnici prazen, torej območja, ki bi ga snemale vse kamere, sploh ni. V ta namen si lahko pomagamo z vektorskim produktom; ta je sicer definiran za trodimenzionalne vektorje, naši pa so dvodimenzionalni, zato jim v mislih dodajmo še z-koordinato 0. Spomnimo se na definicijo vektorskega produkta: $\mathbf{u} \times \mathbf{v}$ je vektor, pravokoten na $\mathbf{u}$ in $\mathbf{v}$, njegova dolžina je enaka ploščini paralelograma, ki ga oklepata $\mathbf{u}$ in $\mathbf{v}$, njegova smer pa je določena s pravilom desne roke. Vrnimo se k našemu trikotniku in za $\mathbf{u}$ vzemimo vektor T_0T_1, za $\mathbf{v}$ pa T_0T_2. Rekli smo, da ležijo vse tri točke (in s tem tudi $\mathbf{u}$ in $\mathbf{v}$) na ravnini $z = 0$; zato je vektorski produkt $\mathbf{u} \times \mathbf{v}$ pravokoten na to ravnino, torej je to vektor oblike $(0, 0, Z)$ za $Z = u.x \cdot v.y - u.y \cdot v.x$. Iz pravila desne roke pa tudi sledi, da če je orientacija trikotnika $T_0T_1T_2$ pozitivna, potem kaže $\mathbf{u} \times \mathbf{v}$ nad ravnino $z = 0$ (torej ima $Z > 0$), sicer pa pod njo (in ima torej $Z < 0$). Tako torej vidimo, da lahko orientacijo trikotnika $T_0T_1T_2$ preverimo tako, da pogledamo predznak Z-ja. Koristna pa je tudi njegova absolutna vrednost; iz definicije vektorskega produkta vemo, da je to ravno ploščina paralelograma, ki ga določata $\mathbf{u} = T_0T_1$ in $\mathbf{v} = T_0T_2$; naš trikotnik pa je ravno polovica tega paralelograma, tako da je njegova ploščina $Z/2$.

Oglejmo si še implementacijo te rešitve v C++. Za lažje delo z dvodimenzionalnimi vektorji si pripravimo razred, ki hrani par koordinat (x, y) in podpira operacije, kot so seštevanje vektorjev in množenje s skalarjem:

```
#include <math.h>
struct Vektor {
  double x, y;
  Vektor(double X = 0, double Y = 0) : x(X), y(Y) { } };
Vektor operator + (Vektor t, Vektor u) { return Vektor(t.x + u.x, t.y + u.y); }
Vektor operator - (Vektor t, Vektor u) { return Vektor(t.x - u.x, t.y - u.y); }
Vektor operator * (double a, Vektor t) { return Vektor(a * t.x, a * t.y); }

// Vrne enotski vektor v dani smeri.
Vektor Smer(double smer) { return Vektor(cos(smer), sin(smer)); }

// Vrne skalarni produkt danih vektorjev.
double Skalarni(Vektor t, Vektor u) { return t.x * u.x + t.y * u.y; }

// Vektorski produkt vektorjev (t.x, t.y, 0) in (u.x, u.y, 0) je
// vektor oblike (0, 0, Z) in spodnja funkcija vrne ta Z.
double Vektorski(Vektor t, Vektor u) { return t.x * u.y - t.y * u.x; }
```

Potrebovali bomo tudi podprogram za računanje presečišča dveh premic. Recimo, da imamo premico skozi točko $\mathbf{t}_1$ v smeri $\mathbf{s}_1$ in premico skozi točko $\mathbf{t}_2$ v smeri $\mathbf{s}_2$. Vsaka točka na prvi premici ima torej koordinate oblike $\mathbf{t}_1 + \lambda\mathbf{s}_1$ za neko realno število λ; podobno ima vsaka točka na drugi premici koordinate oblike $\mathbf{t}_2 + \mu\mathbf{s}_2$ za neko realno

število μ. Presečišče premic je točka, ki leži na obeh premicah hkrati, torej se jo mora dati izraziti na oba načina: $\mathbf{t}_1 + \lambda \mathbf{s}_1 = \mathbf{t}_2 + \mu \mathbf{s}_2$. To ujemanje mora veljati tako pri x- kot pri y-koordinatah; tako dobimo par enačb

$$t_1.x + \lambda\, s_1.x = t_2.x + \mu\, s_2.x$$
$$t_1.y + \lambda\, s_1.y = t_2.y + \mu\, s_2.y$$

Iz prve enačbe dobimo $\lambda = (t_2.x - t_1.x + \mu\, s_2.x)/s_1.x$; če to nesemo v drugo enačbo, lahko iz nje sčasoma dobimo

$$\mu = \frac{s_1.x(t_1.y - t_2.y) - s_1.y(t_1.x - t_2.x)}{s_1.x\, s_2.y - s_1.y\, s_2.x}.$$

Ta μ moramo le še vstaviti v formulo $\mathbf{t}_2 + \mu \mathbf{s}_2$, pa dobimo koordinati presečišča obeh premic. Opazimo lahko še, da se v naši formuli za μ pravzaprav pojavljajo izrazi prav take oblike, kot jih računa naša zgoraj omenjena funkcija Vektorski. Zato je lahko naš podprogram za računanje presečišča dveh premic zelo preprost:

```
Vektor Presecisce(Vektor t1, Vektor s1, Vektor t2, Vektor s2) {
  double mu = Vektorski(s1, t1 − t2) / Vektorski(s1, s2);
  return t2 + mu * s2; }
```

Zdaj imamo vse, kar potrebujemo za glavni podprogram, ki bo zares računal presek naših trikotnikov:

```
double Kamere(int n, double alpha, double beta, double xi[], double yi[], double vi[])
{
  double u[3], smer[3]; Vektor T[3], U[3];
  const double pi = 4 * atan(1.0);
  // Izračunajmo smeri stranic, če hodimo po robu trikotnika v pozitivni smeri.
  smer[0] = alpha − beta / 2; smer[1] = alpha + pi / 2; smer[2] = alpha + beta / 2 + pi;
  for (int i = 0; i < n; i++) {
    // Izračunajmo dolžino stranic i-tega trikotnika (a je krak, c pa osnovnica).
    double a = vi[i] / cos(beta / 2), c = 2 * vi[i] * tan(beta / 2);
    // Izračunajmo koordinate oglišč tega trikotnika.
    T[0] = Vektor(xi[i], yi[i]);
    T[1] = T[0] + a * Smer(smer[0]);
    T[2] = T[1] + c * Smer(smer[1]);
    // Izračunajmo projekcije stranic na smer, pravokotno na te stranice.
    for (int k = 0; k < 3; k++) {
      double s = smer[k] + pi / 2; // pravokotna na stranico k, v smeri proti trikotniku
      double proj = Skalarni(T[k], Smer(s));
      if (i == 0 || proj > u[k]) u[k] = proj, U[k] = T[k]; }}
  // Presek vseh vhodnih trikotnikov je trikotnik, nosilke njegovih stranic pa so premice
  // skozi U[k] v smeri smer[k] za k = 0, 1, 2. Izračunajmo koordinate njegovih oglišč.
  for (int k = 0; k < 3; k++) T[k] = Presecisce(
    U[k], Smer(smer[k]), U[(k + 1) % 3], Smer(smer[(k + 1) % 3]));
  // Izračunajmo ploščino in preverimo, če so oglišča v pravi orientaciji.
  double ploscina = Vektorski(T[1] − T[0], T[2] − T[0]) / 2;
  return (ploscina < 0) ? 0 : ploscina;
}
```

17. Produkt

Recimo, da so $x_1, \ldots, x_n$ naša vhodna števila. Za vsako od teh števil poglejmo, s kakšno stopnjo se pojavita števili 2 in 5 v njegovem razcepu na prafaktorje; tako dobimo pri vsakem x_i izražavo oblike $x_i = 2^{\alpha_i} 5^{\beta_i} y_i$, pri čemer y_i ni deljiv niti z 2 niti s 5.

Naj bo $B \subseteq 1..n$ množica indeksov izbranih k števil. Naloga pravi, da bi radi B izbrali tako, da se produkt teh števil konča na čim manj ničel (v desetiškem zapisu). Ta produkt ima vrednost

$$x = \prod_{i \in B} x_i = \prod_{i \in B} 2^{\alpha_i} 5^{\beta_i} y_i = 2^{\alpha} 5^{\beta} y$$

za $\alpha = \sum_{i \in B} \alpha_i$, $\beta = \sum_{i \in B} \beta_i$ in $y = \prod_{i \in B} y_i$.

Ker y_i niso deljivi z 2 ali 5, tudi y ni in zato na število ničel na koncu x nima nobenega vpliva. Hitro se lahko prepričamo, da se desetiški zapis števila x konča na natanko $t = \min\{\alpha, \beta\}$ ničel. (Res: ker je x deljiv z 2^{α}, je tudi z 2^t; ker je deljiv s 5^{β}, je tudi s 5^t; torej je deljiv z 10^t, torej se konča na t ničel. Če bi se x končal na več kot t ničel, recimo na $u > t$ ničel, bi moral biti deljiv z 10^u, torej tudi z 2^u in 5^u, torej bi moral biti $u \leq \alpha$ in $u \leq \beta$, torej $u \leq \min\{\alpha, \beta\} = t$, kar je v protislovju z $u > t$.)

Pri različnih izborih B je seveda tudi število ničel na koncu lahko različno, zato ga bomo v nadaljevanju pisali kot $t(B)$. Naloga zdaj sprašuje po

$$\begin{aligned} & \min\{t(B) : B \subseteq 1..n, |B| = k\} \\ = \; & \min\{\min\{\textstyle\sum_{i \in B} \alpha_i, \sum_{i \in B} \beta_i\} : B \subseteq 1..n, |B| = k\} \\ = \; & \min\{\min\{\textstyle\sum_{i \in B} \alpha_i : B \subseteq 1..n, |B| = k\}, \min\{\sum_{i \in B} \beta_i : B \subseteq 1..n, |B| = k\}\}. \end{aligned}$$

V zadnji vrstici lahko seveda $\min\{\sum_{i \in B} \alpha_i : B \subseteq 1..n, |B| = k\}$ dobimo tako, da števila α_i uredimo naraščajoče in seštejemo najmanjših k. Podobno naredimo potem še z β_i in obdržimo manjšo od obeh vsot.

18. Špekulacije

(a) S tremi gnezdenimi zankami lahko sistematično pregledamo vse možne trojice (a, b, c) in pri vsaki preverimo, ali je zaporedje menjav $a \to b \to c \to a$ profitni trikotnik ali ne. Paziti moramo še na to, da so vse tri valute v trikotniku različne. Nekaj časa lahko prihranimo tudi s tem, da se omejimo na trojice, pri katerih je a največje število v trikotniku; s tem zagotovimo, da bomo vsak trikotnik pregledali le enkrat namesto trikrat (kot (a, b, c), (b, c, a) in še kot (c, a, b)).

```
void Spekulant(double m[n][n])
{
  int a, b, c;
  for (a = 2; a < n; a++)
    for (b = 0; b < a; b++)
      for (c = 0; c < a; c++) if (c != b)
        if (m[a][b] * m[b][c] * m[c][a] > 1) {
          printf("%d %d %d\n", a, b, c); return; }
  printf("ni rešitve\n");
}
```

Razmislimo še o časovni zahtevnosti tega postopka. V najslabšem primeru, torej če ni nobenega profitnega trikotnika, bo šla zunanja zanka po vseh a od 2 do $n-1$, pri vsakem a pa mora pregledati a^2 parov (b,c) (za a od teh parov sicer ugotovi, da je $b=c$ in ji zato trikotnika ni treba preverjati). Tako imamo torej opravka z $\sum_{a=2}^{n-1} a^2$ trojicami, kar se z malo telovadbe izkaže za enako $(n-1)n(2n-1)/6-1 = \frac{1}{3}n^3-\frac{1}{2}n^2+\frac{1}{6}n-1$; trikotnikov pa moramo pregledati največ $\sum_{a=2}^{n-1} a(a-1)$, kar je enako $n(n-1)(n-2)/3 = \frac{1}{3}n^3-n^2+\frac{2}{3}n$. Do istega števila trikotnikov lahko pridemo tudi po drugi poti: izmed n valut lahko tri različne valute izberemo na $\binom{n}{3}$ načinov, taka trojica pa nam dá dva trikotnika ($a\to b\to c\to a$ in $a\to c\to b\to a$); tako imamo torej $2\cdot\binom{n}{3}$ trikotnikov, kar je spet enako $n(n-1)(n-2)/3$.

(b) Če smo profitne trikotnike iskali s tremi gnezdenimi zankami, bi lahko profitne cikle k točk seveda iskali s k gnezdenimi zankami; težava pa je, da dobimo k kot parameter, torej vnaprej ne vemo, koliko gnezdenih zank potrebujemo. Zato je bolje namesto zank uporabiti rekurzijo: ob vsakem rekurzivnem klicu preizkusimo vse možne načine, kako dodati novo valuto v cikel (pri tem pazimo, da se razlikuje od dosedanjih), in nato z rekurzijo pregledamo še vsa možna nadaljevanja cikla. Sproti lahko tudi računamo zmnožek dosedanjih menjav v ciklu, tako da na koncu ne bomo imeli preveč dela s preverjanjem, ali je cikel profitni ali ne. Podobno kot pri (a) lahko tudi tu vpeljemo omejitev, da mora imeti prva valuta v ciklu večjo številko od ostalih; tako bomo preprečili, da bi isti cikel obdelali po večkrat.

V spodnji rešitvi se rekurzija dogaja v podprogramu NadaljujCikel, ki mu parameter t pove, koliko valut je doslej v ciklu, produkt pa je produkt dosedanjih t − 1 menjav.

```
bool NadaljujCikel(double m[n][n], int cikel[], int t, int k, double produkt)
{
  int i, u; double noviProdukt;
  for (u = 0; u < cikel[0]; u++)
  {
    /* Ali je valuta u različna od vseh dosedanjih v ciklu? */
    for (i = 1; i < t; i++) if (u == cikel[i]) break;
    if (i < t) continue;
    /* Dodajmo u v cikel. */
    cikel[t] = u; noviProdukt = produkt * m[cikel[t - 1]][u];
    /* Če cikel še ni dosegel dolžine k, nadaljujmo z rekurzijo. */
    if (t + 1 < k) {
      if (NadaljujCikel(m, cikel, t + 1, k, noviProdukt)) return true; }
    /* Sicer pa preverimo, ali je profitni. */
    else if (noviProdukt * m[u][cikel[0]] > 1) {
      /* Cikel je profitni, torej ga izpišimo. */
      for (i = 0; i < k; i++) printf("%d%c", cikel[i], (i == k - 1 ? '\n' : ' '));
      return true; }
  }
  return false;
}

void Spekulant2(double m[n][n], int k)
{
  int cikel[n], a;
  for (a = k - 1; a < n; a++) {
    cikel[0] = a;
    if (NadaljujCikel(m, cikel, 1, k, 1)) return; }
```

```
  printf("ni rešitve\n");
}
```

Postopek bi lahko še malo pospešili, če bi v še eni tabeli (ali pa kar v neuporabljenem delu tabele cikel) vzdrževali seznam valut (z območja od 0 do cikel$[0]-1$), ki jih v doslej sestavljenem delu cikla še nismo uporabili; potem nam ne bi bilo treba iti z u po vseh valutah od 0 do cikel$[0]-1$ in pri vsaki z gnezdeno zanko po i pregledovati, ali se je ta u že pojavil kdaj prej v ciklu; morali bi iti le po seznamu neuporabljenih valut in bi za vsako od njih že vedeli, da je primerna za nadaljevanje cikla.

V vsakem primeru pa ima naš postopek eksponentno časovno zahtevnost. Izmed n valut lahko izberemo k (različnih) valut na $\binom{n}{k}$ načinov in vsako tako skupino k valut lahko na $(k-1)!$ načinov sestavimo v cikel. Možnih ciklov je torej $\binom{n}{k}\cdot(k-1)! = n!/[k(n-k)!] = n(n-1)\dots(n-k+1)/k$ in če nobeden od njih ni profitni, bo naša rešitev prej ali slej pregledala prav vse te cikle.

Do še učinkovitejše rešitve (ki pa porabi precej več pomnilnika) lahko pridemo z dinamičnim programiranjem; oštevilčimo valute od 1 do n in definirajmo podprobleme takole: za vsako $A \subseteq \{1,\dots,n\}$ in $s,t \in A$ naj bo $f(s,t,A)$ najboljša profitna pot (ne cikel) od s do t, ki pri tem uporabi vse valute iz A (vsako natanko enkrat). Te poti lahko računamo s formulo $f(s,t,A) = \max\{f(s,u,A-\{t\})m[u][t] : u \in A-\{s,t\}\}$. Če tako pot zaključimo v cikel, ima dobiček $f(s,t,A)m[t][s]$; poiskati moramo maksimum tega po vseh $|A| = k$.[11]

Rešitve s polinomsko časovno zahtevnostjo do nadaljnjega žal ne poznamo, saj je naš problem NP-težak. O tem se lahko prepričamo tako, da nanj prevedemo problem Hamiltonovega cikla, ki je znan NP-težak problem. Naj bo G graf z n točkami, v katerem iščemo Hamiltonov cikel; mislimo si zdaj nabor n valut, po eno za vsako točko grafa, in menjalno matriko m, definirano takole: $m[u][v] = 2$, če je graf G vsebuje povezavo $u \to v$; sicer pa naj bo $m[u][v] = 1/2^n$.

Prepričajmo se, da je v tej menjalni matriki prisoten profitni cikel dolžine n natanko tedaj, ko je v prvotnem grafu G prisoten nek Hamiltonov cikel. ($\Rightarrow$) Če imamo cikel n menjav in ima vsaj ena od teh menjav vrednost $1/2^n$, potem je produkt po celem ciklu gotovo manjši od 1, kajti ostalih menjav je le $n-1$ in četudi ima vsaka od njih vrednost 2, bo produkt cikla še vedno zgolj $(1/2^n)\cdot 2^{n-1} = 1/2$. Profitni cikel dolžine n je torej mogoč le tako, da imajo vse menjave na njem vrednost 2; to pa pomeni, da za vsako od teh menjav obstaja tudi povezava v grafu G, torej je v grafu G prisoten Hamiltonov cikel. ($\Leftarrow$) Če je v G prisoten Hamiltonov cikel, potem vsaki povezavi tega cikla ustreza v naši menjalni matriki neka menjava z vrednostjo 2, tako da te menjave tvorijo cikel z vrednostjo 2^n, ki je torej profitni cikel dolžine k.

Tako torej vidimo, da če bi imeli algoritem, ki v polinomskem času išče profitne cikle za poljuben k, bi lahko z njim v polinomskem času rešili problem Hamiltonovega

[11] Časovno zahtevnost lahko ocenimo takole: vemo, da moramo pregledati vse množice A z največ k elementi; pri velikosti r si lahko A izberemo na $\binom{n}{r}$ načinov, nato pa s in t (iz A) vsakega na r načinov; tako imamo $r^2\binom{n}{r}$ podproblemov, z vsakim pa $O(r)$ dela. Skupna časovna zahtevnost je torej reda velikosti $S = \sum_{r=1}^{k} r^3\binom{n}{r}$; če je $k \le n/2$, velja $\binom{n}{r} \le \binom{n}{k}$ in zato $S \le k^3\binom{n}{k}$, kar je bolje od $(k-1)!\binom{n}{k}$ pri prejšnji rešitvi, razen za majhne k (do $k=6$). Če pa je $k \le n/2$, si lahko pomagamo z $\binom{n}{r} \le \binom{n}{n/2}$, kar je po Stirlingovi formuli $\approx 2^{n+1}/\sqrt{2\pi n}$, tako da je $S \le k^3 2^{n+1}$; izboljšava v primerjavi s prejšnjo rešitvijo je tu še bolj dramatična. Slabost te rešitve z dinamičnim programiranjem je, da porabi eksponentno veliko pomnilnika za shranjevanje rešitev podproblemov $f(s,t,A)$.

cikla; ker je slednji NP-težak, je torej tudi naš problem profitnih ciklov NP-težak.

(*c*) Ravnokar smo videli eno od povezav med našim problemom in grafi; nekaj podobnega bo prišlo prav tudi za nalogo (*c*). Vzemimo naš problem profitnih ciklov in sestavimo usmerjen graf G z n točkami, po eno za vsako valuto. Graf naj bo poln, torej imamo povezavo $u \to v$ za vsak možen par točk (u, v). Vsakemu ciklu menjav seveda ustreza tudi nek cikel povezav na grafu. Spomnimo se, da je cikel $u_0 \to u_1 \to \ldots \to u_k = u_0$ profitni natanko tedaj, ko je produkt vrednosti teh menjav večji od 1, torej ko je $\prod_{i=1}^{k} m[u_{i-1}][u_i] > 1$.

Pa recimo, da bi ta pogoj logaritmirali in pomnožili z -1. Iz produkta nastane vsota: $\sum_{i=1}^{k}(-\ln m[u_{i-1}][u_i]) < 0$. Če bi v grafu G povezavam pripisali tudi dolžine $d_{uv} = -\ln m[u][v]$, bi bila opisana vsota ravno dolžina cikla $u_0 \to u_1 \to \ldots \to u_k$ v našem grafu. Tako torej vidimo, da si lahko problem iskanja profitnega cikla predstavljamo tudi kot problem iskanja cikla negativne dolžine v usmerjenem grafu G. (Da bo krajše, bomo ciklu z negativno dolžino rekli kar negativen cikel.)

Z negativnimi cikli se pri grafih srečujemo predvsem takrat, ko iščemo najkrajše poti v grafu; če namreč lahko pot speljemo prek negativnega cikla, potem najkrajše poti sploh ni, saj lahko pot poljubno skrajšamo, če se večkrat zapeljemo po ciklu. Za naš problem je koristno, če začnemo s Floyd-Warshallovim algoritmom za iskanje najkrajših poti med vsemi pari točk; oštevilčimo točke grafa od 1 do n in naj bo $d_{uv}^{(k)}$ dolžina najkrajše take poti od u do v, ki med tema dvema točkama ne obišče nobene točke iz $\{k+1, \ldots, n\}$. Pri $k = 0$ so torej dovoljene le take poti, ki gredo iz u naravnost v v, tako da je $d_{uv}^{(0)} = d_{uv}$, če ima graf povezavo $u \to v$, sicer pa je $d_{uv}^{(0)} = \infty$. Pri večjih k lahko razmišljamo takole: pot od u do v lahko gre skozi k ali pa ne; če ne gre, sme uporabljati le točke od 1 do $k-1$, najkrajšo tako pot pa že imamo v $d_{uv}^{(k-1)}$; če pa gre skozi k, potem imamo pravzaprav dve poti: od u do k in od k do v, pri čemer vmes uporabljamo le točke od 1 do $k-1$. Tako smo dobili zvezo $d_{uv}^{(k)} = \min\{d_{uv}^{(k-1)}, d_{uk}^{(k-1)} + d_{kv}^{(k-1)}\}$. Te dolžine lahko računamo sistematično s tremi gnezdenimi zankami (po k, u in v).

Negativni cikli povzročajo temu algoritmu težave med drugim tudi zato, ker se mu prej ali slej splača kakšno pot speljati po negativnem ciklu, potem pa to v resnici ni več pot, ampak zgolj nek splošen sprehod (ker se nekatere točke na njem pojavijo več kot enkrat). Če ima graf negativne cikle, moramo torej razmišljati o tem, katere vrednosti $d_{uv}^{(k)}$ se res nanašajo na najkrajšo pot, katere pa zgolj na nek sprehod (sicer tak sprehod, ki gotovo ni daljši od najkrajše poti).

Recimo zdaj, da ima naš graf nek negativen cikel. Naj bo z tista točka cikla, ki ima največjo številko, y pa tista z drugo največjo številko. Če je negativnih ciklov več, vzemimo tistega, ki ima najmanjši y; med cikli s tem y pa vzemimo tistega z najmanjšim z; tako dobljenemu ciklu recimo C. To med drugim pomeni, da samo iz točk $\{1, \ldots, y-1\}$ ni mogoče sestaviti negativnega cikla. Na pomislek iz prejšnjega odstavka zdaj lahko odgovorimo, da se dolžine $d_{uv}^{(k)}$ za vse $k < y$ vsekakor nanašajo na poti, ne pa na neke splošnejše sprehode, saj je tak sprehod lahko krajši od poti le, če vsebuje nek negativen cikel, takega pa, kot smo videli, iz točk $\{1, \ldots, k\}$ za nek $k < y$ ni mogoče sestaviti.

Naš cikel C si lahko predstavljamo kot sestavljenega iz dveh poti: pot ρ od z do y in pot ρ' od y do z, pri čemer obe poti vmes obiskujeta le točke iz $\{1, \ldots, y-1\}$. Ravnokar pa smo videli, da je $d_{zy}^{(y-1)}$ dolžina najkrajše take poti od z do y, ki vmes

obiskuje le točke $\{1,\ldots,y-1\}$ (in podobno za $d_{yz}^{(y-1)}$); torej imamo $d_{zy}^{(y-1)} \le d(\rho)$ in $d_{yz}^{(y-1)} \le d(\rho')$, zato pa tudi $d_{zy}^{(y-1)} + d_{yz}^{(y-1)} \le d(\rho) + d(\rho') = d(C) < 0$, pri čemer zadnji korak sledi iz dejstva, da je C negativen. Toda vsota $d_{zy}^{(y-1)} + d_{yz}^{(y-1)}$ je ena od možnosti, o katerih razmišlja Floyd-Warshallov algoritem, ko računa $d_{zz}^{(y)}$ (druga možnost je $d_{zz}^{(y-1)}$); tako torej vidimo, da bo $d_{zz}^{(y)}$ vsekakor manjša od 0.

Ali se lahko zgodi, da bi se na diagonali pojavila kakšna negativna vrednost že kaj prej, torej da bi imeli $d_{uu}^{(k)} < 0$ pri nekem $k < y$? Pa vzemimo najmanjši k, pri katerem je (za nek primerno izbran u) vrednost $d_{uu}^{(k)} < 0$. Če je to $k = 0$, to pomeni, da je že v menjalni matriki vrednost $m[u][u] > 1$; to vsekakor lahko obravnavamo kot poseben primer, sploh pa ni verjetno, da bi nam kdo zamenjal nek znesek valute u za večji znesek iste valute. Pri $k > 0$ pa torej vemo, da negativna vrednost v $d_{uu}^{(k)}$ ni mogla priti iz $d_{uu}^{(k-1)}$ (saj smo rekli, da vzamemo najmanjši k, pri katerem je tam negativna vrednost), torej je morala priti iz $d_{uk}^{(k-1)} + d_{ku}^{(k-1)}$. Ker je $k-1 < y$, vemo, da sta $d_{uk}^{(k-1)}$ in $d_{ku}^{(k-1)}$ dolžini najkrajših poti (in ne nekih drugačnih sprehodov); in ker tidve poti vmes uporabljata le točke $\{1,\ldots,k-1\}$, ju lahko združimo in dobimo cikel od u do u, ki vmes uporablja le točke $\{1,\ldots,k\}$. Njegova dolžina je $d_{uu}^{(k)}$ in ker smo rekli, da je to < 0, pomeni, da smo našli negativen cikel; poleg k in u vsebuje ta cikel le točke iz $\{1,\ldots,k-1\}$; druga največja številka na ciklu je torej bodisi k (če je $u > k$) bodisi je še manjša od k (če je $u < k$), v vsakem primeru pa je torej manjša od y; tako smo prišli v protislovje (cikel C smo izbrali namreč tako, da je druga največja številka na njem najmanjša možna).

Tako torej vidimo, da (če odmislimo primere, ko so že v vhodni matriki na diagonali kakšne vrednosti, večje od 1) bo do negativne vrednosti na diagonali prvič prišlo pri $k = y$; in če je ta negativna vrednost pri $d_{zz}^{(y)}$, potem vemo, da lahko negativni cikel sestavimo tako, da gremo od z do y in od tam nazaj do z, obakrat po najkrajši taki poti, ki vmes uporablja le točke iz $\{1,\ldots,y-1\}$. Konkreten potek te poti lahko rekonstruiramo, če smo si zapomnili vrednosti $d_{uv}^{(k)}$ za $k < y$, neugodno pri tem pa je, da za to porabimo $O(kn^2)$ pomnilnika (kar je v najslabšem primeru $O(n^3)$). Še ena možnost je, da iz grafa pobrišemo vse točke razen $\{1,\ldots,y,z\}$ in v njem iščemo najkrajšo pot od z do y in nazaj z Bellman-Fordovim algoritmom (ki bo za to porabil $O(n^3)$ časa in $O(n)$ prostora).

Zapišimo naš algoritem še s psevdokodo; valute si mislimo oštevilčene 1 do n.

algoritem ŠPEKULANT(m):
  for $u := 1$ **to** n **do if** $m[u][u] > 1$:
    izpiši cikel dolžine 1, ki ga sestavlja le točka u; **return**;
  for $u := 1$ **to** n **do for** $v := 1$ **to** n **do** $d_{uv}^{(0)} = -\ln m[u][v]$;
  for $k := 1$ **to** n:
    for $u := 1$ **to** n **do for** $v := 1$ **to** n:
      $d_{uv}^{(k)} = \min\{d_{uv}^{(k-1)}, d_{uk}^{(k-1)} + d_{kv}^{(k-1)}\}$;
      if $u = v$ **and** $d_{uu}^{(k)} < 0$:
        izpiši cikel, ki ga sestavljata najkrajša pot od u do k in od k do u,
          pri čemer vmes uporabljamo le točke iz $\{1,\ldots,k-1\}$;
        return;
    matriko $d^{(k-1)}$ lahko zdaj pozabimo, saj je ne bomo več potrebovali;
  izpiši `"ni rešitve"`;

Tako smo poiskali nek profitni cikel v $O(n^3)$ časa in porabili $O(n^2)$ pomnilnika.

(b') Če se odpovemo zahtevi, da morajo biti valute v ciklu različne, postane problem lažji. Enako kot zgoraj pri (c) sestavimo graf, ki ima po eno točko za vsako valuto in po eno povezavo za vsak par valut, dolžine povezav pa naj bodo spet $d_{uv} = -\ln m[u][v]$. Naloga zdaj sprašuje, ali obstaja v grafu kakšen obhod (ne nujno cikel, ker točke na njem zdaj niso nujno različne) s k koraki in negativno dolžino.

Zdaj si lahko pomagamo z Bellman-Fordovim algoritmom. Naj bo $d_{uv}^{(k)}$ dolžina najkrajšega takega sprehoda od u do v, ki ga sestavlja k korakov. Pri $k = 1$ je tak sprehod seveda lahko le neposredna povezava od u do v, tako da imamo $d_{uv}^{(1)} = d_{uv}$. Za večje k pa lahko razmišljamo takole: recimo, da je zadnji korak na tem sprehodu $w \to v$; pred tem imamo torej sprehod v $k-1$ korakih od u do w. Ker iščemo najkrajši sprehod, imamo $d_{uv}^{(k)} = \min\{d_{uw}^{(k-1)} + d_{wv} : w \in 1..n\}$.

S podobnim razmislekom se lahko prepričamo, da velja to še splošneje: $d_{uv}^{(s+t)} = \min\{d_{uw}^{(s)} + d_{wv}^{(t)} : w \in 1..n\}$. Ker imamo n možnosti, kako izbrati w, nam izračun tega minimuma vzame $O(n)$ časa; in ker je tudi za u in v po n možnosti, nam izračun celotne matrike $d^{(s+t)}$ iz $d^{(s)}$ in $d^{(t)}$ vzame $O(n^3)$ časa. Ta izračun je zelo podoben množenju matrik v linearni algebri, le da se tam namesto maksimuma vsot računa vsoto produktov; zato bomo tudi tu rekli, da smo $d^{(s)}$ in $d^{(t)}$ „zmnožili“ v $d^{(s+t)}$. Ker je $O(n^3)$ časa za vsako množenje matrik precej drago, je koristno razmisliti o tem, kako izračunati želeni rezultat s čim manj takšnimi množenji.

Naloga sprašuje, če obstaja kakšen negativen obhod s k koraki, torej moramo izračunati matriko $d^{(k)}$ in pogledati, če je v njej kakšen od diagonalnih elementov negativen. Da čim hitreje pridemo do $d^{(k)}$, si lahko pomagamo z razpolavljanjem. Oglejmo si konkreten primer; recimo, da nas zanima $k = 29$. Ker je $29 = 14 + 15$, lahko izračunamo $d^{(29)} = d^{(14)} \cdot d^{15}$; če zdaj razpolovimo 14 in 15, vidimo, da bomo lahko računali $d^{(14)} = d^{(7)} \cdot d^{(7)}$ in $d^{(15)} = d^{(7)} \cdot d^{(8)}$; do 7 in 8 pridemo z $d^{(7)} = d^{(3)} \cdot d^{(4)}$ in $d^{(8)} = d^{(4)} \cdot d^{(4)}$; in tako naprej. V splošnem vidimo, da lahko $d^{(t)}$ in $d^{(t+1)}$ izračunamo z dvema množenjema iz $d^{(s)}$ in $d^{(s+1)}$, če je $s = \lfloor t/2 \rfloor$. Tako potrebujemo samo $O(\log k)$ množenj, da pridemo do $d^{(k)}$.

Tudi vmesne matrike si je koristno zapomniti, saj jih bomo potrebovali pri rekonstrukciji iskanega obhoda. Če iščemo najkrajši sprehod od u do v v k korakih in če vemo, da smo $d^{(k)}$ dobili z množenjem $d^{(s)} \cdot d^{(t)}$, potem moramo najprej poiskati tisti w, ki je dal najmanjšo vsoto $d_{uw}^{(s)} + d_{wv}^{(t)}$; nato poiščimo najkrajši sprehod od u do w v s korakih in najkrajši sprehod od w do v v t korakih; stik teh dveh sprehodov je potem sprehod od u do v v k korakih, ki smo ga iskali.

Tako smo prišli do rešitve, ki porabi $O(n^3 \log n)$ časa in $O(n^2 \log n)$ pomnilnika. Če nam je ta poraba pomnilnika prehuda, lahko nekatere matrike sproti pozabljamo in jih kasneje računamo ponovno; pridemo lahko do rešitve, ki porabi le $O(n^2)$ pomnilnika, vendar ima časovno zahtevnost $O(n^3(\log n)^2)$; obstaja tudi vmesna možnost, ki porabi $O(n^2 \log\log n)$ pomnilnika in $O(n^3 \log n \log\log n)$ časa.

19. AVL-drevo

Naj bo h_u višina poddrevesa s korenom v vozlišču u. Če ima u otroke $u_1, \ldots, u_k$, potem je $h_u = 1 + \max\{h_{u_1}, \ldots, h_{u_k}\}$; če pa u nima otrok, je $h_u = 1$. Vidimo torej, da je za izračun h_u koristno, če takrat že poznamo višine u-jevih otrok, zato bomo višine najlažje računali tako, da bomo drevo pregledovali od spodaj navzgor.

Koristno je, če si za vsako vozlišče pripravimo seznam njegovih otrok. Spodnja rešitev uporablja za to kar dve tabeli, prvi in nasl. Pri tem nam prvi[u] pove prvega otroka vozlišča u; nasl[u] pa nam pove naslednjega otroka u-jevega starša (torej otroka, ki je v seznamu otrok u-jevega starša takoj za u-jem samim). Konec seznama označimo z vrednostjo -1 v ustreznem elementu tabele. Obe tabeli lahko zgradimo z enim prehodom po seznamu povezav, ki smo ga dobili kot vhodni podatek.

Če koren ni eksplicitno podan kot vhodni podatek, ga lahko poiščemo tako, da pogledamo, katero vozlišče nima starša (to počne tudi spodnja rešitev in si v ta namen pomaga s tabelo stars). Ko enkrat poznamo koren, se lahko sistematično sprehodimo po celem drevesu od zgoraj navzdol: koren dodamo v vrsto, nato pa na vsakem koraku vzamemo eno vozlišče z začetka vrste in dodamo vse njegove otroke na konec vrste. Spodnji program hrani vrsto v tabeli vrsta, pri čemer indeks glava kaže na začetek vrste, rep pa na prvo prazno celico za koncem vrste. Ko se ta postopek ustavi, imamo v tabeli vrsta ravno seznam vseh vozlišč, urejen tako, da se vsako vozlišče v seznamu pojavi pred vsemi svojimi otroci. Če torej ta seznam pregledamo od konca proti začetku in pri vsakem vozlišču izračunamo višino, bomo takrat že imeli pri roki višine njegovih otrok, to pa je točno tisto, kar potrebujemo za učinkovit izračun višin. Za izračun višine vozlišča potrebujemo maksimum višin njegovih otrok, spotoma pa si ni težko zapomniti še minimuma višin njegovih otrok, s pomočjo katerega lahko sproti preverjamo, če kakšno vozlišče krši pogoj uravnoteženosti, po katerem sprašuje naloga.

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

#define n 7
int povezave[n − 1][2] = { /* primer iz besedila naloge */
  {1, 2}, {1, 3}, {3, 4}, {3, 5}, {4, 6}, {2, 7}};

bool Resitev()
{
  int prvi[n], nasl[n], stars[n], vrsta[n], visina[n], i, u, v, koren, hMin, hMax, glava, rep;

  /* Pripravimo sezname otrok; za vsako vozlišče si zapomnimo tudi starša. */
  for (u = 0; u < n; u++) { prvi[u] = −1; nasl[u] = −1; stars[u] = −1; }
  for (i = 0; i < n − 1; i++) {
    u = povezave[i][0] − 1; v = povezave[i][1] − 1;
    nasl[v] = prvi[u]; prvi[u] = v; stars[v] = u; }

  /* Poiščimo koren. */
  for (u = 0; u < n; u++) if (stars[u] < 0) koren = u;

  /* Naštejmo vozlišča drevesa od zgoraj navzdol. */
  glava = 0; rep = 0; vrsta[rep++] = koren;
  while (glava < rep) {
    u = vrsta[glava++];
    for (v = prvi[u]; v >= 0; v = nasl[v]) vrsta[rep++] = v; }

  /* Izračunajmo višine poddreves od spodaj navzgor in preverjajmo uravnoteženost. */
  while (rep−− > 0) {
    u = vrsta[rep]; hMin = n + 1; hMax = 0;
    for (v = prvi[u]; v >= 0; v = nasl[v]) {
      if (visina[v] < hMin) hMin = visina[v];
      if (visina[v] > hMax) hMax = visina[v]; }
    if (hMin < hMax − 1) return false;
```

```
    visina[u] = hMax + 1; }
  return true;
}
```

Razmislimo zdaj še o težji različici naloge, pri kateri drevo nima korena, ampak je le nek skupek vozlišč in (neusmerjenih) povezav med njimi. Podobno, kot smo prej pripravljali sezname otrok, lahko zdaj pripravimo sezname sosedov: ko naletimo na neusmerjeno povezavo $(u : v)$, dodamo u na seznam v-jevih sosedov in v na seznam u-jevih sosedov. Če si zdaj neko konkretno vozlišče r izberemo za koren drevesa, ni težko za vsako vozlišče sestaviti seznama otrok, s čimer dobi drevo podobno obliko kot v prvem delu naloge: ko za nek u pregledujemo seznam sosedov, vsi sosedje postanejo u-jevi otroci, razen tistega, ki je u-jev starš. Spodnji podprogram pričakuje, da mu poleg u-ja podamo še njegovega starša s. Otroke vozlišča u lahko obdelamo na enak način z rekurzivnimi klici.

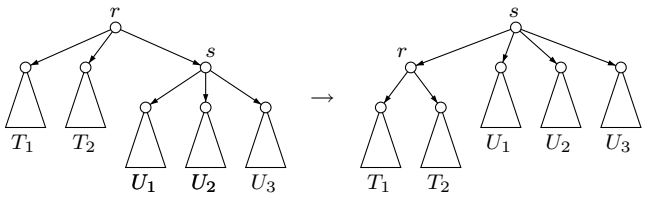
podprogram ObiščiPoddrevo(u, s):
  za vsakega u-jevega soseda v:
    if $v = s$ **then continue**;
    dodaj v na seznam u-jevih otrok;
    ObiščiPoddrevo(v, u);

Začetni klic bi bil ObiščiPoddrevo$(r, -1)$. Po koncu tega postopka imamo za vsako vozlišče pripravljen seznam otrok in lahko nadaljujemo enako kot v prvem delu naloge. Lahko bi šli še korak dlje in dopolnili ObiščiPoddrevo tako, da bi že kar sproti računal višine poddreves in preverjal uravnoteženost. V spodnji različici je ObiščiPoddrevo2 funkcija, ki vrne višino poddrevesa s korenom u (in pri čemer je s starš vozlišča u); če pa je to poddrevo ali kakšno od nižje ležečih pod-poddreves v njem neuravnoteženo, vrne -1.

podprogram ObiščiPoddrevo2(u, s):
  $h_{min} := \infty$; $h_{max} := 0$;
  za vsakega u-jevega soseda v:
    if $v = s$ **then continue**;
    $h :=$ ObiščiPoddrevo2(v, u);
    if $h < 0$ **then return** -1;
    $h_{min} := \min\{h_{min}, h\}$; $h_{max} := \min\{h_{max}, h\}$;
  if $h_{min} < h_{max} - 1$ **then return** -1 **else return** $h_{max} + 1$;

Lahko bi torej poklicali ObiščiPoddrevo2$(r, -1)$ in pogledali, ali je rezultat različen od -1; tedaj je drevo s korenom r uravnoteženo, sicer pa ne. To bi nam vzelo $O(n)$ časa. Ta postopek lahko zdaj ponovimo za vse možne r in preštejemo, pri koliko r-jih dobimo uravnoteženo drevo. Tako dobimo želeni rezultat v $O(n^2)$ časa.

Za učinkovitejšo rešitev pa se je bolje vrniti nazaj k prvotni različici podprograma ObiščiPoddrevo, ki gradi sezname otrok. Recimo, da smo neko vozlišče r izbrali za koren, izračunali višine vseh poddreves in preverili uravnoteženost. Recimo, da bi zdaj namesto r-ja izbrali za koren kakšnega od njegovih sosedov, recimo s. Dokler je bil koren v r-ju, je bil s kot r-jev sosed seveda njegov otrok; po novem pa s postane koren, r pa njegov otrok. Večina drevesa pa se sploh ne spremeni kaj posebej; označimo r-jeva poddrevesa (razen tistega, ki se začne pri s) s T_i in s-jeva poddrevesa z U_i. Drevo se pri premiku korena iz r v s spremeni takole:



Znotraj poddreves T_i in U_i se ne spremeni čisto nič; njihova višina in uravnoteženost sta taki kot prej. Vprašanje je torej le, kaj je z višino in uravnoteženostjo poddreves z začetkom pri r in pri s.

Za lažje razmišljanje vpeljimo še nekaj oznak. Če smo že obdelali prvotno različico drevesa (tisto, pri kateri je koren r), smo najbrž za vsako vozlišče u izračunali količine, kot so: $h[u]$ (višina poddrevesa s korenom u), $h_{min}[u]$ (minimum vrednosti $h[v]$ po vseh u-jevih otrocih v) in $h_{max}[u]$ (kot $h_{min}[u]$, le maksimum).

Kot smo že videli, v poddrevesih T_i in U_i ni prišlo do nikakršnih sprememb, zato se h, h_{min} in h_{max} pri vozliščih iz teh poddreves nič ne spremenijo. Pri r in s pa lahko pride do sprememb; pri $h_{min}[r]$ smo prej računali minimum višin po vseh T_i in še po s, po novem pa le še po T_i, kajti s ni več u-jev otrok; podobno je tudi s $h_{max}[r]$. Ko imamo novi $h_{max}[r]$, lahko izračunamo tudi novo $h[r]$, saj je to le $1 + h_{max}[r]$. Zdaj pa ni težko izračunati tudi novih $h_{min}[s]$ in $h_{max}[s]$. Za tidve količini smo pred spremembo korena gledali minimum oz. maksimum po vseh U_i, po novem pa ju moramo gledati po vseh U_i in še po r, saj je slednji na novo postal s-jev otrok. Treba je torej le pogledati, če je (novi) $h[r]$ manjši od dosedanjega $h_{min}[s]$ ali večji od dosedanjega $h_{max}[s]$ (in ju po potrebi ustrezno popraviti).

Pri izračunu nove vrednosti $h_{min}[r]$ in $h_{max}[r]$ moramo biti pazljivi; ne moremo si privoščiti, da bi se sprehodili po vseh r-jevih otrocih, saj je teh lahko $O(n)$ in bi to na dolgi rok povzročilo, da bi imel naš postopek na koncu spet časovno zahtevnost $O(n^2)$. Koristno je, če za vsako vozlišče drevesa vzdržujemo neko primerno podatkovno strukturo, v kateri so njegovi otroci urejeni po $h[u]$; ta struktura mora podpirati dodajanje in brisanje otroka ter izračun minimuma in maksimuma. Primerne strukture so na primer rdeče-črno drevo, AVL-drevo ali B-drevo; z njimi lahko vsako od naštetih operacij izvedemo v $O(\log n)$ časa. Vse ostale operacije, ki smo jih morali izvesti pri premiku korena iz r v s, vzamejo le $O(1)$ časa, tako da je cena enega premika korena $O(\log n)$.

Če v neki spremenljivki vzdržujemo trenutno število uravnoteženih vozlišč, tega ni težko popraviti, ko premaknemo koren. Pred premikom korena zmanjšamo števec uravnoteženih vozlišč za 1, če je bil r uravnotežen (in podobno za s); po premiku korena pa števec povečamo za 1, če je r zdaj uravnotežen (in podobno za s). Če ima po tej spremembi števec vrednost n, potem vemo, da je drevo v trenutni obliki uravnoteženo.

Doslej smo videli, kako lahko koren premaknemo iz dosedanjega korena v enega od njegovih dosedanjih otrok. S to osnovno operacijo moramo zdaj drevo počasi predelovati tako, da se bo prav vsako od njegovih n vozlišč vsaj enkrat znašlo v korenu. Primeren vrstni red vozlišč lahko sestavimo takole:

podprogram ObiščiPoddrevo3(u, s, L):
  za vsakega u-jevega soseda v:

if $v = s$ **then continue**;
dodaj v na konec seznama L;
ObiščiPoddrevo3(v, u, L);
dodaj u na konec seznama L;

Postopek poženemo tako, da si izberemo poljuben začetni koren r in prazen seznam L ter pokličemo ObiščiPoddrevo3$(r, -1, L)$. Ob vrnitvi iz tega klica bomo imeli v L seznam, v katerem se vsako vozlišče pojavi vsaj enkrat (natančneje povedano, vsako vozlišče se v njem pojavi tolikokrat, kolikor sosedov ima; skupna dolžina tega seznama pa je zato $2(n-1)$, kajti vsaka povezava v drevesu — teh pa je $n-1$ — prispeva v seznam dva elementa) in vsako vozlišče v seznamu je sosed prejšnjega vozlišča v seznamu. Zato lahko začnemo z drevesom, ki ima koren v r, nato pa po vrsti pregledujemo vozlišča iz seznama L in v vsakem koraku premaknemo koren v naslednje vozlišče seznama (ki je torej zagotovo sosed trenutnega korena). Po vsakem premiku korena popravimo tabele h, h_{min}, h_{max} in število uravnoteženih vozlišč; v neki novi tabeli si zapomnimo, kateri koreni so dali uravnoteženo drevo, in na koncu s pomočjo te tabele preštejemo, koliko je takih korenov. Ker smo potrebovali $O(n)$ premikov korena in ker je z vsakim premikom korena $O(\log n)$ dela, je časovna zahtevnost celotne rešitve $O(n \log n)$.

20. Birokrati

V nalogi se skriva problem najkrajše poti po grafu; za vsakega birokrata imejmo po eno točko, usmerjeno povezavo $u \to v$ pa imejmo tam, kjer lahko u odloži papir na v-jevo mizo (torej če je $u < v \leq u + n_u$). Naloga zdaj pravzaprav sprašuje po najkrajši poti od točke 1 do točke n. Pri tem štejejo vse povezave za enako dolge, torej je dolžina poti le število povezav na njej. Naj bo $d[u]$ dolžina najkrajše poti od 1 do u; ker kažejo vse povezave od manjših številk proti večjim, lahko tudi najkrajše poti $d[u]$ računamo sistematično od manjših u proti večjim. Če poznamo dolžino najkrajše poti od 1 do u (namreč $d[u]$ korakov) in če obstaja povezava $u \to v$, potem tudi vemo, da bi se dalo od 1 do v priti v $d[u]+1$ korakih; če je to manj od najkrajše doslej znane poti od 1 do v, si novo dolžino zapomnimo v $d[v]$.

for $u := 1$ **to** n: $d[u] := \infty$;
$d[1] := 0$;
for $u := 1$ **to** n:
  for $v := u + 1$ **to** $u + n_u$:
    (* *Od 1 do u se dá priti v* $d[u]$ *korakih; do v torej v* $d[u] + 1$ *korakih.* *)
    if $d[v] > d[u] + 1$ **then** $d[v] := d[u] + 1$;

Na koncu je v $d[n]$ rezultat, po katerem sprašuje naloga. Neugodno pri tem postopku je, da je n lahko velik, pa tudi posamezni n_u so lahko veliki in v najslabšem primeru bi lahko postopek porabil $O(n^2)$ časa.

Rešitev lahko izboljšamo, če opazimo naslednjo zakonitost: če je $u < v$, potem je najkrajša pot od 1 do v vsaj tako dolga kot najkrajša pot od 1 do u. Recimo namreč, da to ne bi veljalo; vzemimo poljuben tak par točk, za kateri je $u < v$ in je najkrajša pot od 1 do u daljša kot najkrajša pot od 1 do v. Oglejmo si najkrajšo pot od 1 do v; to je recimo $v_0 \to v_1 \to v_2 \to \ldots \to v_{k-1} \to v_k$, pri čemer je $v_0 = 1$, $v_k = k$ in dolžina poti je k korakov. Iz tega tudi sledi, da je najkrajša pot od 1 do

v_i dolga i korakov za vse i od 1 do k. Ker vse povezave kažejo od manjših številk proti večjim, obstoj te poti pomeni tudi, da je $1 = v_0 < v_1 < \ldots < v_k = v$. Ker velja $1 \leq u < v$, kar je seveda isto kot $v_0 \leq u < v_k$, mora biti u na enem od podintervalov oblike $v_{i-1} \leq u < v_i$ za nek $i \in \{1, \ldots, k\}$. Če je $u = v_{i-1}$, pomeni, da je najkrajša pot od 1 do u dolga le i korakov (kar je manj kot k); če pa je $v_{i-1} < u < v_i$, potem, ker lahko birokrat v_{i-1} preda papir birokratu v_i, ga lahko preda tudi birokratu u; in ker je najkrajša pot od 1 do v_i dolga i korakov, to pomeni, da je najkrajša pot od 1 do u dolga kvečjemu $i + 1$ korakov, kar je $\leq k$. V vsakem primeru torej vidimo, da je najkrajša pot od 1 do u dolga največ k korakov (to pa je ravno dolžina najkrajše poti od 1 do v); tako smo prišli v protislovje s predpostavko, da je najkrajša pot od 1 do u daljša kot najkrajša pot od 1 do v.

Recimo zdaj, da je naš gornji algoritem v nekem trenutku prvič našel neko pot do v; do nje je prišel iz neke u in v celico $d[v]$ je pravkar vpisal vrednost $d[u] + 1$. Ali je mogoče, da bi naš algoritem nekoč kasneje vrednost $d[v]$ še kaj spreminjal? To bi pomenilo, da bi pri nekem kasnejšem u — recimo mu u' — našel neko še krajšo pot do v. Tista nova pot bi bila seveda dolga $d[u'] + 1$; toda ker je $u' > u$, je gotovo $d[u'] \geq d[u]$ (o tem smo se prepričali v prejšnjem odstavku), zato je tudi $d[u'] + 1 \geq d[u] + 1$. Torej ni mogoče, da bi pri nekem kasnejšem u' našli do v-ja kakšno krajšo pot od tiste prve.

Tako torej vidimo, da nam v notranji zanki sploh ni treba pregledovati vseh v-jev od $u + 1$ naprej; dovolj je, če začnemo pri prvem takem v, do katerega sploh še ne poznamo nobene poti.

```
for u := 1 to n: d[u] := ∞;
v := 1; d[1] := 0;
for u := 1 to n:
  (* Na tem mestu poznamo najkrajše poti od 1 do prvih v točk. *)
  while v < u + n_u:
    v := v + 1;
    (* Od 1 do u se dá priti v d[u] korakih; do v torej v d[u] + 1 korakih. *)
    if d[v] < d[u] + 1 then d[v] := d[u] + 1;
```

Zdaj sta torej obe zanki prepleteni: ko u narašča od 1 do n, tudi v počasi narašča do 1 do (največ) n, tako da je časovna zahtevnost celotnega postopka le še $O(n)$.

21. Konjska vprega

Naloga je zelo podobna problemu najkrajše poti po grafih (v našem primeru točke grafa predstavljajo vasi, povezave pa ceste med njimi). Pri problemu najkrajše poti ima vsaka povezava $(u : v)$ dolžino d_{uv}, dolžina poti $u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ pa je vsota dolžin povezav na njej: $d_{u_0u_1} + d_{u_1u_2} + \ldots + d_{u_{k-1}u_k}$. Pri naši nalogi povezave (torej posamezne ceste) nimajo dolžine, imajo pa podatek o minimalnem potrebnem številu konj. Najmanjše primerno število konj, s katerim lahko opravimo pot $u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$, pa je maksimum teh omejitev po vseh povezavah na poti: $\max\{d_{u_0u_1}, d_{u_1u_2}, \ldots, d_{u_{k-1}u_k}\}$. Stvar je torej zelo podobna iskanju najkrajših poti, le da namesto vsote dolžin povezav tukaj računamo maksimum.

Podobno kot pri iskanju najkrajših poti tudi tukaj velja, da najboljšo pot od s do v (recimo tej poti ρ_v) dobimo tako, da vzamemo najboljšo pot od s do neke druge

(primerno izbrane) točke u (recimo tej poti ρ_u) in jo podaljšamo s povezavo od u do v. Pri najkrajših poteh velja, da je dolžina poti ρ_v potem vsota dolžine poti ρ_u in dolžine nove povezave d_{uv}, pri naši nalogi pa bomo tu namesto vsote morali vzeti maksimum. Zato lahko uporabimo kar kakšnega od znanih algoritmov za iskanje najkrajših poti v grafu, le namesto seštevanja dolžin računamo maksimume. Ker imajo pri naši nalogi povezave različno „dolžino“ (v resnici potrebno število konj), je koristno uporabiti Dijkstrov algoritem:

```
za vsako točko v: d[v] := ∞; p[v] := NIL;
H := {s}; d[s] := 0;
while H ni prazna:
  naj bo u tista točka iz H, ki ima najmanjšo d[u];        (†)
  pobriši u iz H;
  za vsako u-jevo sosedo v:
    c := max{d[u], d_uv};
    if c < d[v]:
      d[v] := c; p[v] := u; če v še ni v H, jo dodaj vanjo;
```

Na koncu tega postopka imamo za vsako točko v v tabeli $d[v]$ podatek o najmanjšem številu konj, ki jih potrebujemo za pot od s do v (če je $d[v] = \infty$, pomeni, da od s do v sploh ne moremo priti), $p[v]$ pa je neposredna predhodnica v-ja na najboljši poti od s do v. Če nas zanima pot od s do t, jo lahko torej odčitamo od konca proti začetku: t, $p[t]$, $p[p[t]]$ in tako nazaj. Da bo postopek učinkovit, je koristno množico H predstaviti z dvojiško kopico; časovna zahtevnost celotnega postopka je potem $O((n+m)\log n)$.[12]

Naloge pa se lahko lotimo tudi malo drugače. Predstavljajmo si najboljšo pot od s do t (torej tisto, ki zahteva najmanj konj); potrebno število konj pri tej poti je enako potrebnemu številu poti po eni od povezav na njej (namreč po tisti povezavi, ki zahteva največ konj). Tako torej vemo, da je minimalno potrebno število konj za pot od s do t gotovo enako vrednosti d_{uv} za eno od povezav $(u : v)$, čeprav vnaprej še ne vemo, za katero povezavo. Možnih vrednosti d_{uv} pa je največ toliko, kolikor je povezav, torej m. Uredimo jih naraščajoče in jih v tem vrstnem redu označimo z $d_1, \ldots, d_m$. V mislih dodajmo še $d_0 = d_1 - 1$ in $d_{m+1} = \infty$. Zdaj si lahko pomagamo z bisekcijo:

```
l := 0; r := m + 1;
while r − l > 1:
  (* Invarianta: od s do t je mogoče priti z d_r konji, ne pa z d_l konji. *)
  m := ⌊(l + r)/2⌋;
  if je mogoče od s do t priti z d_m konji then r := m else l := m;
```

[12] O pravilnosti postopka se lahko prepričamo s pomočjo invariante, ki velja pred vrstico (†). Vse točke lahko na tistem mestu razdelimo v tri skupine: tiste, ki so v H; tiste, ki niso v H, so pa nekoč že bile v njej (recimo jim B); in tiste, ki še nikoli niso bile v H (recimo jim D; prepoznamo jih po tem, da imajo $d[v] = \infty$). Pred vrstico (†) zdaj velja naslednje: za vsako $u \in B$ že poznamo najboljšo pot od s do u; v H so vse take točke u, ki niso v B, imajo pa kakšno sosedo v B; in za vsako točko iz H poznamo najboljšo tako pot od s do u, ki se giblje ves čas znotraj množice B, le zadnji korak naredi v u; v D pa so vse tiste točke, ki niso niti v B niti niso sosede točk iz B. O tem, da ta invarianta res velja na začetku vsake iteracije in da iz nje sledi pravilnost postopka, se lahko prepriča bralec sam.

Naš postopek torej vzdržuje interval možnih števil konjev, pri čemer ves čas vemo, da je d_l konj že premalo, d_r pa jih je gotovo dovolj. Na vsakem koraku interval zožimo približno na polovico, tako da potrebujemo $O(\log m)$ iteracij zunanje zanke (to pa je isto kot $O(\log n)$, saj je $m = O(n^2)$). Na koncu tega postopka imamo $l = r - 1$, tako da vemo, da je d_r ravno najmanjše število konj, s katerim je še mogoče priti od s do t. Če je $d_r = \infty$, to pomeni, da poti od s do t sploh ni.

Za preverjanje, ali je mogoče od s do t priti z d_m konji, lahko uporabimo katerega koli od postopkov za sistematično pregledovanje grafa, na primer iskanje v širino ali iskanje v globino; paziti moramo le na to, da se pri tem za povezave $(u : v)$, ki zahtevajo več kot d_m konj, pretvarjamo, kot da jih v grafu sploh ni. Časovna zahtevnost takega pregleda grafa je $O(n+m)$, tako da je časovna zahtevnost celotne rešitve zdaj $O((n+m)\log n)$. Tako smo približno na istem kot zgoraj pri Dijkstrovem algoritmu, pa še z implementacijo kopice se nam ni bilo treba ukvarjati. Ko enkrat poznamo minimalno potrebno število konj, lahko še enkrat poženemo iskanje v širino (s tem številom konj), da poiščemo neko konkretno najboljšo pot od s do t.

22. Mutacije

Naj bo s naš začetni niz, t naš končni niz, $w_k \to x_k$ pa naj bo k-ta mutacija. Naloga pravi, da moramo mutacije izvajati od leve proti desni; recimo, da najprej izvedemo mutacijo $w_k \to x_k$. Niz s je torej moral biti oblike $s'w_k s''$, pri čemer s' predstavlja del niza s levo od tiste pojavitve podniza w_k, na kateri smo izvedli prvo mutacijo, s'' pa del niza s desno od nje. Z mutacijo $w_k \to x_k$ dobimo zdaj niz $s'x_k s''$, omejitev iz besedila naloge pa nam tudi pove, da bomo morali vse nadaljnje mutacije izvajati v celoti znotraj niza s'', ne pa več v $s'x_k$. Ta začetni del niza se torej ne bo več spreminjal in naš niz se bo odtlej vedno začenjal na $s'x_k$; opisano mutacijo si torej lahko privoščimo le, če se tudi končni niz t (ki bi ga radi dobili) začne na $s'x_k$. V tem primeru je t oblike $t = s'x_k t''$ za nek t'' in naloga vseh nadaljnjih mutacij (vseh razen prve) je torej ta, da s'' predelajo v t''. Tako smo prvotni problem (kako predelati s v t) prevedli na enak problem, le s krajšima nizoma (kako predelati s'' v t''), ki sta za povrhu še celo končnici (sufiksa) prvotnih nizov s in t.

Recimo, da sta vhodna niza dolga po n oz. m znakov in predstavljena s tabelama $s[1..n]$ in $t[1..m]$. Naj bo zdaj $f(i,j)$ najmanjše število mutacij, potrebnih za predelavo $s[i..n]$ v $t[j..m]$. Videli smo, da ga lahko računamo takole:

funkcija $f(i,j)$:
  (* *Prazen niz lahko predelamo v prazen niz, ne pa v nepraznega.* *)
  if $i > n$ **then**
    if $j > m$ **then return** 0 **else return** ∞;
  (* *Če se trenutni črki* $s[i]$ *in* $t[j]$ *ujemata, si lahko privoščimo, da tu ne naredimo mutacije.* *)
  if $j \le m$ **and** $s[i] = t[j]$ **then** $r := f(i+1, j+1)$ **else** $r := \infty$;
  (* Katere mutacije lahko izvedemo na tem mestu? *)
  for $k := 1$ **to** *SteviloMutacij*:
    if se $s[i..n]$ začne na w_k in se $t[j..m]$ začne na x_k (†)
    then $r := \min\{r, 1 + f(i + |w_k|, j + |x_k|)\}$;
  return r;

Rezultat, po katerem sprašuje naloga, je potem $f(1,1)$. Da bi postal algoritem učinkovit, bi morali paziti še na to, da ne računamo rezultata $f(i,j)$ po večkrat za en in isti par (i,j), kar bi se sicer v zgornji različici algoritma lahko zgodilo. Zato bi morali že izračunane vrednosti $f(i,j)$ sproti shranjevati v neko tabelo, kjer nam bodo pri roki, če jih bomo kasneje še kdaj potrebovali. Lahko pa bi se rekurzivnim klicem sploh odpovedali in računali funkcijo f čisto sistematično, po padajočih i in pri vsakem i po padajočih j; tako bomo vedno imeli že izračunane rešitve vseh podproblemov, ki jih potrebujemo za reševanje trenutnega podproblema.

Koristno bi bilo tudi na začetku označiti vse pojavitve nizov w_k v s in nizov x_k v t, da nam kasneje v vrstici (†) ne bo treba tega preverjati znova in znova (pri različnih i in j). V ta namen lahko uporabimo kakšnega od znanih algoritmov za iskanje več podnizov v nizu (npr. Rabin-Karp, Aho-Corasick, Wu-Manber; glej npr. rešitev naloge 2007.3.3, str. 67 v biltenu 2007).

Mimogrede, omejitev iz besedila naloge, da se mutacije izvajajo od leve proti desni in da mutacija ne more delovati nad delom niza, ki je nastal kot rezultat kakšne prejšnje mutacije, je zelo pomembna; brez te omejitve postane naša naloga eden od klasičnih neodločljivih problemov, torej takih, za katere dokazano ne obstaja noben algoritem, ki bi se vedno ustavil po končno mnogo korakih in izračunal pravilni odgovor.

23. Potapljanje ladjic

Nalogo lahko rešujemo z rekurzijo. Vemo, da ima nasprotnik $n = 5$ ladij, pri čemer ima k-ta ladja dolžino d_k. Najprej torej poskusimo na razne načine postaviti v mrežo prvo ladjo (dolžine d_1), seveda le tako, da bo njen položaj konsistenten z danim zaporedjem potez in izidov. Nato poskusimo na razne načine postaviti drugo ladjo in tako naprej. Če pridemo v položaj, ko naslednje ladje ne moremo postaviti nikamor, se vrnemo na prejšnjo ladjo in poskusimo njo postaviti na naslednji možni položaj.

podprogram PostaviLadje(k):
  za vsako potezo (x_i, y_i), ki ima izid „zadetek“ ali „potopljena“:
    if to polje pripada kakšni od že postavljenih ladij **then continue**;
    ladja dolžine d_k lahko pokrije polje (x_i, y_i) le tedaj, če njen
      zgornji levi konec leži na $(x_i - t, y_i)$ ali $(x_i, y_i - t)$ za $t \in \{0, \ldots, d_k - 1\}$;
    za vsakega od teh položajev:
      if bi se ta ladja takrat prekrivala s kakšno od že postavljenih
        ali pa bi pokrila kakšno polje, za katero imamo potezo z izidom „zgrešil“
        ali pa bi segala čez rob igralne mreže **then continue**;
      if bi ladja pokrila kakšno polje, ki je v seznamu potez z izidom
        „potopljena“ in se pred njim v seznamu ne pojavljajo poteze z izidom
        „zadetek“ za vsa ostala polja te ladje **then continue**;
      postavi ladjo k na ta položaj;
      if $k < n$ **then** PostaviLadje($k + 1$) **else** KonecRekurzije();
      pobriši ladjo k s tega položaja;
  (* *Ostane še možnost, da ladja k sploh nikoli ni bila zadeta.* *)
  if $k < n$ **then** PostaviLadje($k + 1$) **else** KonecRekurzije();

Ta rekurzivni postopek torej poskuša na mrežo postaviti ladje (ne nujno vseh) tako, da vsaka od njih pokrije vsaj eno zadeto polje (torej tista, za katera je bila izvedena poteza z izidom „zadetek“ ali „potopljena“) in da se pri tem ne prekrivajo, ne gledajo čez rob mreže in ne pokrivajo polj, za katera vemo, da morajo biti prazna.

Ko najdemo tak razpored, pokličemo podprogram KonecRekurzije, ki mora dokončno preveriti, če je razpored ustrezen. To predvsem pomeni, da morajo naše ladje pokriti *vsa* zadeta polja, ne le nekaterih; poleg tega pa moramo preveriti še, ali bi se dalo tiste ladje, ki jih doslej nismo razporedili, postaviti na mrežo tako, da v celoti ležijo na poljih, ki jih ni odkrila nobena od vhodnih potez, in da se pri tem seveda ne prekrivajo in ne gledajo čez rob mreže. To bi morali spet narediti z rekurzijo.

Zakaj smo si to drugo rekurzijo prihranili na konec, namesto da bi o konkretnih položajih takih ladij razmišljal že postopek PostaviLadje? Predvsem zato, ker se lahko zgodi, da je tu primernih polj (torej takih, za katera sploh ni bila izvedena nobena poteza) bistveno več kot zadetih polj, zato je možnih položajev vsake ladje tu lahko zelo veliko. Če bi o teh položajih razmišljal že postopek PostaviLadje, bi dobil zelo veliko takih razporedov, pri katerih večina ladij sploh ne pokrije nobenega zadetega polja; za te razporede bi potem seveda KonecRekurzije ugotovil, da so neveljavni, vendar bi dotlej za delo z njimi že porabili veliko časa. Zato je bolje, da si razporejanje ladij, ki nikoli niso bile zadete, prihranimo za konec, po tistem, ko že vemo, da smo uspešno pokrili vsa zadeta polja.

Kot je običajno pri rekurzivnih postopkih, je koristno razmisliti o tem, kako čim prej prepoznati neobetavne razporede, da ne bomo pri njih po nepotrebnem nadaljevali z rekurzijo in razporejali še preostalih ladij. En pogoj, ki bi ga lahko dodali v gornjo rešitev, je na primer naslednji: če smo že razporedili k ladij in je ostalo še več kot $n - k$ takih polj, za katere imamo potezo tipa „potopljena“ in ki ne pripadajo nobeni od dosedanjih k ladij, potem vemo, da je dosedanji razpored brezupen. Podobno je razpored brezupen, če smo že razporedili k ladij in je ostalo več kot $\sum_{k'=k+1}^{n} d_{k'}$ takih zadetih polj, ki ne pripadajo nobeni od dosedanjih k ladij (torej več polj, kot pa jih lahko pokrijejo vse preostale ladje skupaj).

24. Bubble sort

Naj bo $b_1, \ldots, b_k$ zaporedje vrstic iz naše datoteke. Če zdaj poženemo postopek za urejanje iz besedila naloge na zaporedju b_1, nas pravzaprav zanima, ali ta postopek nekoč med izvajanjem izpiše b_2 in nekoč kasneje za tem izpiše tudi b_3 in nekoč še kasneje izpiše tudi b_4 in tako naprej. To lahko torej preverimo tako, da med izvajanjem postopka vzdržujemo podatek o tem, koliko prvih vrstic naše vhodne datoteke je postopek že izpisal. Recimo, da je že izpisal vrstice $b_1, \ldots, b_t$, ne pa še vrstic $b_{t+1}, \ldots, b_k$. Če je naslednja vrstica, ki jo postopek izpiše, enaka b_{t+1}, moramo povečati števec t za 1. Na koncu moramo le še preveriti, če je t dosegel vrednost k ali ne (če je ni, pomeni, da naša vhodna datoteka ni mogla nastati na način, ki ga opisuje besedilo naloge).

Algoritem 1

```
1   a := b_1; t := 1;
2   for i := 1 to n − 1:
3       for j := 1 to n − i:
```

```
4         if a[j] > a[j + 1]:
5           zamenjaj ta dva elementa v tabeli a;
6           if t < k and a = b_{t+1}:
7             t := t + 1;
8   return (t == k);
```

Pravzaprav bi bilo postopek koristno prekiniti, čim t doseže vrednost k, saj v tem primeru že vemo, da bo odgovor na vprašanje iz naloge pritrdilen in nam urejanja ni treba izvesti do konca.

Kakšna je časovna zahtevnost tega postopka? Do zamenjave lahko pride največ $O(n^2)$-krat (npr. če je začetno zaporedje b_1 urejeno padajoče) in lahko se zgodi, da moramo po vsaki zamenjavi izvesti vrstico 6, ki primerja tabeli a in b_{t+1}. To sta torej dve tabeli s po n elementi in pojavi se vprašanje, kako ju primerjati. Najpreprostejša rešitev je, da bi ju primerjali kar z zanko, ki gre od 1 do n in primerja istoležne elemente obeh tabel. Tako bi primerjanje tabel a in b_{t+1} vzelo $O(n)$ časa in ker potrebujemo $O(n^2)$ takšnih primerjanj, bi časovna zahtevnost celotnega postopka znašala kar $O(n^3)$. (Načeloma moramo k temu prišteti še $O(nk)$ časa za branje vhodnih podatkov, torej zaporedja $b_1, \ldots, b_k$. Toda pri urejanju tabele n števil z bubble sortom se izvede največ $n(n-1)/2$ zamenjav; torej če je $k > n(n-1)/2 + 1$, potem vemo, da tako dolga vhodna datoteka gotovo ni mogla nastati pri urejanju tabele n elementov, torej se nam s takšnimi primeri sploh ni treba ukvarjati.)

Primerjanje a in b_{t+1} lahko izvedemo tudi bolj učinkovito. Recimo, da smo a in b_{t+1} že primerjali z zanko, ki gre po indeksih od 1 do n in primerja istoležne elemente obeh tabel; in recimo, da je ta zanka ugotovila, da se tabeli razlikujeta. Naj bo d število indeksov, pri kateh se istoležna elementa obeh tabel razlikujeta (torej število takih r, za katere je $a[r] \neq b_{t+1}[r]$). V nadaljevanju si lahko pomagamo z dejstvom, da naš postopek za urejanje spreminja tabelo a zelo počasi: do sprememb pride le v vrstici 5, pa še takrat le tako, da se zamenjata elementa $a[j]$ in $a[j+1]$. Ker se torej večina elementov tabele a ne spremeni, se tudi od tabele b_{t+1} še vedno razlikuje na približno istih mestih kot prej; do sprememb pri tem lahko pride le na indeksih j in $j+1$. Število neujemanj d lahko torej popravimo zelo poceni: odšteti moramo morebitni neujemanji na mestih j in $j+1$ pri starem stanju tabele (pred zamenjavo) in nato prišteti morebitni neujemanji na teh mestih pri novem stanju tabele (po zamenjavi). Če števec neujemanj d pade na 0, pa vemo, da je tabela a zdaj popolnoma enaka tabeli b_{t+1}. V tem primeru moramo povečati t za 1, pa tudi na novo izračunati d (ki mora zdaj šteti neujemanja med a in novo b_{t+1}, ki je bivša b_{t+2}). Dopolnjeni postopek je zdaj takšen:

Algoritem 2

```
1   a := b_1; t := 1;
1'  d := 0; for r := 1 to n do if a[r] ≠ b_{t+1}[r] then d := d + 1;
2   for i := 1 to n − 1:
3     for j := 1 to n − i:
4       if a[j] > a[j + 1]:
4'        for r := j − 1 to j do if a[r] ≠ b_{t+1}[r] then d := d − 1;
5         zamenjaj elementa a[j] in a[j + 1];
5'        for r := j − 1 to j do if a[r] ≠ b_{t+1}[r] then d := d + 1;
6         if t < k and d = 0:
```

7 $t := t + 1$;
7′ $d := 0$; **for** $r := 1$ **to** n **do if** $a[r] \neq b_{t+1}[r]$ **then** $d := d + 1$;
8 **return** $(t == k)$;

Pri vrstici 1′ bi morali pravzaprav paziti še na primer, ko je $k = 1$; takrat bi bil b_{t+1} v tej vrstici nedefiniran in bi bilo najbolje, če bi naš postopek kar takoj vrnil **true**.

Ker nam zdaj pri posamezni zamenjavi vse vrstice od 4′ do 7 vzamejo le $O(1)$ časa, je njihov skupni prispevek k časovni zahtevnosti postopka le $O(n^2)$. Vrstica 7′ pa sicer vzame vsakič $O(n)$ časa, vendar se izvede le po vsakem povečanju t-ja, torej največ $O(k)$-krat. Časovna zahtevnost celotnega postopka je torej $O(n^2 + nk)$. Če je k majhen v primerjavi z n^2, je ta postopek precej učinkovitejši od prejšnjega.

Oglejmo si še malo drugačen pristop k reševanju naloge. Izkaže se, da lahko za posamezno vrstico vhodne datoteke — recimo b_t — v $O(n \log n)$ časa ugotovimo, ali je nastala med izvajanjem postopka za urejanje iz besedila naloge (in kakšni sta bili takrat vrednosti števcev i in j). Če to preverjanje uspe pri vsaki od k vrstic vhodne datoteke in če so dobljeni pari (i, j) naraščajoči, potem je vhodna datoteka res lahko nastala po postopku iz besedila naloge, sicer pa ne. Tako smo nalogo rešili v času $O(nk \log n)$, kar je učinkoviteje od dosedanjih postopkov, če je k majhen v primerjavi z n.

Kako lahko torej za posamezno vrstico b_t ugotovimo, ali je (in kdaj je) nastala med izvajanjem našega postopka za urejanje? Naloga pravi, da postopek začne z neko permutacijo števil od 1 do n. Torej mora biti tudi b_t neka permutacija števil od 1 do n; to vsekakor lahko preverimo v $O(n)$ časa. Če imamo zdaj neko permutacijo $a[1..n]$, lahko definiramo *tabelo inverzij* $q[1..n]$ te permutacije takole: $q[x]$ (da bo manj oklepajev, ga bomo v bodoče zapisovali kot q_x) naj bo število elementov, ki so večji od x in ki v tabeli a ležijo levo od elementa z vrednostjo x. Tabelo inverzij lahko izračunamo v $O(n \log n)$ časa takole:

naj bo T prazno drevo;
for $i := 1$ **to** n:
  $x := a[i]$; $q[x] :=$ število elementov T, ki so večji od x;
  dodaj x v T;

Za T lahko uporabimo na primer rdeče-črno drevo, AVL-drevo, Fenwickovo drevo ali pa $\log_2 n$ tabel, od katerih je vsaka pol manjša od prejšnje. Oglejmo si še obraten postopek, ki iz tabele inverzij q rekonstruira permutacijo a:

naj bo L prazen seznam;
for $x := n$ **downto** 1:
  vrini x v seznam L tako, da bo zdaj levo od njega q_x elementov;

Na koncu tega postopka vsebuje seznam L ravno permutacijo a. Če hočemo, da bo postopek učinkovit, je pametno L namesto v seznam organizirati v dvojiško drevo, v katerem ob vsakem vozlišču (elementu seznama) piše tudi, koliko elementov je v celotnem poddrevesu s korenom v tem vozlišču. To nam bo omogočilo v $O(\log n)$ časa določiti, kam v drevo je treba vriniti novi element x. Paziti moramo tudi na to, da drevo ohranjamo primerno uravnoteženo (lahko z enakimi postopki kot pri rdeče-črnem ali AVL-drevesu); celoten postopek rekonstrukcije a iz q traja potem le $O(n \log n)$ časa.

Recimo zdaj, da si zapomnimo stanje tabele a na začetku ene od iteracij glavne zanke našega postopka iz besedila naloge; nato izvedimo to iteracijo glavne zanke (torej pri trenutnem i izvedimo celotno zanko po j); novo stanje tabele označimo z a'. Iz obeh tabel izračunajmo še tabeli inverzij: q iz a in q' iz a'. Če primerjamo tabeli q in q', se izkaže, da za vsak x velja zveza $q'_x = \max\{0, q_x - 1\}$ (dokaz prepuščamo bralcu za vajo). Med izvajanjem notranje zanke (po j) se tabela inverzij q počasi spreminja v q': ko zamenjamo elementa $a[j]$ in $a[j+1]$, se q_x zmanjša za 1, če z x označimo element, ki je bil pred zamenjavo v celici $a[j+1]$ (po zamenjavi pa v $a[j]$).

Če namesto ene iteracije glavne zanke (po i) izvedemo m iteracij, bi veljala zveza $q'_x = \max\{0, q_x - m\}$. Če nato izvedemo še del $(m+1)$-ve iteracije (torej pri njej izvedemo notranjo zanko po j deloma, ne pa nujno v celoti), pa bi med končnim stanjem tabele inverzij q' in začetno tabelo inverzij q veljala naslednja zveza: za vsak x je q'_x enak bodisi $\max\{0, q_x - m\}$ bodisi $\max\{0, q_x - (m+1)\}$.

Če imamo torej zdaj pred sabo neko vrstico vhodne datoteke, recimo b_t, lahko razmišljamo takole. Izračunajmo iz nje tabelo inverzij q'; če so v njej vsi elementi enaki 0, to pomeni, da je b_t že popolnoma urejena naraščajoče. Takšno zaporedje bi postopek iz besedila naloge res izpisal, vendar šele čisto na koncu; če torej do tega pride pri $t < k$, potem vemo, da vhodna datoteka ni mogla nastati na način iz besedila naloge. Če pa v q' niso vsi elementi enaki 0, lahko q' zdaj primerjamo s tabelo inverzij q, kakršna je veljala na začetku urejanja (torej taka, kot jo izračunamo iz b_1). Preveriti moramo, ali obstaja tak m, da za vsak x velja $q'_x = \max\{0, q_x - m\}$ ali pa $q'_x = \max\{0, q_x - (m+1)\}$. Pri vsakem x dobimo interval možnih m-jev, s katerimi bi bil ta pogoj izpolnjen; če je $q'_x > 0$, sta možni vrednosti $m = q_x - q'_x$ in $m = q_x - q'_x - 1$; če pa je $q'_x = 0$, imamo pogoj $m \geq q_x - 1$. Hkrati imamo seveda tudi omejitev, da mora biti $m \geq 0$. Ko izračunamo presek vseh teh omejitev, nam za m ostaneta največ dve možni vrednosti, lahko pa tudi ena ali nobena. Če nobena, potem vemo, da naša b_t ni mogla nastati med urejanjem tabele b_1; če sta možna dva m-ja, sta to gotovo dve zaporedni števili, recimo $M-1$ in M, in je tabela b_t nastala po M v celoti izvedenih prehodih čez tabelo (torej iteracijah zunanje zanke algoritma iz besedila naloge), torej postavimo $m = M$. Če pa je možen en sam m, potem pač vzemimo tistega.

Zdaj za nek konkreten m vemo, da če je b_t nastala med urejanjem tabele b_1, je do tega prišlo po m v celoti izvedenih iteracijah zunanje zanke (in mogoče še po delno izvedeni $(m+1)$-vi iteraciji). Naj bo q'' stanje tabele inverzij po m v celoti izvedenih iteracijah zunanje zanke; torej $q''_x = \max\{0, q_x - m\}$. Iz q'' lahko (kot smo videli zgoraj) v $O(n \log n)$ časa rekonstruiramo stanje tabele a po tistih m iteracijah. Nato lahko na enak način kot v algoritmu 2 odsimuliramo $(m+1)$-vo iteracijo zunanje zanke in pri tem sproti preverjamo, ali se stanje tabele a po kakšni zamenjavi v celoti ujema z b_t; če se ne, potem vemo, da b_t pač ni mogla nastati med urejanjem tabele b_1. To preverjanje nam vzame le $O(n)$ časa, tako da imamo vsega skupaj s tabelo b_t le $O(n \log n)$ dela, za obdelavo celotne vhodne datoteke pa torej porabimo $O(nk \log n)$ časa.[13]

[13] Koristno branje o tabelah inverzij je na primer Knuth, *The Art of Computer Programming*, 3. knjiga, § 5.1.1; definicija tabele inverzij na str. 12; naloga 5.1.1.5 na str. 19 in njena rešitev na str. 578–9 pa podajata še en postopek za rekonstrukcijo permutacije iz njene tabele inverzij.

25. Lonci

Naloge se lahko lotimo s požrešnim algoritmom. Lonce pregledujmo po naraščajočem polmeru. Za vsak lonec L poglejmo med predhodnimi lonci tiste, ki so nižji od L; med njimi izberimo najvišjega (recimo mu N) in ga položimo v L (skupaj s tistimi, ki so že od prej v N, če je kaj takih).

Prepričajmo se, da nas ta postopek res pripelje do najboljše rešitve. Naša rešitev v L položi lonec N, torej najvišji tak lonec, ki je po premeru manjši od L; pa recimo, da bi namesto N-ja v L položili nek nižji lonec N'. Torej bodisi N pristane v nekem kasnejšem loncu L' ali pa celo ostane najnižji v svojem lastnem skladu. (1) Če ostane N v svojem skladu, bi ga lahko položili v L namesto lonca N', slednji pa bi potem imel svoj sklad; tako ne bi bili nič na slabšem (mogoče pa bi lahko N' kasneje celo položili v kak lonec, ki pride kasneje od L, in bi bili zato še kaj na boljšem). (2) Če pa N pristane v nekem kasnejšem loncu L' (torej takem, ki je širši od L), bi lahko N in N' zamenjali; N' dajmo v L' (saj če je L' višji od N, je gotovo tudi višji od N'), N pa dajmo v L, pa imamo še vedno veljavno rešitev. Tako torej lahko vsako rešitev, ki se razlikuje od požrešne, postopoma predelamo v požrešno, ne da bi se kaj poslabšala; torej je požrešna rešitev tudi optimalna.

Dobro je razmisliti še o tem, kako lahko pri posameznem L učinkovito poiščemo najvišji tak lonec, ki je po premeru manjši od njega. Koristno je vzdrževati neko drevesasto podatkovno strukturo, v katero dodajamo lonce, ki smo jih že pregledali. Ko gledamo lonec L, so v drevesu torej že vsi ožji lonci (taki z manjšim polmerom); urejeni pa naj bodo po višini. S takim drevesom lahko v $O(\log n)$ časa poiščemo najvišji lonec, ki je nižji od določene višine (v našem primeru od višine lonca L), pa tudi dodajanje novega lonca v drevo nam vzame $O(\log n)$ časa. Uporabimo lahko na primer kakšno od različic binarnega iskalnega drevesa (rdeče-črno drevo, AVL-drevo).

Namesto drevesa lahko uporabimo tudi nabor tabel: uredimo lonce po višini in jih oštevilčimo od 0 do $n-1$; za vsak k od 0 do $\lceil \log_2 n \rceil$ imejmo tabelo $n/2^k$ logičnih vrednosti, pri čemer i-ta med njimi pove, ali smo doslej že obdelali kakšnega od loncev s številkami od $2^k \cdot i$ do $2^k \cdot (i+1) - 1$. S pomočjo teh tabel lahko prav tako kot z drevesom v $O(\log n)$ časa poiščemo najvišji lonec, ožji od L, pa tudi dodajanje novega lonca v tabele nam vzame $O(\log n)$ časa. Časovna zahtevnost celotnega postopka je tako $O(n \log n)$.

Razmislimo še o težji različici naloge, pri kateri namesto loncev zlagamo škatle. Definiramo lahko graf, v katerem je za vsako škatlo ena točka, usmerjena povezava od ene točke do druge pa je prisotna tam, kjer lahko prvo škatlo položimo v drugo. Vsaka pot po tem grafu predstavlja neko možno skladovnico škatel. Ker hočemo škatle zložiti v čim manj skladovnic, smo prišli do problema pokrivanja grafa s čim manj potmi. Ta problem pa lahko prevedemo na problem pretoka v grafu ali pa na problem ujemanja v dvodelnem grafu.[14]

[14] Če je $G = (V, E)$ usmerjen graf, ki ga hočemo pokriti s čim manj potmi, in so točke (torej škatle) oštevilčene kot $V = \{1, \ldots, n\}$, lahko sestavimo nov dvodelen graf $G' = (V', E')$ s točkami $V' = \{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ in povezavami (x_u, y_v) za vsako $(u, v) \in E'$. Poiščimo maksimalno ujemanje v G', torej največjo množico povezav $M \subset E'$, za katero velja, da noben par povezav v M nima skupnega krajišča. Poženimo naslednji postopek:

za vsako v, če x_v ni krajišče nobene povezave v M:
  naj bo ρ nova pot, ki zaenkrat vsebuje le točko v;
  dokler je y_v krajišče kakšne povezave v M:

26. Ropar na podzemni

Železniško omrežje v tej nalogi pravzaprav tvori neusmerjen graf z n točkami in m povezavami. Označimo z M_t množico vseh tistih postaj, na katerih bi se ropar utegnil nahajati ob času t. Na začetku točno vemo, da je na postaji s, torej je $M_0 = \{s\}$. V nadaljevanju pa razmišljamo takole: ob času t se lahko ropar nahaja na postaji v natanko tedaj, če je bil ob času $t-1$ na eni od sosed postaje v in če ob času t na postaji v ni bilo nobenega agenta. Če označimo z $N(u)$ množico sosed postaje u, z A_t pa množico točk, kjer ob času t stojijo agentje, smo tako dobili rekurzivno zvezo $M_t = (\cup_{u \in M_{t-1}} N(u)) - A_t$. Naloga tako pravzaprav sprašuje, koliko elementov ima množica M_p.

Ostane le še vprašanje, kako te množice računati v praksi. Množico, kot je M_t, lahko predstavimo s tabelo n logičnih vrednosti (**bool** oz. boolean), ki nam za vsako postajo povedo, ali pripada tej množici ali ne. Enako lahko naredimo tudi za množice A_t, ki si jih pripravimo že ob branju poti agentov iz vhodne datoteke. Tudi sezname sosed za vsako točko lahko predstavimo s tabelami; ko pri branju vhodnih podatkov vidimo povezavo $(u : v)$, dodamo u v seznam sosed točke v in obratno. Tako pridemo do naslednje rešitve:

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

#define MaxN 10000
#define MaxM 100000
#define MaxP 100
#define MaxSosed 100

int nSosed[MaxN], sosede[MaxN][MaxSosed];
bool jeAgent[MaxP][MaxN]; /* jeAgent[t][u] = ali je ob času t na postaji u kakšen agent */
bool Mozna[2][MaxN];

int main()
{
  FILE *f = fopen("ropar.in", "rt");
  int n, m, k, s, p, u, v, i, t; bool *mozna, *prejMozna, *tmp;
  fscanf(f, "%d %d %d %d %d", &n, &m, &k, &p, &s); s--;
  /* Preberimo podatke o povezavah in za vsako postajo pripravimo seznam sosed. */
  for (u = 0; u < n; u++) nSosed[u] = 0;
  for (i = 0; i < m; i++) {
    fscanf(f, "%d %d", &u, &v); u--; v--;
    sosede[u][nSosed[u]++] = v;
    sosede[v][nSosed[v]++] = u; }
  /* Preberimo poti agentov in pripravimo tabelo, ki pove, kje in kdaj stojijo agentje. */
```

---

recimo, da je to povezava (x_u, y_v);
dodaj u na začetek poti ρ; $v := u$;
izpiši pot ρ;

Pokazati je mogoče, da izpiše ta postopek $n - |M|$ poti, ki pokrijejo prvotni graf G, in da se z manj kot toliko potmi grafa G sploh ne da pokriti. Namesto z ujemanjem v dvodelnem grafu lahko nalogo rešujemo tudi s pretoki: dodajmo v graf G' še točki s in t ter povezave (s, x_u) in (y_u, t) za vse $u \in V$; vsem povezavam pripišimo kapaciteto 1 in poiščimo največji možni pretok od s do t. Za več o teh stvareh glej npr. Cormen *et al.*, *Introduction to Algorithms*, naloga 27-2 (str. 692) v prvi izdaji. S podobnim problemom smo se srečali že v biltenu 2011 (rešitev naloge 2009.X.14, str. 102).

```
  for (t = 0; t < p; t++) for (u = 0; u < n; u++) jeAgent[t][u] = false;
  for (i = 0; i < k; i++) for (t = 0; t < p; t++) {
    fscanf(f, "%d", &v); jeAgent[t][v - 1] = true; }
  fclose(f);

  /* Na začetku vemo, da je ropar na postaji s in nikjer drugje. */
  mozna = Mozna[0]; prejMozna = Mozna[1];
  for (u = 0; u < n; u++) mozna[u] = false; mozna[s] = true;
  /* Poglejmo, kje bi lahko bil v kasnejših časovnih trenutkih. */
  for (t = 0; t < p; t++) {
    tmp = mozna; mozna = prejMozna; prejMozna = tmp;
    for (u = 0; u < n; u++) mozna[u] = false;
    for (u = 0; u < n; u++) if (prejMozna[u])
      /* Ker je bil v prejšnjem trenutku ropar lahko na postaji u, je v tem trenutku
         lahko na kateri koli sosedi postaje u. */
      for (i = 0; i < nSosed[u]; i++) mozna[sosede[u][i]] = true;
    /* Ropar ne more biti na tistih postajah, kjer so ob tem času agentje. */
    for (u = 0; u < n; u++) if (jeAgent[t][u]) mozna[u] = false; }

  /* Preštejmo možne postaje po p korakih in ta rezultat izpišimo. */
  for (u = 0, t = 0; u < n; u++) if (mozna[u]) t++;
  f = fopen("ropar.out", "wt"); fprintf(f, "%d\n", t); fclose(f); return 0;
}
```

Pri tej rešitvi je možnih še več različic, ki bi bile lahko v nekaterih primerih učinkovitejše. Na primer, zgornja rešitev računa M_t iz M_{t-1} na podlagi dejstva, da če je neka u dosegljiva v času $t-1$, potem so vse njene sosede dosegljive v času t. Lahko pa bi razmišljali tudi drugače: neka v je dosegljiva v času t, če ima kakšno tako sosedo, ki je dosegljiva v času $t-1$. Tako pridemo do:

```
  for (v = 0; v < n; v++)
    for (mozna[v] = false, i = 0; i < nSosed[v]; i++) {
      u = sosede[v][i]; if (prejMozna[u]) { mozna[v] = true; break; }}
```

V najslabšem primeru porabita obe različici $O(p \cdot m)$ časa. Prednost prve rešitve je, da moramo notranjo zanko (po u-jevih sosedah) opraviti le za tiste u, ki pripadajo množici M_{t-1}, za ostale u pa ne; druga rešitev pa mora seznam sosed pregledati pri vseh točkah v, vendar ne nujno do konca, saj se lahko ustavi takoj, ko ugotovi, da je kakšna v-jeva soseda pripadala množici M_{t-1}. To bi utegnilo biti hitreje, če so množice M_t praviloma velike (torej če je običajno roparju dosegljivih veliko postaj).

Potencialna slabost naše dosedanje rešitve je še ta, da hrani sezname sosed v dvodimenzionalni tabeli (sosede) velikosti $n \cdot D$, če je D največja možna stopnja točke v grafu (naloga zagotavlja, da je $D \leq 100$). Ker pa so vsi seznami skupaj dolgi le $2m$ elementov (vsaka povezava prispeva po en element v seznam sosed vsakega od obeh krajišč povezave), bi lahko z malo pazljivosti vse sezname skupaj hranili v eni sami tabeli s samo $O(m)$ elementi. Tudi pri shranjevanju množic A_t (v tabeli jeAgent) zdaj porabimo $O(n \cdot p)$ pomnilnika, kar je lahko potratno, če je agentov malo; če bi posamezno A_t predstavili le s seznamom postaj, na katerih ob času t stoji kakšen agent, bi za vse te sezname skupaj porabili le $O(k \cdot p)$ pomnilnika. Na asimptotično časovno zahtevnost postopka pa te spremembe ne bi vplivale in bi ta še naprej ostala $O(p \cdot m)$.

27. Prenos podatkov

Kot vidimo iz besedila naloge, moramo pred ustvarjanjem direktorija poskrbeti, da že obstaja njegov naddirektorij in da je ta naddirektorij takrat naš trenutni direktorij. Načeloma bi torej lahko vsako ime datoteke iz vhodnega seznama obravnavali čisto sistematično: recimo, da imamo ime oblike

$$\texttt{X:}\backslash d_1 \backslash d_2 \backslash \ldots \backslash d_n \backslash \texttt{ime.ext}$$

Če ga razbijemo pri vseh znakih \ in zavržemo zadnji del (`ime.ext`), lahko sistematično ustvarimo direktorije od korena naprej s takšnim zaporedjem ukazov:

$$\begin{aligned}
&\texttt{chdir X:}\backslash \\
&\texttt{mkdir } d_1 \\
&\texttt{chdir X:}\backslash d_1 \\
&\texttt{mkdir } d_2 \\
&\vdots \\
&\texttt{chdir X:}\backslash d_1 \backslash d_2 \backslash \ldots \backslash d_{n-1} \\
&\texttt{mkdir } d_n
\end{aligned}$$

Imena iz vhodnega seznama bi lahko obravnavali enega za drugim in pri vsakem izpisali takšno zaporedje ukazov. Slabost te rešitve je, da se pri njej lahko zgodi, da poskusimo isti direktorij ustvariti po večkrat (npr. če se v vhodnem seznamu najprej pojavi `X:\a\u.txt` in nato še `X:\a\v.txt`, bomo direktorij `X:\a` skušali ustvariti dvakrat). Besedilo naloge ne pove nič o tem, ali `mkdir` povzroči kakšno napako, če od njega zahtevamo, naj ustvari direktorij, ki že obstaja; za vsak primer (pa tudi zato, ker je naloga s tem malo težja in zanimivejša) pa je vendarle dobro razmisliti tudi o tem, kako bi se večkratnemu ustvarjanju istega direktorija izognili.

Ena možnost je, da v pomnilniku hranimo podatke o tem, katere direktorije smo že ustvarili; preden izpišemo ukaz `mkdir`, preverimo, ali ne bi moral ta direktorij obstajati že od prej (od nekega zgodnejšega `mkdir`a), in če bi, potem novi `mkdir` raje preskočimo. Takšna podatkovna struktura bi bila lahko seznam nizov, razpršena tabela ali pa kar drevo, ki bi odražalo strukturo nad- in poddirektorijev na disku.

Elegantno pa se lahko podvajanju izognemo tudi tako, da poti vseh direktorijev, ki se pojavljajo v vhodnem seznamu, uredimo po abecedi. V tako urejenem seznamu lahko zdaj primerjamo trenutno pot s prejšnjo; recimo, da se v prvih k segmentih ujemata, nato pa se razlikujeta:

$$\begin{aligned}
\text{prejšnja:}\quad & \texttt{X:}\backslash d_1 \backslash d_2 \backslash \ldots \backslash d_k \backslash d'_{k+1} \backslash \ldots \backslash d'_m \\
\text{trenutna:}\quad & \texttt{X:}\backslash d_1 \backslash d_2 \backslash \ldots \backslash d_k \backslash d_{k+1} \backslash \ldots \backslash d_n
\end{aligned}$$

Ta primerjava nam pove, da ko se ukvarjamo s trenutno potjo, gotovo že obstaja direktorij $\texttt{X:}\backslash d_1 \backslash d_2 \backslash \ldots \backslash d_k$ (saj smo ga ustvarili pri obravnavi prejšnje poti, če ne že prej); direktorij $\texttt{X:}\backslash d_1 \backslash d_2 \backslash \ldots \backslash d_k \backslash d_{k+1}$ pa gotovo še ne obstaja, saj je po abecedi večji od vseh doslej obdelanih poti. Torej točno vemo, da moramo začeti s $\texttt{chdir X:}\backslash d_1 \backslash d_2 \backslash \ldots \backslash d_k$ in ustvarjati poddirektorije od tu naprej.

Oglejmo si še implementacijo takšne rešitve v jeziku C++:

```
#include <string>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  vector<string> poti; int n; cin >> n;
  string s; getline(cin, s); // Preberimo konec vrstice, v kateri je bil n.
  while (n-- > 0) {
    getline(cin, s);
    // Pobrišimo ime datoteke, tako da ostane le pot do nje.
    size_t i = s.find('\\'), j = s.rfind('\\');
    // Datoteke v korenskem imeniku lahko preskočimo, saj ta imenik že obstaja.
    if (i < j) poti.push_back(s.substr(0, j)); }
  sort(poti.begin(), poti.end());
  for (size_t i = 0; i < poti.size(); i++)
  {
    s = poti[i]; const char *S = s.c_str();
    size_t prviBsl = s.find('\\');
    size_t j = prviBsl;
    if (i >= 0)
    {
      const char *p = S, *P = poti[i - 1].c_str();
      // Poglejmo, do katerega znaka \ se trenutna pot ujema s prejšnjo.
      for ( ; *p == *P && *p; p++, P++) if (*p == '\\') j = p - S;
      // Če sta poti enaki, se s trenutno potjo ni treba ukvarjati.
      if (*p == *P) continue;
      // Mogoče je do neujemanja prišlo na koncu prejšnje poti,
      // ker je trenutna poddirektorij prejšnje.
      if (! *P && *p == '\\') j = p - S;
    }
    while (s[j])
    {
      // Znak s[j] je \, pri katerem se konča pot do imenika, ki že obstaja.
      // Postavimo se v ta imenik.
      cout << "chdir " << s.substr(0, j + (j == prviBsl ? 1 : 0)) << endl;
      // Poiščimo naslednji \.
      const char *p = S + j + 1; while (*p && *p != '\\') p++;
      size_t k = p - S;
      // Izpišimo ukaz za ustvaritev podimenika in se postavimo na naslednji \.
      cout << "mkdir " << s.substr(j + 1, k - j - 1) << endl;
      j = k;
    }
  }
  return 0;
}
```

Potencialna pomanjkljivost te rešitve je, da loči med velikimi in malimi črkami. Če dobimo v vhodnem seznamu na primer `X:\a\b.txt` in nato `X:\A\c.txt`, bo program poskusil ustvariti direktorija `X:\a` in `X:\A`; naloga nič ne govori o tem, ali sta v našem hipotetičnem datotečnem sistemu to dva različna direktorija ali pa le dva različna zapisa imena enega in istega direktorija (in bi torej mogoče `mkdir` povzročil napako,

če ga bomo skušali ustvariti dvakrat). Če bi se hoteli izogniti tej možnosti, bi morali pri urejanju nizov in pri primerjanju trenutne poti s prejšnjo obravnavati velike in male črke kot enakovredne (ali pa že ob branju vhodnega seznama spremeniti vse velike črke v male).

28. Kendallov koeficient

Za začetek se omejimo na računanje koeficienta τ med dvema konkretnima sodnikoma x in y; s posplošitvijo na več sodnikov se bomo ukvarjali kasneje. Ker imamo le dva konkretna sodnika, tudi pri številu skladnih in neskladnih parov ni treba posebej omenjati x in y, tako da ju lahko označimo preprosto kot n_c in n_d, naš koeficient pa je zdaj $\tau = (n_c - n_d)/n_0$.

Posvetimo se najprej štetju neskladnih parov, n_d. Na število teh parov seveda nič ne vpliva to, v kakšnem vrstnem redu imamo tekmovalce oštevilčene oz. zapisane. Zato je koristno, če tekmovalce najprej uredimo naraščajoče po x_i; v nadaljevanju bomo torej predpostavili, da je $x_1 \le x_2 \le \ldots \le x_n$. Očitno bi lahko neskladne pare prešteli takole:

$n_d := 0$;
for $j := 2$ **to** n:
  $n_d := n_d +$ število takih $i \in \{1, 2, \ldots, j-1\}$, ki tvorijo neskladne pare z j;

Vprašanje je, kako naj pri posameznem j ugotovimo, koliko i-jev tvori neskladne pare z njim. Lahko bi naredili gnezdeno zanko po i, vendar bi tako nastal postopek s časovno zahtevnostjo $O(n^2)$; ker pa naloga pravi, da je n lahko velik, bi radi našli učinkovitejšo rešitev.

Spomnimo se definicije neskladnega para: i in j sta neskladna, če je $x_i < x_j$ in $y_i > y_j$ ali pa $x_i > x_j$ in $y_i < y_j$. Druga možnost v našem primeru odpade, ker vedno gledamo $i < j$ in ker smo naše tekmovalce oštevilčili po naraščajočih x. Ostane torej le prva možnost: koliko tekmovalcev $i \in \{1, 2, \ldots, j-1\}$ ima $x_i < x_j$ in $y_i > y_j$? Če smemo predpostaviti, da posamezni sodnik ne more dati enake ocene različnim tekmovalcem, potem vemo, da je pogoj $x_i < x_j$ pri vseh $i < j$ že izpolnjen (saj imamo tekmovalce urejene po x) in nam ostane le še vprašanje, koliko izmed teh tekmovalcev ima $y_i > y_j$. Pri tem si lahko pomagamo z drevesom:

$n_d := 0$; $T :=$ prazno drevo;
for $j := 2$ **to** n:
  $n_d := n_d +$ število elementov drevesa T, ki so večji od y_j;
  dodaj y_j v drevo T;

Če za T uporabimo kakšno primerno podatkovno strukturo, bomo lahko tako poizvedbo po drevesu (torej štetje elementov, ki so večji od y_j) kot dodajanje vanj izvedli v času $O(\log n)$, tako da bo celoten postopek trajal $O(n \log n)$ časa (toliko ga porabimo tudi za urejanje tekmovalcev po x). Ena možnost je na primer kakšno od uravnoteženih binarnih iskalnih dreves (AVL-drevo, rdeče črno drevo), pri katerem pa moramo v vsakem vozlišču vzdrževati še podatek o številu vseh vozlišč v njegovem poddrevesu. Če so ocene sodnika y cela števila od 1 do n, bi lahko T implementirali tudi kot družino tabel $a_0, a_1, \ldots$, pri čemer je a_k dolga $n/2^k$ elementov in v elementu $a_k[z]$ hranimo podatek o tem, koliko elementov T-ja leži na območju $[z \cdot 2^k, (z+1) \cdot 2^k)$.

Še bolj elegantno bi šlo tudi s Fenwickovim drevesom (kjer potrebujemo pravzaprav eno samo tabelo z $n+1$ elementi).

Če ne smemo predpostaviti, da imajo različni tekmovalci različne ocene, se postopek malo zaplete. Gornji postopek bi lahko v tem primeru dajal napačne rezultate, saj je izhajal iz predpostavke, da ko se ukvarjamo s tekmovalcem j, je pogoj $x_i < x_j$ izpolnjen za vse tekmovalce i iz drevesa T (zato je moral preveriti le še y-ocene). Ta pogoj pa ne drži več, če smejo imeti različni tekmovalci enako oceno — lahko se na primer zgodi, da je $x_{j-1} = x_j$, zato tekmovalec $j-1$ ne bo tvoril neskladnega para z j, ne glede na to, kakšna sta y_{j-1} in y_j. Temu problemu se izognemo, če tekmovalca dodamo v drevo T šele po tistem, ko smo že obdelali vse druge tekmovalce z enako x-oceno. V spodnji psevdokodi nam spremenljivka m pove indeks prvega takega tekmovalca, ki ga še nismo dodali v drevo.

```
n_d := 0; T := prazno drevo; m := 1;
for j := 2 to n:
  n_d := n_d+ število elementov drevesa T, ki so večji od y_j;
  if j = n or x_j < x_{j+1}:
    while m ≤ j:
      dodaj y_m v drevo T;  m := m + 1;
```

Zgoraj smo videli, da je za nekatere implementacije strukture T koristno, če so ocene kar cela števila od 1 do n. Če v naših vhodnih podatkih to ne drži, lahko podatke pred nadaljnjo obdelavo predelamo: pripravimo si seznam parov (y_i, i) in jih uredimo po y; prvemu tekmovalcu (tistemu z najmanjšo oceno) popravimo y_i na 1, drugemu na 2 in tako naprej. Paziti moramo le še na to, da če ima več zaporednih tekmovalcev enako oceno, morajo tudi po popravku dobiti vsi enako oceno. V spodnji psevdokodi nam spremenljivka k pove, do katere ocene smo pri tem popravljanju že prišli.

```
L[1..n] := tabela parov (y_i, i) za i = 1, ..., n;
uredi L naraščajoče po y_i;
k := 0;
for t := 1 to n:
  if t = 1 or L[t].y > L[t − 1].y then k := k + 1;
  i := L[t].i; y_i := k;
```

Vidimo, da nam tudi da postopek porabi le $O(n \log n)$ časa (zaradi urejanja seznama L), tako da nam nič ne poslabša asimptotične časovne zahtevnosti celotne rešitve. Enako kot tu za ocene y ga lahko izvedemo tudi za ocene vsakega od preostalih sodnikov.

Oglejmo si še malo drugačen pristop k štetju neskladnih parov. Naše tekmovalce predstavimo kot zaporedje parov (x_t, y_t) za $t = 1, \ldots, n$ in jih uredimo naraščajoče po x, tiste z enakim x pa naraščajoče po y. Recimo zdaj, da bi hoteli to zaporedje urediti naraščajoče po y z vstavljanjem (*insertion sort*):

```
vhod: tabela a[1..n], pri čemer je a[t] = (x_t, y_t); urejena je naraščajoče po x,
      tekmovalci z enakim x pa naraščajoče po y;
s := 0;
for j := 2 to n:
  (* Invarianta: v a[1..j − 1] so isti tekmovalci kot ob začetku izvajanja tega
```

algoritma, le da so urejeni naraščajoče po y (tisti z enakim y pa naraščajoče po x); v s je število vseh takih neskladnih parov, pri katerih sta oba tekmovalca iz a[1..j − 1]. *)
$i := j - 1$; $y^\star := a[j].y$;
while $i > 0$ **and** $y_i > y^\star$:
zamenjaj $a[i]$ in $a[i+1]$; $i := i - 1$; $s := s + 1$;

Podrobnosti dokaza, da invarianta res drži za začetku vsake iteracije glavne zanke, prepuščamo bralcu za vajo. Za naš namen je pri tem postopku zanimivo predvsem to, da je ob urejanju tabele po y uspel tudi prešteti neskladne pare. To je posledica dejstva, da ko se tekmovalec iz $a[j] = (x_j, y_j)$ premika (v notranji zanki) nazaj po tabeli, gre v vsakem koraku mimo nekega takega tekmovalca (x_i, y_i), ki ima $y_i > y_j$ (če to ne bi držalo, bi se zanka **while** že ustavila) in $x_i < x_j$ (to je posledica tega, da je bila vhodna tabela urejena naraščajoče po x in pri enakem x tudi naraščajoče po y). Ko se notranja zanka ustavi, imajo vsi tekmovalci na manjših indeksih i tudi manjšo (ali enako) y-oceno kot tekmovalec j, ki smo ga doslej premikali nazaj po tabeli, obenem pa seveda tudi manjšo ali enako x-oceno (ker je bila vhodna tabela urejena po x), torej gotovo ne tvorijo neskladnega para z njim. Tako torej vidimo, da je ta postopek res preštel točno vse neskladne pare.

Slabost tega postopka je, da ima časovno zahtevnost $O(n^2)$. Lahko pa ga izboljšamo takole. Začnimo z našo tabelo $a[1..n]$ (urejeno po x in pri enakem x še po y) in jo razdelimo na dva približno enaka dela: levi del $L = a[1..m]$ in desni del $D = a[m+1..n]$, pri čemer je $m \approx n/2$. Vsak od teh delov je sam zase seveda še vedno urejen po x in bi lahko na njem pognali zgoraj opisani postopek, ki se zgleduje po urejanju z vstavljanjem. Tako bi dobili število neskladnih parov v L (recimo s_L) in število neskladnih parov v D (recimo s_D). Vsi ti pari so seveda neskladni tudi v prvotni tabeli a. Do skupnega števila neskladnih parov v a nam manjkajo le še tisti pari, ki imajo enega tekmovalca iz L in enega iz D (recimo, da je takih parov s_M). Te pare lahko preštejemo ob zlivanju zaporedij L in D v novo zaporedje U, v katerem so tekmovalci iz L in D vsi skupaj urejeni naraščajoče po y (pri enakem y pa naraščajoče po x):

algoritem Zlivanje:
vhod: levi del $L[1..m]$ in desni $D[1..r]$, $r = n - m$;
vsak od njiju je sam zase že urejen (naraščajoče po y in pri istem y naraščajoče po x); nastala pa sta kot levi in desni del vhodne tabele a, ki je bila urejena naraščajoče po x (pri enakem x pa naraščajoče po y); to med drugim pomeni, da za vsakega tekmovalca (x_i, y_i) iz L in vsakega (x_j, y_j) iz D velja $x_i < x_j$ ali pa $x_i = x_j$ in $y_i \le y_j$;
izhod: zlito zaporedje $U[1..n]$; in s_M, število neskladnih parov z enim tekmovalcem iz L in enim iz D;

$i := 0$; $j := 0$; $k := 0$; $s_M := 0$;
while $i < m$ **or** $j < r$:
(* *Invarianta:*
(*1*) *v* $U[1..k]$ *smo že prepisali člene iz* $L[1..i]$ *in* $D[1..j]$ *in to tako, da je* $U[1..k]$ *urejen naraščajoče po y (in pri istem y naraščajoče po x);*
(*2*) *obenem je seveda vsak od* $L[1..i]$ *po y manjši od* $D[j+1]$ *(ali pa po y*

enak in nato po x manjši ali enak); in vsak od $D[1..j]$ je po y manjši od $L[i+1]$ (ali pa po y enak in nato po x manjši ali enak);

(3) v s_M je število vseh takih neskladnih parov, v katerih je en tekmovalec iz $L[1..n]$ in en iz $D[1..j]$. *)

(* *Za potrebe naslednjega if-a si mislimo $L[m+1] = D[r+1] = (\infty, \infty)$.* *)
if $D[j+1].y < L[i+1].y$:
$k := k+1$; $j := j+1$; $U[k] := D[j]$;
$s_M := s_M + m - i$;
else:
$k := k+1$; $i := i+1$; $U[k] := L[i]$;
return (U, s_M);

Ob koncu tega postopka imamo v $U[1..n]$ vse tekmovalce iz tabel L in D, urejene naraščajoče po y (in pri enakem y še naraščajoče po x); obenem pa imamo v s število neskladnih parov, v katerih nastopa po en tekmovalec iz L in en iz D. Prvi dve točki invariante govorita le o tem, da gre res za zlivanje, ki bo dalo pravilno urejeno izhodno zaporedje (vendar pa ju potrebujemo pri dokazovanju tretje točke). Pri tretji točki pride do spremembe le, ko se j poveča, torej če je $D[j+1].y < L[i+1].y$ in smo v izhodno zaporedje prenesli naslednjega tekmovalca iz D. Takrat se moramo vprašati, s koliko tekmovalci zaporedja L je tvoril naš $D[j+1]$ (ki bo kmalu postal $D[j]$, ker se bo j povečal za 1) neskladne pare. Po x je $D[j+1]$ večji od vseh iz L, po y pa je večji od tistih iz $L[1..i]$ (kar prepoznamo po tem, da so $L[1..i]$ že prišli v izhodno zaporedje, $D[j+1]$ pa bo tja prišel šele zdaj) in manjši od tistih iz $L[i+1..m]$ (kar prepoznamo po tem, da bomo zdaj skopirali v izhodno zaporedje tekmovalca $D[j+1]$ in ne naslednjega iz L, to je $L[i+1]$). Neskladne pare tvori torej natanko z vsemi tekmovalci iz $L[i+1..m]$, teh pa je $m-i$, zato moramo tudi s_M povečati za $m-i$. (Morali bi razmisliti še o nekaj malenkostih v primerih, ko ima več tekmovalcev enake ocene. Te podrobnosti, kakor tudi dokaz, da držita tudi prvi dve točki invariante, prepuščamo bralcu za vajo.)

Malo prej smo rekli, da bi pred tem zlivanjem najprej uredili levi in desni del posebej in prešteli neskladne pare v vsakem od njiju. Takrat smo rekli, da bi za to lahko uporabili prej opisano različico urejanja z vstavljanjem; še bolje pa je, če bi tudi vsakega od teh dveh delov razbili na dva pol manjša dela, uredili vsakega posebej in ju nato zlili. Tako pridemo do naslednjega rekurzivnega postopka:

algoritem Rekurzija:
vhod: tabela $a[1..n]$, urejena naraščajoče po x (in pri istem x še po y);
izhod: tabela $a'[1..n]$, v kateri so tekmovalci iz a urejeni naraščajoče po y (in pri istem y še po x); in s, število neskladnih parov v tabeli a;

(* *Rekurzija se ustavi, če je tabela dovolj kratka.* *)
if $n = 1$ **then return** $(a, 0)$;
(* *Razbijmo a na dva dela.* *)
$m := \lfloor n/2 \rfloor$; $L := a[1..m]$; $D := a[m+1..n]$;
(* *Z rekurzijo uredimo vsak del posebej.* *)
$(L', s_L) :=$ Rekurzija(L); $(D', s_D) :=$ Rekurzija(D);
(* *Zlijmo oba urejena dela.* *)
$(a', s_M) :=$ Zlivanje(L, D);
return $(a', s_L + s_D + s_M)$;

Namesto pri $n = 1$ bi lahko rekurzijo ustavili že prej — ko je seznam a dovolj kratek, je verjetno hitreje, če z rekurzijo in zlivanjem končamo in uporabimo kar urejanje z vstavljanjem.

Kakšna je časovna zahtevnost tega postopka? Pri klicu s tabelo dolžine n nam nastaneta dva rekurzivna klica na tabelah dolžine $n/2$, poleg tega pa imamo še $O(n)$ dela z zlivanjem. Tako imamo $T(n) = 2 \cdot T(n/2) + O(n)$, za kar se hitro izkaže, da je $T(n) = O(n \log n)$. Pred vsem skupaj imamo tudi še $O(n \log n)$ dela za začetno urejanje tekmovalcev po x. Časovna zahtevnost postopka je torej $O(n \log n)$, enako kot pri rešitvi z drevesom.

Doslej smo se ukvarjali s štetjem neskladnih parov, n_d; v formuli za $\tau = (n_c - n_d)/n_0$ pa nastopa tudi število skladnih parov, n_c. Če velja omejitev, da sodnik ne more dati dvema ali več tekmovalcem enake ocene, potem je vsak par tekmovalcev bodisi skladen bodisi neskladen in je $n_c = n_0 - n_d$, tako da nam n_c ni treba računati posebej. Če pa te omejitve ni, se lahko pri nekaterih parih zgodi, da niso niti skladni niti neskladni; do tega pride, če je $x_i = x_j$ in/ali $y_i = y_j$. Do n_c bi lahko prišli z zelo podobnimi postopki, kot smo jih doslej uporabljali za izračun n_d; lahko bi tudi vse y-ocene pomnožili z -1, x-ocene pa pustili nespremenjene in na tako popravljenih podatkih prešteli neskladne pare; s tem bi dobili ravno število skladnih parov v originalnih podatkih.

Še ena možnost pa je naslednja. Naj bo n_x število parov tekmovalcev, ki imajo $x_i = x_j$; n_y naj bo število parov, ki imajo $y_i = y_j$; in n_{xy} naj bo število parov, ki imajo $x_i = x_j$ in $y_i = y_j$. Skupno število parov, ki niso niti skladni niti neskladni, je potem $n_t := n_x + n_y - n_{xy}$ — pri tem smo morali odšteti n_{xy} zato, ker so bili ti pari zajeti tako v n_x kot v n_y in jih je torej vsota $n_x + n_y$ štela dvakrat. Zdaj lahko n_c računamo kot $n_0 - n_d - n_t$. Lepo pri tem je, da je izračun števil n_x, n_y in n_{xy} zelo preprost. Ko imamo tekmovalce urejene po x, nam v tem vrstnem redu pridejo skupaj taki z enakim x; moramo se le sprehoditi po tem seznamu in ko v njem naletimo na skupino k zaporednih tekmovalcev z enakim k, vemo, da nam ta skupina prispeva $k(k-1)/2$ parov k številu n_x. Podobno lahko računamo tudi n_y in n_{xy}. S tem ni veliko dodatnega dela, saj bomo primerno urejene sezname tekmovalcev tako ali tako potrebovali pri izračunu n_d.[15]

Zdaj torej znamo izračunati τ za poljuben par sodnikov (x, y). Naloga zahteva, da ga moramo izračunati za vse pare sodnikov. Tu lahko prihanimo nekaj časa, če upoštevamo, da je $\tau(x, y) = \tau(y, x)$, torej nam ni treba računati obojega. Lahko tudi pri istem x obdelamo več različnih y tako, da uporabimo postopek z drevesom in hkrati (v enem samem prehodu čez seznam tekmovalcev) gradimo več dreves, po eno za vsak y; vendar pa s tem asimptotično nismo ničesar pridobili in časovna zahtevnost celotnega postopka je še vedno $O(k^2 n \log n)$.

29. Kidanje

Mislimo si usmerjen graf s točkami $1, 2, \ldots, n+1$; povezava $u \to v$ obstaja, če človek, ki kida dvorišče u, lahko vrže sneg na dvorišče v (torej če je $u < v \leq u + d_u$). Če se u

[15] Za več o Kendallovem koeficientu glej Wikipedijo *s. v.* Kendall tau rank correlation coefficient; W. R. Knight, *A computer method for calculating Kendall's tau with ungrouped data*, J. of the American Statistical Association, 61(314):436–439, June 1966; D. Christensen, *Fast algorithms for the calculation of Kendall's* τ, Computational Statistics, 20(1):51–62, March 2005.

zbudi pred v, naj bo dolžina te povezave 0, sicer pa 1. Poleg tega imejmo še povezavo $(u, n+1)$ dolžine 1 za vsak $u \leq n$. Označimo z $D(u)$ dolžino najkrajše poti od u do $n+1$; to je zdaj ravno najmanjše število dni, v katerem je mogoče spraviti sneg z dvorišča u na končno dvorišče $(n+1)$. Ker nas zanima, v koliko dneh je mogoče počistiti vsa dvorišča, moramo izračunati $\max\{D(u) : 1 \leq u \leq n\}$.

Pri iskanju najkrajših poti si lahko pomagamo z dejstvom, da lahko sneg odmetavamo le v eno smer (od manjših številk vozlišč proti večjim). Če se najkrajša pot od u do $n+1$ začne s korakom $u \to v$, potem vsekakor velja $v > u$ in nadaljevanje te poti mora biti najkrajša pot od v do $n+1$. Pri računanju najkrajše poti od u do $n+1$ je torej koristno, če že poznamo najkrajše poti od v do $n+1$ za dvorišča $v > u$. Vemo tudi, da je prvi korak $u \to v$ mogoč le, če je $u < v \leq u + d_u$; to je torej interval v-jev, ki jih moramo pregledati, ko računamo $D(u)$. Tako smo dobili naslednji postopek:

$D[n+1] := 0;$
for $u := n, n-1, n-2, \ldots, 1$:
  $D[u] := \infty;$
  for $v := u+1$ **to** $u + d_u$:
    if se v zbudi pred u **then** $c := D[v] + 1$ **else** $c := D[v]$;
    if $c < D[u]$ **then** $D[u] := c$;

Pri vsakem u torej pogledamo vse možne prve korake na poti od u do $n+1$; v c izračunamo dolžino poti, ki jo sestavlja korak $u \to v$ in potem najkrajša pot od v do $n+1$; med tako dobljenimi c-ji si zapomnimo najmanjšega. Ko je ta postopek končan, poznamo dolžine najkrajših poti od vseh točk $1..n$ do $n+1$ in moramo se le še sprehoditi po tabeli D in poiskati največjo vrednost v njej.

Slaba stran te rešitve je njena časovna zahtevnost; v najslabšem primeru (če so razdalje d_u dovolj velike) imamo z notranjo zanko lahko pri posameznem u-ju do $O(n)$ dela in časovna zahtevnost celotnega postopka je zato $O(n^2)$. Pri večjih n bi bil torej ta postopek neugodno počasen.

Ko računamo $D[u]$, si pomagamo z vrednostmi $D[v]$ za $u < v \leq u + d_u$. Te vrednosti lahko v mislih razdelimo na dve skupini: tiste, pri katerih prištejemo 1 (ker se stanovalec v zbudi prej kot u, zato bo lahko sneg, ki mu ga bo u nametal na dvorišče, skidal šele naslednji dan), in tiste, pri katerih tega ne storimo (ker se stanovalec v zbudi kasneje kot u, zato bo lahko sneg, ki mu ga bo u nametal na dvorišče, še isti dan skidal naprej). Označimo s t_u čas, ob katerem se zbudi stanovalec u; besedilo naloge pravi, da je i_j številka stanovalca, ki se zbudi j-ti po vrsti, torej postavimo $t_{i_j} = j$ za vsak $j = 1, \ldots, n$. Zdaj lahko izračun $D[u]$ zapišemo takole:

$$D[u] := \max\{\quad \max\{D[v] : u < v \leq u + d_u, t_v > t_u\},$$
$$1 + \max\{D[v] : u < v \leq u + d_u, t_v < t_u\}\}.$$

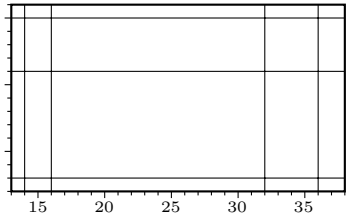
Doslej smo si ulico predstavljali kot nekakšno premico in stanovalce kot točke na njej; toda na taki premici je težko ločiti tiste, ki se zbudijo pred u, od tistih, ki se zbudijo za u. Lažje bo, če bomo stanovalce predstavili kot točke v ravnini: stanovalcu u naj pripada točka s koordinatama (u, t_u); poleg x-koordinate (položaj hiše na ulici) imamo zdaj še y-koordinato, ki nam pove čas, ob katerem se stanovalec u zbudi (oz. natančneje, pove nam položaj tega stanovalca v vrstnem redu zbujanja). Obe koordinati sta pri vsakem stanovalcu celi števili iz $\{1, \ldots, n\}$.

Oba maksimuma, s katerima smo se srečali zgoraj pri formuli za $D[u]$, sta zdaj pravzaprav maksimuma po vseh točkah na nekem pravokotnem območju: enkrat je to $[u+1, u+d_u] \times [1, t_u-1]$, enkrat pa $[u+1, u+d_u] \times [t_u+1, n]$ (v primerih, ko je $t_u = 1$ ali pa $t_u = n$, je eno od obeh območij prazno). Prišli smo torej do vprašanja, kako bi za neko tako pravokotno poizvedovalno območje učinkovito izračunali $\max D[v]$ po vseh točkah v na tem območju.

V ta namen si lahko pomagamo s kakšno prostorsko podatkovno strukturo, na primer k-d drevesom (v našem primeru za $k = 2$, ker smo v dveh dimenzijah). Izkaže se, da lahko takšno drevo zgradimo v $O(n \log n)$ časa, posamezna poizvedba po njem (s pravokotnim območjem) pa lahko traja do $O(\sqrt{n})$ časa. Ker moramo pri vsakem u izvesti dve takšni poizvedbi, da izračunamo $D[u]$, bi bila časovna zahtevnost celotnega postopka $O(n\sqrt{n})$. Potrebovali bi tudi $O(n)$ pomnilnika za predstavitev drevesa.

Še ena možnost pa je naslednja. Izberimo si števili a in b in razrežimo našo ravnino na pravokotne celice velikosti $2^a \times 2^b$: za vsak par celih števil (c, d) imejmo torej celico $R_{ab}(c, d) := [c \cdot 2^a, (c+1) \cdot 2^a) \times [d \cdot 2^b, (d+1) \cdot 2^b)$. Naj bo $M_{ab}(c, d)$ maksimum vrednosti $D[v]$ po vseh točkah v, ki ležijo v $R_{ab}(c, d)$. Nekatere celice so seveda lahko prazne (ne vsebujejo nobene izmed naših n točk) in pri njih si mislimo $M_{ab}(c, d) = -\infty$. Pravzaprav je nepraznih celic lahko največ n, saj je tudi točk samo n. Hranimo jih v razpršeni tabeli (*hash table*) H_{ab}, v kateri kot ključ uporabimo par (c, d), kot pripadajočo vrednost pa $M_{ab}(c, d)$. Takšne tabele ni težko popravljati, ko za nekega stanovalca u izračunamo vrednost $D[u]$. Ugotoviti moramo, kateri celici (c, d) pripada točka (u, t_u): to je $c = \lfloor u/2^a \rfloor$ in $d = \lfloor t_u/2^b \rfloor$; potem pa, če v H_{ab} še ni ključa (c, d), ga dodamo (s pripadajočo vrednostjo $D[u]$), sicer pa za obstoječi ključ (c, d) v H_{ab} pogledamo pripadajočo vrednost in če je le-ta manjša od $D[u]$, jo postavimo na $D[u]$. V vsakem primeru je to operacija, ki vzame povprečno $O(1)$ časa.

Takšno razpršeno tabelo H_{ab} bomo vzdrževali za vse možne pare $a, b \in \{0, \ldots, m\}$ za $m = \lfloor \log_2 n \rfloor$. Lepo pri tem naboru tabel je, da lahko zdaj vsako poizvedovalno območje, ki nas zanima pri izračunu $D[u]$, sestavimo iz največ $O((\log_2 n)^2)$ takšnih pravokotnih celic različnih velikosti; za vsako od njih lahko v pripadajoči tabeli H_{ab}) v času $O(1)$ poiščemo maksimum $D[v]$ po vseh točkah iz te celice, tako da lahko maksimum za celo poizvedovalno bomočje izračunamo v času $O((\log_2 n)^2)$. Spodnja slika kaže primer, kako lahko na celice razdelimo pravokotno poizvedovalno območje $[13, 38) \times [7, 21)$:



Primer, kako lahko pravokotno poizvedovalno območje $[13, 38) \times [7, 21)$ razdelimo na celice velikosti $2^a \times 2^b$ (za različne a in b), za katere imamo v tabelah H_{ab} že pri roki maksimume vrednosti $D[u]$ po vseh točkah u v celici. Interval $[13, 38)$ razdelimo na $[13, 14)$, $[14, 16)$, $[16, 32)$, $[32, 36)$ in $[36, 38)$; podobno tudi interval $[7, 21)$ razdelimo na $[7, 8)$, $[8, 16)$, $[16, 20)$ in $[20, 21)$. Celice, ki jih moramo pregledati, so ravno vsi možni kartezični produkti teh podintervalov.

Oglejmo si pobliže postopek za delitev intervala na primerne podintervale. Za primer vzemimo $[13, 38)$ z gornje slike. Ko začnemo pri $x = 13$, opazimo, da je to število liho, pri lihih x-koordinatah pa se začnejo le celice širine 1; zato uporabimo interval $[13, 14)$ in pridemo na $x = 14$. To število je večkratnik 2, ne pa tudi večkratnik 4, zato

se pri tem x lahko začne celica širine 2, ne pa kakšna širša; vzemimo torej interval $[14, 16)$ in se premaknimo na $x = 16$. Slednji je večkratnik 16, ne pa večkratnik 32; najširša celica, ki se začne pri $x = 16$, ima torej širino 16, zato vzemimo interval $[16, 32)$ in se postavimo na $x = 32$. Slednji je večkratnik 32, ne pa večkratnik 64, zato se pri $x = 32$ načeloma začne celica širine 32; toda interval $[32, 64)$, ki bi s tem nastal, nam nič ne pomaga, ker sega že čez desni rob našega poizvedovalnega območja ($x = 38$). Najširša celica, ki se začne pri $x = 32$ in še ne sega čez $x = 38$, je celica širine 4; tako dobimo podinterval $[32, 36)$ in se postavimo na $x = 36$. Pri tem z enakim razmislekom vidimo, da moramo uporabiti celico širine 2, torej imamo še podinterval $[36, 38)$ in postopek je končan. Podoben razmislek bi morali opraviti tudi pri y-koordinatah.

Ta postopek si lahko predstavljamo še bolj elegantno, če koordinate pišemo v dvojiškem zapisu. Recimo, da razbijamo interval $[l, r)$ na podintervale.

$t := l$;
while true:
  naj bo b najnižji prižgani bit v t; $t' := t + 2^b$;
  if $t' > r$ **then break**;
  izpiši podinterval $[t, t')$; $t := t'$;
$s := r$;
while $s > t$:
  $s' :=$ s brez najnižjega prižganega bita;
  izpiši podinterval $[s', s)$; $s := s'$;

Izračun t' iz t in s' iz s je mogoče izvesti zelo preprosto z logičnimi operatorji na bitih: $t' = t + (t \;\&\; (t-1))$ in $s' = s - (s \;\&\; (s-1))$.

Po tistem, ko izračunamo $D[u]$ za nek u, bomo porabili še $O((\log_2 n)^2)$ za popravljanje vseh tabel (saj smo zgoraj videli, da porabimo $O(1)$ časa za popravljanje vsake tabele). Časovna zahtevnost celotnega postopka je torej zdaj le še $O(n(\log n)^2)$, slabost pa je, da porabimo tudi $O(n(\log n)^2)$ pomnilnika (za vzdrževanje vseh tabel H_{ab}).

Naloge so sestavili: CamelCase, Galci in Rimljani, Doroteja, drevo — Nino Bašić; koren besed, špekulacije, potapljanje ladjic — Andrej Bauer; AVL-drevo — Andrej Brodnik; skakačev obhod — Primož Gabrijelčič; povprečne temperature — Matija Grabnar; podniz, kamere, produkt, lonci — Tomaž Hočevar; pravokotni meseci — Nace Hudobivnik; zemljevid, ropar na podzemni, prenos podatkov — Jurij Kodre; mutacije — Aleš Košir; DNSx20 — Mark Martinec; vagoni, zanimivi datumi, računovodstvo — Mitja Lasič; birokrati, konjska vprega, Kendallov koeficient, kidanje — Mitja Trampuš; semaforji — Klemen Žagar; bubble sort — Janez Brank.

NASVETI ZA MENTORJE O IZVEDBI ŠOLSKEGA TEKMOVANJA IN OCENJEVANJU NA NJEM

[Naslednje nasvete in navodila smo poslali mentorjem, ki so na posameznih šolah skrbeli za izvedbo in ocenjevanje šolskega tekmovanja. Njihov glavni namen je bil zagotoviti, da bi tekmovanje potekalo na vseh šolah na približno enak način in da bi ocenjevanje tudi na šolskem tekmovanju potekalo v približno enakem duhu kot na državnem.—*Op. ur.*]

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). Pri reševanju si lahko tekmovalci pomagajo tudi z literaturo in/ali zapiski, ni pa mišljeno, da bi imeli med reševanjem dostop do interneta ali do kakšnih datotek, ki bi si jih pred tekmovanjem pripravili sami. Čas reševanja je omejen na 180 minut.

Nekatere naloge kot odgovor zahtevajo program ali podprogram v kakšnem konkretnem programskem jeziku, nekatere naloge pa so tipa „opiši postopek“. Pri slednjih je načeloma vseeno, v kakšni obliki je postopek opisan (naravni jezik, psevdokoda, diagram poteka, izvorna koda v kakšnem programskem jeziku, ipd.), samo da je ta opis dovolj jasen in podroben in je iz njega razvidno, da tekmovalec razume rešitev problema.

Glede tega, katere programske jezike tekmovalci uporabljajo, naše tekmovanje ne postavlja posebnih omejitev, niti pri nalogah, pri katerih je rešitev v nekaterih jezikih znatno krajša in enostavnejša kot v drugih (npr. uporaba perla ali pythona pri problemih na temo obdelave nizov).

Kjer se v tekmovalčevem odgovoru pojavlja izvorna koda, naj bo pri ocenjevanju poudarek predvsem na vsebinski pravilnosti, ne pa na sintaktični. Pri ocenjevanju na državnem tekmovanju zaradi manjkajočih podpičij in podobnih sintaktičnih napak odbijemo mogoče kvečjemu eno točko od dvajsetih; glavno vprašanje pri izvorni kodi je, ali se v njej skriva pravilen postopek za rešitev problema. Ravno tako ni nič hudega, če npr. tekmovalec v rešitvi v C-ju pozabi na začetku **#include**ati kakšnega od standardnih headerjev, ki bi jih sicer njegov program potreboval; ali pa če podprogram `main()` napiše tako, da vrača **`void`** namesto **`int`**.

Pri vsaki nalogi je možno doseči od 0 do 20 točk. Od rešitve pričakujemo predvsem to, da je pravilna (= da predlagani postopek ali podprogram vrača pravilne rezultate), poleg tega pa je zaželeno tudi, da je učinkovita (manj učinkovite rešitve dobijo manj točk).

Če tekmovalec pri neki nalogi ni uspel sestaviti cele rešitve, pač pa je prehodil vsaj del poti do nje in so v njegovem odgovoru razvidne vsaj nekatere od idej, ki jih rešitev tiste naloge potrebuje, naj vendarle dobi delež točk, ki je približno v skladu s tem, kolikšen delež rešitve je našel.

Če v besedilu naloge ni drugače navedeno, lahko tekmovalčeva rešitev vedno predpostavi, da so vhodni podatki, s katerimi dela, podani v takšni obliki in v okviru takšnih omejitev, kot jih zagotavlja naloga. Tekmovalcem torej načeloma ni treba pisati rešitev, ki bi bile odporne na razne napake v vhodnih podatkih.

V nadaljevanju podajamo še nekaj nasvetov za ocenjevanje pri posameznih nalogah.

1. Kamen, papir, škarje

- V naših rešitvah je naloga rešena tako, da bere vhodne podatke po potezah (vsakič dva znaka) in ne dela nobenih predpostavk o tem, kako dolgo zaporedje potez imamo. Za enako dobro naj se šteje tudi rešitev, ki prebere celotno vhodno zaporedje naenkrat kot niz oz. tabelo znakov, tudi če pri tem naredi kakšno predpostavko o največji možni dolžini vhodnega niza.
- Format izpisa pri tej nalogi ni posebej določen; dovolj dober je vsak izpis, pri katerem je pač jasno razvidno, kdo je zmagovalec oz. ali je rezultat neodločen.

2. Papajščina

- V naših rešitvah je naloga rešena tako, da bere vhodne podatke znak po znak in jih sproti izpisuje. Za enako dobro naj se šteje tudi rešitev, ki bere vhodne podatke po vrsticah in shrani celo besedo v neko spremenljivko oz. tabelo (tudi če pri tem naredi kakšno predpostavko o največji možni dolžini vhodnega niza).
- Če bi rešitev zaradi neučinkovitega ravnanja z nizi porabila za predelavo besede, dolge n črk, v papajsko različico te besede $O(n^2)$ časa namesto le $O(n)$ časa, naj se ji zaradi tega točk ne odbija.

3. Obfuskator

- Besedilo naloge pravi, naj naloga besede s premešanimi črkami izpisuje sproti. Rešitvi, ki prebere vse vhodne besede v pomnilnik, preden začne izpisovati rezultate, naj se odbije tri točke.
- Vseeno je, ali rešitev uporablja funkcijo Nakljucno, podano v besedilu naloge, ali funkcije, ki jih za generiranje psevdonaključnih števil ponuja standardna knjižnica njenega programskega jezika (na primer rand v C/C++).
- Če rešitev ne pusti prvega in zadnjega znaka pri miru, naj se ji odbije štiri točke.
- Rešitev sme predpostaviti, da v vhodnih podatkih ni praznih vrstic (torej besed dolžine 0). Na besedah dolžine 1 in 2 pa mora pravilno delovati (in jih torej izpisati nespremenjene), sicer naj se ji odbije štiri točke.
- Od tekmovalca ne pričakujemo, da zna utemeljiti oz. dokazati, da njegova rešitev res lahko generira vse možne premešane oblike neke besede z enako verjetnostjo. Vseeno pa, če bi rešitev na nek občuten in sistematičen način odstopala od te enakomerne porazdelitve, naj se ji število točk primerno zmanjša.

4. Zarota

- Če ima rešitev časovno zahtevnost $O(n^2)$, lahko dobi največ 18 točk. Če ima časovno zahtevnost, večjo od $O(n^2)$, lahko dobi največ 10 točk. Rešitve s časovno zahtevnostjo pod $O(n^2)$ (npr. $O(n \log n)$, $O(n + k)$) dobijo vseh 20 točk (če so pravilne).
- Če rešitev predpostavi, da v danem zaporedju ni negativnih števil, naj se ji točk zaradi tega ne odbija.
- Ker gre za nalogo tipa „opiši postopek“, je vseeno, ali je rešitev opisana z izvorno kodo, s psevdokodo, v naravnem jeziku ali še kako drugače, samo da je opis dovolj jasen.

5. Sveče

- Če rešitev ne upošteva, da lahko sveča med trenutkom, ko jo prižgemo, in trenutkom, ko jo poskušamo ugasniti, dogori že sama od sebe (ker se ji višina zmanjša na 0), naj se ji odbije 5 točk.
- Če rešitev pomotoma predpostavi, da so sveče oštevilčene od 0 do $n - 1$ (namesto od 1 do n), naj se ji zaradi tega ne odšteva točk.
- Rešitev mora pravilno delovati tudi v robnih primerih: če poskusimo prižgati svečo, ki že gori; če poskusimo prižgati svečo, ki je že povsem dogorela (ima višino 0); če poskusimo ugasniti sveče, ko so vse že ugasnjene; ipd. Če v kakšnem od teh primerov deluje napačno, naj se ji za vsakega od njih odbije 2 točki.
- Če je časovna zahtevnost rešitve linearno (ali še kaj bolj) odvisna od časov t, ki nastopajo v vhodnih podatkih (npr. ker program sam pri sebi simulira gorenje sveč in pri tem uporablja časovne korake fiksne dolžine), lahko dobi največ 10 točk.

Težavnost nalog

Državno tekmovanje ACM v znanju računalništva poteka v treh težavnostnih skupinah (prva je najlažja, tretja pa najtežja); na tem šolskem tekmovanju pa je skupina ena sama, vendar naloge v njej pokrivajo razmeroma širok razpon zahtevnosti. Za občutek povejmo, s katero skupino državnega tekmovanja so po svoji težavnosti primerljive posamezne naloge letošnjega šolskega tekmovanja:

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM in IJS
1. Kamen, papir, škarje	lažja naloga v prvi skupini
2. Papajščina	srednje težka naloga v prvi skupini
3. Obfuskator	težja v prvi ali lahka naloga v drugi skupini
4. Zarota	težja naloga v drugi skupini
5. Sveče	lažja do srednja naloga v II. ali lahka v III. skupini

Če torej na primer nek tekmovalec reši le prvo nalogo in del druge, pri ostalih pa ne naredi (skoraj) ničesar, to še ne pomeni, da ni primeren za udeležbo na državnem tekmovanju; pač pa je najbrž pametno, če na državnem tekmovanju ne gre v drugo ali tretjo skupino, pač pa v prvo.

Podobno kot prejšnja leta si tudi letos želimo, da bi čim več tekmovalcev s šolskega tekmovanja prišlo tudi na državno tekmovanje in da bi bilo šolsko tekmovanje predvsem v pomoč tekmovalcem in mentorjem pri razmišljanju o tem, v kateri težavnostni skupini državnega tekmovanja naj kdo tekmuje.

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 pa so se rezultati izšli tako, da smo v drugi skupini izjemoma podelili tri druge nagrade, v tretji skupini pa le eno prvo, dve drugi in eno tretjo. Poleg nagrad na državnem tekmovanju v skladu s pravilnikom podeljujemo tudi zlata in srebrna priznanja. Število zlatih priznanj je omejeno na eno priznanje na vsakih 25 udeležencev šolskega tekmovanja (teh je bilo letos 273) in smo jih letos podelili devet. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot v prejšnjih letih pohvale; prejmejo jih tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil zlatega priznanja; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 20 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen srebrnih priznanj 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. Poleg zlatih in srebrnih priznanj obstajajo tudi bronasta, ta pa so dobili najboljši tekmovalci v okviru šolskih tekmovanj.

V tabelah na naslednjih straneh so prejemniki nagrad označeni z „1“, „2“ in „3“ v prvem stolpcu, prejemniki priznanj pa z „Z“ (zlato) in „S“ (srebrno).

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Sebastijan Kužner	4	SERŠ Maribor	20	17	20	18	19	94
1Z	2	Nejc Smrkolj Koželj	2	Vegova Ljubljana	19	20	17	15	17	88
1Z		Lojze Žust	1	Škof. klas. gimn. Lj.	20	20	20	9	19	88
1Z		Beno Šircelj	4	STŠ Koper	20	19	10	19	20	88
2S	5	Domen Soklič	4	TŠC Kranj, SIPŠ	19	20	5	20	20	84
3S	6	Matej Reberc	3	ŠC Ptuj, ERŠ	20	20	8	15	19	82
3S		Aljaž Jeromel	2	II. gimnazija Maribor	20	20	9	15	18	82
S	8	Filip Koprivec	2	Gimnazija Vič	16	17	8	18	20	79
S	9	Rok Ljalič	3	Vegova Ljubljana	15	20	6	20	17	78
S	10	Juš Debelak	4	TŠC Kranj, Str. gim.	20	12	12	17	15	76
S		Aljaž Frančič	4	II. gimnazija Maribor	15	20	9	12	20	76
S	12	Andrej Muhič	3	ŠC Novo mesto	15	20	5	15	19	74
S	13	Gregor Miklošič	4	II. gimnazija Maribor	18	19	3	18	15	73
S	14	Rok Lekše	2	Šk. kl. g. Lj. + ZRI	18	19	10	5	19	71
S		Žiga Šmelcer	2	Škof. klas. gimn. Lj.	12	17	5	18	19	71
S	16	Žiga Simončič	2	Vegova Ljubljana	20	20	5	5	18	68
S		Gašper Pustinek	4	TŠC Kranj, SIPŠ	18	17	3	15	15	68
S	18	Timi Lah	4	SERŠ Maribor	17	20	2	12	15	66
S	19	Aljaž Borštnik	3	Gimnazija Vič	18	10	18	0	19	65
S	20	Alenka Bahovec	4	Škof. klas. gimn. Lj.	17	20	6	0	20	63
S	21	Ines Meršak	3	Gimnazija Vič	20	15	8	3	16	62
S		Marko Lavrinec	3	TŠC Kranj, Str. gim.	20	15	8	0	19	62
S	23	Peter Fajdiga	4	TŠC Kranj, Str. gim.	18	20	5	15	3	61
S	24	Alec Smrekar	3	Gimnazija Piran	20	17	8	3	11	59
S	25	Metod Medja	1	TŠC Kranj, Str. gim.	20	10	7	3	18	58
S		Vid Pavše	2	ŠC Ravne na Koroškem	16	17	2	15	8	58
S	27	Jakob Košir	3	ŠC Novo mesto	20	17	8	10	2	57
S		Toni Kocjan Turk	2	ŠC Novo mesto	18	17	6	1	15	57
S	29	Jan Perme	2	Gimnazija Vič	12	16	1	17	10	56
S	30	Aleksander Rajhard	2	Gimnazija Škofja Loka	19	12	7	0	17	55
S	31	Vid Štrancar	1	SŠ V. Pilon Ajdovščina	15	10	1	12	16	54
S	32	Domen Kosmač	4	Gimnazija Šentvid	20	5	7	0	17	49
S	33	Tobias Mihelčič	2	Vegova Ljubljana	15	0	9	14	10	48
S		Sandi Režonja	1	Gim. Murska Sobota	15	11	2	20	0	48
S		Marko Grešak	3	ŠC Novo mesto	15	16	8	5	4	48
S		Miha Štravs	1	ZRI	17	12	6	12	1	48
S	37	Žan Pevec	3	ŠC Celje, Gim. Lava	20	18	7	0	2	47
S		Rok Jelovšek	4	ŠC Celje, Gim. Lava	20	9	7	2	9	47
S		Rok Poje	3	Vegova Ljubljana	20	16	2	0	9	47

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
S	40	Luka Pogačnik	1	Gimnazija Vič	20	9	2	5	10	46
S		Žiga Franko	2	ŠC Novo mesto	17	20	4	0	5	46
S		Marko Hrup	4	STŠ Koper	17	17	7	5	0	46
S		Žiga Vodušek Resnik	3	ŠC Celje, Gim. Lava	20	15	6	5	0	46
S	44	Alen Verk	3	ŠC Celje, Šola za KER	20	13	11	0	0	44
S	45	Andrej Orehek	4	SŠJJ Ivančna Gorica	18	13	10	0	2	43
S	46	Klemen Plazar	4	ERŠ Velenje	0	14	8	14	6	42
S		Matej Vovko	2	ŠC Novo mesto	17	12	5	8	0	42
S		Ivan Kolundžija	2	Gimnazija Vič	7	15	13	7	0	42
S	49	Aleš Papič	3	ŠC Celje, Šola za KER	20	12	8	1	0	41
S		Gregor Šturm	3	TŠC Kranj, SIPŠ	20	10	4	0	7	41
S		Janez Kuhar	4	Gimnazija Litija	7	7	8	18	1	41
	52	Igor Đukanović	3	ERŠ Velenje	3	2	14	17	4	40
		Jaka Kordež	1	TŠC Kranj, SIPŠ	19	0	6	5	10	40
	54	Matej Lubej	4	SERŠ Maribor	19	15	4	0	1	39
		Rok Novosel	3	ŠC Novo mesto	18	9	7	5	0	39
	56	Luka Kolar	1	Gimnazija Vič	20	5	8	5	0	38
		Rok Kos	9	ZRI	0	14	9	15	0	38
	58	Janez Štular	3	TŠC Kranj, SIPŠ	18	9	9	0	1	37
		Andrej Čuber	3	Gimnazija Piran	10	11	6	8	2	37
		Žan Žerdin Furlan	1	ZRI	12	20	5	0	0	37
		Leon Stanko	4	ŠC Celje, Gim. Lava	15	0	8	10	4	37
	62	Rok Lesjak	2	ERŠ Velenje	17	12	6	0	1	36
		Žan Ivanovič	2	Vegova Ljubljana	16	9	9	1	1	36
	64	Tina Lekše	9	ZRI	12	10	3	8	2	35
		Urban Lavbič	3	ŠC Celje, Šola za KER	18	11	6	0	0	35
	66	Medard Švigelj	2	Škof. klas. gimn. Lj.	15	5	2	0	12	34
		Andraž Papež	3	SŠJJ Ivančna Gorica	17	0	5	2	10	34
		Simon Weiss	2	ZRI	0	12	3	18	1	34
		Matevž Poljanc	1	Škof. klas. gimn. Lj.	14	0	3	16	1	34
	70	Žiga Kokelj	2	Gimnazija Škofja Loka	7	17	5	0	4	33
	71	Jan Golob	2	Gimnazija Vič	9	3	15	0	5	32
		Vid Leskovar	2	II. gimnazija Maribor	12	12	4	0	4	32
		Tilen Pugelj	4	Gimnazija Vič	0	9	7	12	4	32
	74	Rok Janež	3	ŠC Novo mesto	20	5	5	0	1	31
		Tadej Miklavčič	3	ŠC Novo mesto	4	20	6	1	0	31
		Pavle Iliev	3	ŠC Ptuj, ERŠ	10	3	5	13	0	31
		Matic Korošec	2	ERŠ Velenje	15	4	7	0	5	31
	78	Matic Zagmajster	2	Vegova Ljubljana	15	0	5	7	2	29
		Andrej Rokavec	4	SERŠ Maribor	7	13	8	0	1	29
		Mark Lukek	2	Gim. J. Plečnika Lj.	15	12	1	1	0	29

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
	81	Damjan Časar	2	SPTŠ Murska Sobota	14	4	4	5	0	27
	82	Željko Krčmar	3	ŠC Ptuj ERŠ	0	10	6	5	5	26
	83	Rok Hribar	2	ERŠ Velenje	12	6	5	0	1	24
		Jaka Mohorko	1	II. gimnazija Maribor	9	5	2	1	7	24
	85	Nives Bogataj	2	Gimnazija Vič	0	7	2	10	3	22
	86	Martin Oprešnik	3	ŠC Celje, Šola za KER	17	0	3	0	0	20
	87	Luka Prebil Grintar	2	Gimnazija Vič	0	10	6	0	3	19
		Gašper Jelovčan	1	Škof. klas. gimn. Lj.	0	2	10	2	5	19
		Blaž Velkavrh	2	Gimnazija Vič	7	0	9	1	2	19
	90	Miha Majetič	1	Gimnazija Vič	7	2	2	1	5	17
	91	Gregor Ivajnšič	2	SPTŠ Murska Sobota	0	10	5	0	0	15
		Tilen Tomakič	2	TŠC Kranj, SIPŠ	15	0	0	0	0	15
	93	Primož Mihelič	1	Gimnazija Škofja Loka	10	0	2	0	0	12
	94	Darko Kušer	2	ERŠ Velenje	5	1	3	0	1	10
		Luka Toni	4	TŠC Kranj, Str. gim.	0	3	1	3	3	10
		Vid Jožef Humer	2	Gimnazija Poljane	7	2	1	0	0	10
	97	Kristjan Banfi	2	SPTŠ Murska Sobota	0	5	1	0	1	7
		Jure Malovrh	4	Gimnazija Poljane	0	0	7	0	0	7
	99	Rok Šeško	1	II. gimnazija Maribor	3	0	3	0	0	6
		Jan Hajnrihar	1	Gimnazija Škofja Loka	0	0	6	0	0	6
	101	Tilen Vivod	2	ERŠ Velenje	0	0	1	0	1	2
	102	Jan Sušnik	2	TŠC Kranj, SIPŠ	0	0	1	0	0	1
		Tilen Košak	3	Gimnazija Piran	0	0	1	0	0	1

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Žiga Gradišar	2	Šk. kl. g. Lj. + ZRI	20	17	20	18	20	95
1Z	2	Erik Langerholc	3	Gimnazija Bežigrad	18	14	20	19	20	91
2Z	3	Fedja Beader	4	Vegova Ljubljana	15	15	20	20	20	90
2S	4	Luka Horvat	4	ŠC Ptuj, ERŠ	12	18	16	20	20	86
3S	5	Jakob Merljak	2	Gimnazija Bežigrad	20	16	9	20	18	83
3S	6	Urban Škvorc	4	Gimnazija Bežigrad	17	20	13	18	13	81
S	7	Tadej Škvorc	4	Gimnazija Bežigrad	20	14	18	16	12	80
S	8	Marko Novak	3	ZRI	0	20	20	20	19	79
S	9	Peter Filip Lebar	2	Gimnazija Vič	20	20	7	10	20	77
S	10	Sašo Stanovnik	4	Gimnazija Bežigrad	5	16	17	19	15	72
S	11	Blaž Blokar	3	TŠC N. Gorica, Teh. gim.	10	16	13	12	15	66
S	12	Matej Mohar	3	TŠC N. Gorica, Teh. gim.	18	1	18	20	5	62
S	13	Matej Horvat	2	Gimnazija Moste	0	18	9	14	20	61
S	14	Jaka Konda	3	Vegova Ljubljana	13	19	5	18	5	60
S	15	Žan Kusterle	3	Vegova Ljubljana	5	13	10	20	10	58
S	16	Michel Adamič	3	Gimnazija Bežigrad	0	18	20	18	0	56
	17	Klemen Pevec	9	OŠ Veliki Gaber	15	14	5	14	5	53
		Gašper Kolar	3	Vegova Ljubljana	3	15	6	19	10	53
	19	Jan Haložan	4	ŠC Ptuj, ERŠ	2	3	4	20	19	48
	20	Dejan Paradiž	4	ŠC Ravne na Koroškem	15	12	6	14	0	47
		Klemen Kozjek	4	Vegova Ljubljana	2	12	19	4	10	47
	22	Jani Golob	4	TŠC N. Gorica, Teh. gim.	0	13	13	20	0	46
	23	Sebastian Čolić	2	Gimnazija Moste	0	14	6	4	10	34
	24	Jan Živkovič	3	Vegova Ljubljana	0	12	8	10	1	31
		Tadej Petreski	4	II. gimnazija Maribor	0	5	8	8	10	31
	26	Matic Slemič	3	TŠC N. Gorica, Teh. gim.	0	11	8	5	5	29
	27	Miha Černetič		SŠ Veno Pilon Ajdovščina	20	5	3	0	0	28
	28	Gregor Mrak	3	TŠC N. Gorica, Teh. gim.	0	11	6	10	0	27
	29	Patrik Kokol	4	ŠC Ptuj, ERŠ	2	0	5	10	2	19
	30	Sašo Marković	4	II. gimnazija Maribor	0	12	0	0	0	12
	31	Tadej Štadler	4	ERŠ Velenje	0	1	0	8	0	9
	32	Aljaž Razpotnik	3	ŠC Ravne na Koroškem	0	5	0	0	0	5
		Domen Mori	4	ŠC Ravne na Koroškem	0	5	0	0	0	5

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Matjaž Leonardis	4	ZRI	100	100	49	97	78	424
1Z	2	Patrik Zajec	2	ZRI	100	100		20	60	280
2S	3	Jure Slak	4	Gimnazija Vič in ZRI	100	37	55		57	249
2S	4	Maks Kolman	3	Gimnazija Vič	100	40	27	20	60	247
3S	5	Rok Kaufman	4	ZRI	67	40	50	20	37	214
3S	6	Vid Kocijan	2	ZRI	91	37	35		24	187
S	7	Gašper Medved	4	Vegova Ljubljana	100				64	164
S	8	Andraž Dobnikar	4	Vegova Ljubljana	100	40				140
S	9	David Gričar	3	Gimnazija Moste	64	40			27	131
	10	Žiga Gosar	4	Gimnazija Vič	30		50		44	124
	11	Tibor Djurica Potpara	4	Gimnazija Poljane	67	40				107
		Sven Cerk	4	ZRI	100	7				107
	13	Jan Aleksandrov	2	ZRI		37			28	65
	14	Živa Urbančič	2	ZRI	0	40			0	40
		Tadej Ciglarič	3	Gimnazija Bežigrad		40				40
	16	Jasna Urbančič	4	ZRI		37				37
	17	Gregor Gololičič	4	TŠC N. Gorica, Teh. gim.		4				4
	18	Gaj Žižek	3	Gim. Šentvid in ZRI	0	0				0

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Sebastijan Kužner	90 GB SSD disk
1	1	Nejc Smrkolj Koželj	90 GB SSD disk
1	1	Lojze Žust	60 GB SSD disk
1	1	Beno Šircelj	miška Logitech G700
1	2	Domen Soklič	slušalke Gamecom 777
1	3	Matej Reberc	64 GB USB flash disk
1	3	Aljaž Jeromel	naročnina na Monitor z DVDjem
2	1	Žiga Gradišar	90 GB SSD disk Dasgupta *et al.: Algorithms*
2	1	Erik Langerholc	60 GB SSD disk Dasgupta *et al.: Algorithms*
2	2	Fedja Beader	64 GB USB flash disk Dasgupta *et al.: Algorithms*
2	2	Luka Horvat	60 GB SSD disk
2	3	Jakob Merljak	miška Logitech G700
2	3	Urban Škvorc	naročnina na Monitor z DVDjem
3	1	Matjaž Leonardis	Amazon Kindle Knuth: *The Art of Computer Programming*, Vol. 4A
3	1	Patrik Zajec	60 GB SSD disk Cormen *et al.: Introduction to algorithms*
3	2	Jure Slak	64 GB USB flash disk Cormen *et al.: Introduction to algorithms*
3	2	Maks Kolman	60 GB SSD disk
3	3	Rok Kaufman	miška Logitech G700
3	3	Vid Kocijan	64 GB USB flash disk
Tekmovanje programov — Kamen, papir, škarje			
	1	Žiga Gosar	magnetne kroglice NeoCube
	2	Jure Slak	magnetne kroglice NeoCube

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

Druga gimnazija Maribor	Mirko Pešec
Gimnazija Bežigrad	Andrej Šuštaršič, Siniša Ribić
Gimnazija Jožeta Plečnika Ljubljana	Tomi Zebič
Gimnazija Litija	Jan Maver
Gimnazija Moste	Gorazd Kovačič, Aleš Razinger
Gimnazija Murska Sobota	Romana Vogrinčič
Gimnazija Piran	Janez Urevc
Gimnazija Poljane	Janez Malovrh
Gimnazija Šentvid	Klemen Blokar, Nastja Lasič
Gimnazija Škofja Loka	Anže Nunar
Gimnazija Vič	Gašper Ažman, Klemen Bajec, Natan Žabkar
OŠ Veliki Gaber	Matjaž Lavrih
Srednja elektro-računalniška šola Maribor (SERŠ)	Vida Motaln, Slavko Nekrep
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Karel Maček, Boris Ribaš
Srednja šola Josipa Jurčiča Ivančna Gorica	Darko Pandur
Srednja šola Veno Pilon Ajdovščina	Matej Bolko
Srednja tehniška šola (STŠ) Koper	Andrej Florjančič
Škofijska klasična gimnazija Šentvid	Helena Medvešek
Šolski center Celje, Gimnazija Lava	Karmen Kotnik, Tomislav Viher
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Dušan Fugina
Šolski center Novo mesto	Ivan Slinkar, Simon Vovko
Šolski center Ptuj, Elektro in računalniška šola	Zoltan Sep, Franc Vrbančič
Šolski center Ravne na Koroškem, Srednja šola Ravne	Gorazd Geč, Zdravko Pavleković
Šolski center Velenje, Elektro in računalniška šola	Nedeljko Grabant, Miran Zevnik

Tehniški šolski center Kranj, Strokovna in poklicna šola (SIPŠ)	Aleš Hvasti
Tehniški šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Tehniški šolski center Nova Gorica, Tehniška gimnazija	Tomaž Mauri, Barbara Pušnar, Boštjan Vouk
Vegova Ljubljana	Marko Kastelic, Matjaž Kodela, Nataša Makarovič, Darjan Toth
Zavod za računalniško izobraževanje (ZRI), Ljubljana	Jelko Urbančič, Jan Berčič, Žiga Ham

TEKMOVANJE PROGRAMOV — KAMEN, PAPIR, ŠKARJE

Podobno kot v prejšnjih letih smo tudi letos organizirali tekmovanje programov. Opis naloge smo objavili septembra 2011 skupaj z razpisom za tekmovanje v znanju, tekmovalci pa so imeli čas do 21. marca 2012 (tri dni pred tekmovanjem), da pošljejo svoje programe. Letošnja naloga je od tekmovalcev zahtevala, da napišejo logiko za igranje igre „kamen, papir, škarje“.

Opis naloge

„Kamen, papir, škarje“ je igra za dva igralca. Sestavljena je iz zaporedja rund. V vsaki rundi vsak od igralcev izbere enega od treh predmetov (kamen, papir ali škarje), ne da bi vedel, kaj v tej rundi izbira drugi igralec. Če oba izbereta enak predmet, je runda neodločena. Če pa izbereta različna predmeta, je eden od njiju rundo dobil, drugi pa izgubil (kamen premaga škarje, škarje premagajo papir, papir premaga kamen).

Vsak tekmovalec napiše en podprogram za krmiljenje igralca; ta podprogram dobi pred vsako rundo tudi podatke o tem, kakšen je trenutni rezultat igre in kaj je bila zadnja nasprotnikova poteza.

Rezultati

Letos smo pri tekmovanju programov prejeli rešitve osmih tekmovalcev (vsi so bili dijaki). Prejete programe smo preizkusili tako, da smo za vsak par igralcev izvedli 106 400 rund; zmagovalec runde dobi dve točki, poraženec nobene, pri neodločeni rundi pa dobi vsak od igralcev po eno točko. Spodnja tabela prikazuje skupno število točk posameznega tekmovalca po vseh rundah:

Mesto	Ime	Šola	Točke
1	Žiga Gosar	Gim. Vič	1057843
2	Jure Slak	Gim. Vič	986474
3	Beno Šircelj	STŠ Koper	745801
	Vito Meznarič	ŠC Ptuj, ERŠ	744549
	Luka Horvat	ŠC Ptuj, ERŠ	744303
6	Sebastijan Kužner	SERŠ Maribor	729448
7	Klemen Kozjek	Vegova Ljubljana	526046
8	Gašper Medved	Vegova Ljubljana	423992

Tretje mesto si delijo trije tekmovalci, ker uporabljajo vsi trije enako strategijo in je razlika v točkah le plod naključja: njihovi programi namreč vlečejo popolnoma naključne poteze in se ne ozirajo na to, kaj počne nasprotnik.

Rešitev

Očitna in zelo vabljiva strategija pri tej igri je, da vlečemo popolnoma naključne poteze; torej z verjetnostjo 1/3 izberemo kamen, z verjetnostjo 1/3 papir in z verjetnostjo 1/3 škarje. Ne glede na to, kaj v isti potezi naredi nasprotnik, imamo torej 1/3 možnosti, da to rundo dobimo, 1/3 možnosti, da jo izgubimo, in 1/3 možnosti, da bo runda izenačena. Tako bomo v povprečju proti vsakemu nasprotniku igrali izenačeno, ne glede na to, kaj počne ta nasprotnik (razen če bi uspel nasprotnik nekako ugotoviti, kako deluje naš generator psevdonaključnih števil in s katerim začetnim

semenom smo ga pognali — to ni preveč realistično; če pa bi mu to uspelo, bi lahko napovedoval naše prihodnje poteze in nas tako izigral). Lepo pri tej strategiji je, da je tudi zelo preprosta in jo lahko implementiramo že z eno samo vrstico izvorne kode.

Če se pomerita dva igralca, od katerih eden igra naključno, bo torej izid v povprečju približno izenačen (ne glede na to, ali tudi drugi igralec igra naključno ali ne). Če pa se pomerita dva igralca, od katerih nobeden ne igra naključno, potem ni nujno, da bo rezultat izenačen; prav mogoče je, da bo eden od njiju dobil občutno več točk kot drugi. Tako torej vidimo, da v skupini več tekmovalcev naključna strategija ne bo nujno nauspešnejša: naključni igralec bo izenačil proti vsem drugim (tako naključnim kot nenaključnim); nenaključni igralec pa bo izenačil proti naključnim, proti nenaključnim pa bo mogoče zmagoval, mogoče pa izgubljal, odvisno od tega, ali je njegova strategija boljša od njihove ali ne. Do tega pojava je na našem tekmovanju tudi v resnici prišlo: trije od osmih tekmovalcev so poslali naključno strategijo in si delijo tretje mesto; med ostalimi tekmovalci pa sta bila dva boljša od naključnih, trije pa slabši.

Zato se vendarle splača razmisliti tudi o nenaključni strategiji, saj nam šele ta daje možnost, da bomo v skupnem seštevku boljši od naključnih igralcev (seveda le v primeru, če poleg nas obstaja še kakšen nenaključni igralec in če ga mi uspemo premagati; če pa vsi naši tekmeci igrajo naključno, bomo izenačeni skupaj z njimi ne glede na to, ali mi igramo naključno ali ne).

Zmagovalni program je poskušal predvidevati nasprotnikove poteze tako, da je vzdrževal tri števce (recimo jim n_0, n_1 in n_2), ki povedo, kolikokrat je nasprotnik doslej pokazal kateri predmet, pri čemer imajo podatki iz zgodnejših rund manjšo težo kot tisti iz kasnejših. Ko po posamezni rundi izvemo, kateri predmet je pokazal nasprotnik, popravimo števce: $n_k := \lambda n_k + \triangle_k$, pri čemer je $\triangle_k = 1$, če je nasprotnik v prejšnji rundi pokazal predmet k, sicer pa je $\triangle_k = 0$. Množenje s konstanto λ poskrbi, da vpliv posamezne runde pada eksponentno s časom (v zmagovalni rešitvi je imela konstanta λ vrednost 0,9). Program je nato skušal napovedati nasprotnikovo naslednjo potezo tako, da je z verjetnostjo $n_k/(n_0 + n_1 + n_2)$ napovedal potezo k (in potem za svojo potezo izbral tisti predmet, ki bi proti potezi k zmagal).[16]

[16] Še ena možnost je, da bi se naključnosti pri tej napovedi izognili in za nasprotnikovo potezo napovedali tisti k, ki ima največjo vrednost n_k, kar pa je mogoče bolj tvegano, saj bi bile v tem primeru naše poteze za nasprotnika bolj predvidljive.

UNIVERZITETNI PROGRAMERSKI MARATON

Društvo ACM Slovenija sodeluje tudi pri pripravi študenskih tekmovanj v programiranju, ki v zadnjih letih potekajo pod imenom Univerzitetni programerski maraton (UPM, `www.upm.si`) in so odskočna deska za udeležbo na ACMovih mednarodnih študentskih tekmovanjih v programiranju (International Collegiate Programming Contest, ICPC). Ker UPM ne izdaja samostojnega biltena, bomo na tem mestu na kratko predstavili to tekmovanje in njegove letošnje rezultate.

Na študentskih tekmovanjih ACM v programiranju tekmovalci ne nastopajo kot posamezniki, pač pa kot ekipe, ki jih sestavljajo po največ trije člani. Vsaka ekipa ima med tekmovanjem na voljo samo en računalnik. Naloge so podobne tistim iz tretje skupine našega srednješolskega tekmovanja, le da so včasih malo težje oz. predvsem predpostavljajo, da imajo reševalci že nekaj več znanja matematike in algoritmov, ker so to stvari, ki so jih večinoma slišali v prvem letu ali dveh študija. Časa za tekmovanje je pet ur, nalog pa je praviloma 6 do 8, kar je več, kot jih je običajna ekipa zmožna v tem času rešiti. Za razliko od našega srednješolskega tekmovanja pri študentskem tekmovanju niso priznane delno rešene naloge; naloga velja za rešeno šele, če program pravilno reši vse njene testne primere. Ekipe se razvrsti po številu rešenih nalog, če pa jih ima več enako število rešenih nalog, se jih razvrsti po času oddaje. Za vsako uspešno rešeno nalogo se šteje čas od začetka tekmovanja do uspešne oddaje pri tej nalogi, prišteje pa se še po 20 minut za vsako neuspešno oddajo pri tej nalogi. Tako dobljeni časi se seštejejo po vseh uspešno rešenih nalogah in ekipe z istim številom rešenih nalog se potem razvrsti po skupnem času (manjši ko je skupni čas, boljša je uvrstitev).

UPM poteka v štirih krogih (dva spomladi in dva jeseni), pri čemer se za končno razvrstitev pri vsaki ekipi zavrže najslabši rezultat iz prvih treh krogov, četrti (finalni) krog pa se šteje dvojno. Najboljše ekipe se uvrstijo na srednjeevropsko regijsko tekmovanje (CERC, tokrat v Krakowu), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki je bilo tokrat v Sankt Petersburgu v Rusiji).

Na letošnjem UPM je sodelovalo 57 ekip s skupno 154 tekmovalci, ki so prišli z vseh treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednjih dveh straneh prikazuje ekipe, ki so rešile vsaj eno nalogo.

	Ekipa	Št. rešenih nalog*	Čas
1	Jan Berčič (FMF), Klemen Kloboves (FRI), Matjaž Leonardis (I. gim. Celje)	17	17:44:18
2	Žiga Ham (FRI), Matej Aleksandrov, Nace Hudobivnik (FMF)	17	20:09:12
3	Jure Slak, Maks Kolman, Žiga Gosar (Gim. Vič / FMF)	16	17:38:46
4	Tim Kos, Matej Kren, Aleksander Kelenc (FNM Maribor)	13	21:16:17
5	Aleksandar Tošić, Simon Mezgec, Marko Grgurovič (FAMNIT)	11	20:56:48
6	Marko Čavdek, Andraž Bajt, Aljoša Mrak (FRI)	9	8:48:17
7	Ernest Beličič, Aleš Razpotnik, Jure Kolenko (FRI)	9	8:53:24
8	Žiga Zalokar, Andraž Dobnikar, Tibor Djurica Potpara (Vegova Lj.)	9	10:16:04
9	Miha Zidar, Gregor Majcen (FRI)	9	10:48:28
10	Jasna Urbančič, Rok Kaufman, Vid Kocijan (ZRI)	9	10:58:06
11	Primož Godec, Luka Krsnik, Manca Žerovnik (FRI)	9	12:59:24
12	Dragana Božović, Martin Frešer, Martin Duh (FNM Maribor)	8	8:19:30
13	Tomaž Stepišnik Perdih, Eva Breznik (FMF), Matic Kladnik (FRI)	7	6:58:05
14	Duško Topić, Vida Maksimović, Nastja Cepak (FAMNIT)	7	8:18:23
15	Jon Premik, Igor Lalić (FRI)	7	9:42:38
16	Žiga Šmelcer, Žiga Gradišar, Lojze Žust (Škof. klas. gim. Lj.)	7	12:35:06
17	Luka Firm, Luka Jeran, Marko Tavčar (FRI)	7	16:16:08
18	Tomaž Kariž, Primož Kariž, Domen Mladovan (FRI)	6	6:37:46
19	Aleksandar Todorović, Marko Tavčar (FAMNIT)	6	7:24:39
20	Beno Šircelj, Luka Hrvatin (STŠ Koper), Uroš Šavelj (FAMNIT)	6	8:28:07
21	Erik Grabljevec, Blaž Sobočan (FMF), Silvester Jakša (FRI)	6	12:20:43
22	Luka Černe, Filip Kozarski, Matej Petković (FMF)	4	7:28:43
23	Andrej Koželj, Žan Križanovski, Eva Janša (FMF)	4	9:18:48
24	Rok Kralj (FRI)	4	9:44:21
25	Boštjan Lasnik, David Možina, Nejc Škerjanc (FRI)	4	10:46:16
26	Matej Skrbiš (FERI)	3	1:32:10
27	Jure Senegačnik (F. za strojništvo), Tanja Gomilar, Žiga Emeršič (FRI)	3	3:04:56
28	Jurij Slabanja, Mark Hočevar (FRI)	3	5:48:21
29	Gašper Medved, Klemen Kozjek, Jaka Konda (Vegova Lj.)	3	7:06:21
30	Marko Bizjak, Boris Vezenšek, Rok Bračun (FERI)	3	7:06:34
31	Nejc Štebe, Milutin Spasić, Iztok Oder (FRI)	3	8:22:05
32	Jernej Strasner, Aleš Horvat, Jani Zajc (FAMNIT)	3	8:25:24
33	Tjaž Brelih, Jani Bevk (FRI), Rok Krhlikar (FE)	3	10:56:35

* Opomba: naloge z najslabšega od prvih treh krogov se ne štejejo, naloge z zadnjega kroga pa se štejejo dvojno. Enako je tudi pri času, le da se čas zadnjega kroga ne šteje dvojno.

(*nadaljevanje na naslednji strani*)

	Ekipa	Št. rešenih nalog*	Čas
34	Sandi Majninger (FERI)	2	2:00:03
35	Matjaž Hegedič, Anže Nunar, Peter Žužek (FRI)	2	2:00:25
36	Matej Filipović, Tomaž Tomažinčič, Rudi Kovač (FAMNIT)	2	2:11:13
37	Jure Grabnar, David Starič, Matej Zrimšek (FRI)	2	2:20:29
38	Tadej Golobič, Rok Černič, Petra Prešeren (FRI)	2	3:02:02
39	Tadej Vodopivec, Andrej Tratnik (FRI), Martin Mikeln (FE)	2	3:48:24
40	Filip Koprivec, Filip Peter Lebar, Jan Perme (Gimnazija Vič)	2	4:24:59
41	Matic Potočnik, Anže Pečar (FRI)	2	5:06:14
42	Aleš Omerzel, Domen Urh (FRI), Neža Žager Korenjak (FMF)	2	5:26:14
43	Marko Ljubotina, Mitja Zidar, Pavel Kos (FMF)	2	9:28:57
44	Tomaž Šuen, Rok Šket, Robert Pajek (FERI)	1	0:29:53
45	Jurij Podgoršek (FERI)	1	1:20:39
46	KTM (Koper)	1	1:24:40
47	Denis Subotic, Tadej Magajna, Dean Cerin (FAMNIT)	1	1:41:28
48	Jernej Hartman, Boštjan Cigan (FRI)	1	1:42:30
49	Andrej Dolenc, Nejc Maleš, Dore Pavlič (FERI)	1	3:35:18
50	Marko Očko (FERI)	1	3:36:39
51	Gregor Časar, Aleksandar Bobič, Matic Plešec (FRI)	1	4:17:38

* Opomba: naloge z najslabšega od prvih treh krogov se ne štejejo, naloge z zadnjega kroga pa se štejejo dvojno. Enako je tudi pri času, le da se čas zadnjega kroga ne šteje dvojno.

Na srednjeevropskem tekmovanju so nastopile ekipe 2, 3 in 6 kot predstavnice Univerze v Ljubljani, 4 in 12 kot predstavnici Univerze v Mariboru in kombinacija ekip 5 + 19 kot predstavnica Univerze na Primorskem. V konkurenci 77 ekip z 31 univerz iz 6 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
33	Žiga Ham, Matej Aleksandrov, Nace Hudobivnik	4	7:12
34	Žiga Gosar, Jure Slak, Maks Kolman	4	7:24
49	Aleksander Kelenc, Tim Kos, Matej Kren	2	2:03
55	Martin Frešer, Martin Duh, Dragana Božović	2	4:20
57	Aljoša Mrak, Marko Čavdek, Andraž Bajt	2	5:24
63	Aleksandar Tošić, Marko Tavčar, Aleksandar Todorović	2	7:18

Na srednjeevropskem tekmovanju je bilo 11 nalog, od tega jih je zmagovalna ekipa rešila deset.

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 153–159.

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(a)? ☐ sam(a) ☐ 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++, pythonu in javi.
— 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 C-ju.
— Ali razumeš C 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 1. in 2. skupini.] Kakšno je tvoje mnenje o sistemu za oddajanje odgovorov prek računalnika? _____

[Le pri 3. skupini.] Letos v tretji skupini podpiramo 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++/C#/javi: `&`, `|`, `^`, `~`)	☐	☐	☐
Operatorja **shl** in **shr** (v C/C++/C#/javi: `<<`, `>>`)	☐	☐	☐
Znaš uporabiti kakšnega od naslednjih razredov iz standardnih knjižnic:			
hash_map, hash_set, unordered_map, unordered_set (v C++), Hashtable, HashSet (v javi/C#), Dictionary (v C#)	☐	☐	☐
map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#)	☐	☐	☐
priority_queue (v C++), PriorityQueue (v javi)	☐	☐	☐

[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(a)?
- ☐ nisem rešil(a), ker mi je zmanjkalo časa za reševanje
- ☐ nisem rešil(a), ker mi je zmanjkalo volje za reševanje
- ☐ nisem rešil(a), ker mi je zmanjkalo znanja za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo časa za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo volje za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo znanja za reševanje
- ☐ rešil(a) 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(a) raje: ☐ več časa ☐ manj časa ☐ časa je bilo ravno prav
Bi imel(a) raje: ☐ več nalog ☐ manj nalog ☐ nalog je bilo ravno prav

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

Kaj ti je bilo pri tekmovanju všeč? _____

Kaj te je najbolj motilo? _____

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

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

Udeležil bi se oz. z veseljem bi spremljal:

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

Vesel(a) 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/la do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 98 tekmovalcev prve skupine, 26 tekmovalcev druge skupine in 16 tekmovalcev tretje skupine. Vprašanja so bila pri letošnji anketi približno enaka kot lani.

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. 154. 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, vendar malo lažje kot prejšnja leta. 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: 3,39 v prvi skupini (v prejšnjih letih 3,56, 3,34, 3,56), 3,50 v drugi skupini (prejšnja leta 3,39, 3,38, 3,46) in 3,21 v tretji skupini (lani 3,57, predpredlani 3,92).

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 bila ponavadi šibka negativna korelacija, ki pa je v zadnjih letih skorajda povsem izginila ($R^2 = 0{,}21$, lani 0,11, prejšnja leta okoli 0,4).

Največ pripomb o tem, kako da je naloga težka, je bilo pri nalogah 1.4 (mase), 1.5 (v Afganistan!), 2.3 (razpolovišče lika) in 3.3 (leteči pujsi). Pri prvih dveh si lahko mnenje, da sta težki, mogoče razložimo s tem, da sta nalogi bolj nestandardnega tipa, pri 2.3 pa s tem, da so jo ljudje dojemali kot geometrijsko nalogo.

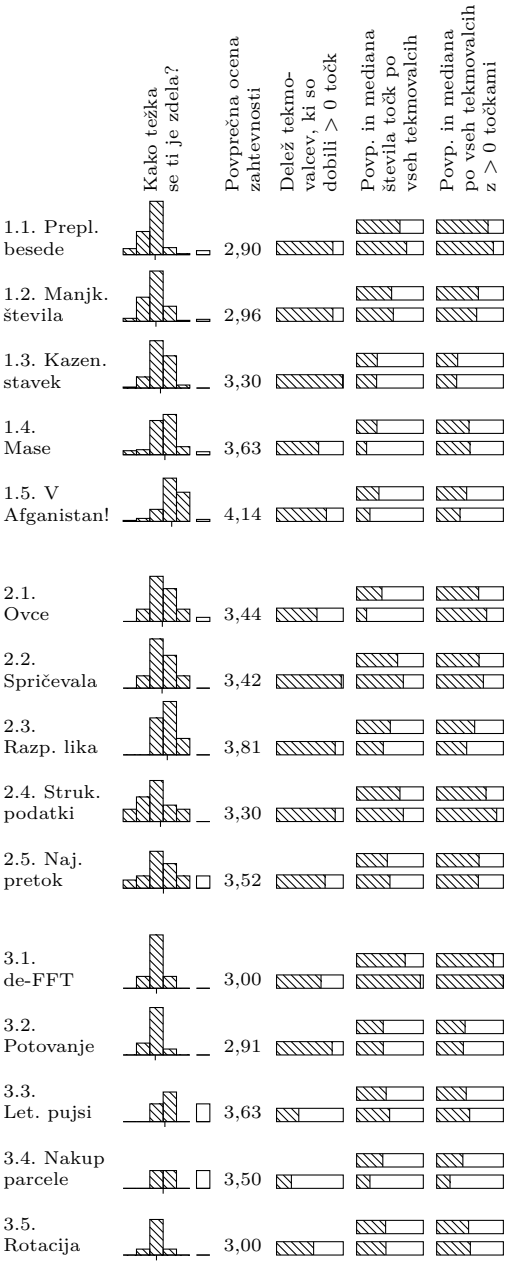
Kot najlažjo so tekmovalci ocenili nalogo 1.1 (prepletene besede), pri kateri je bilo celo tudi nekaj pripomb, da je prelahka.

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 155. Nad razumljivostjo besedil ni veliko pripomb (podobno kot lani in še malo manj kot prejšnja leta); daleč najtežje razumljiva se jim je zdela naloga 1.5 (v Afganistan!), deloma pa tudi 1.4 (mase) in 2.5 (največji pretok). To je verjetno delno krivda dejstva, da te naloge govorijo o stvareh, ki tekmovalcem niso preveč domače (1.5 je naloga o grafih, 2.5 pa je realnočasovna).

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, še bolj kot prejšnja leta. Še največ pripomb na dolžino je pri nalogi 1.5 (v Afganistan!), vendar v obe smeri: nekaterim se je zdelo besedilo prekratko, nekaterim pa predolgo.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta. Več pripomb glede dolgočasnosti nalog je bilo v tretji skupini, še posebej pri nalogah 3.2 (potovanje) in 3.4 (nakup parcele). Pri nalogi 1.1 (prepletene besede) je bilo tudi nekaj pripomb, češ da je že znana.

Pripomb, da bi naloga vzela preveč časa, je bilo približno toliko kot lani. Največ takih pripomb je bilo pri nalogah 1.5 (v Afganistan!), 3.1 (de-FFT permutacija) in



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

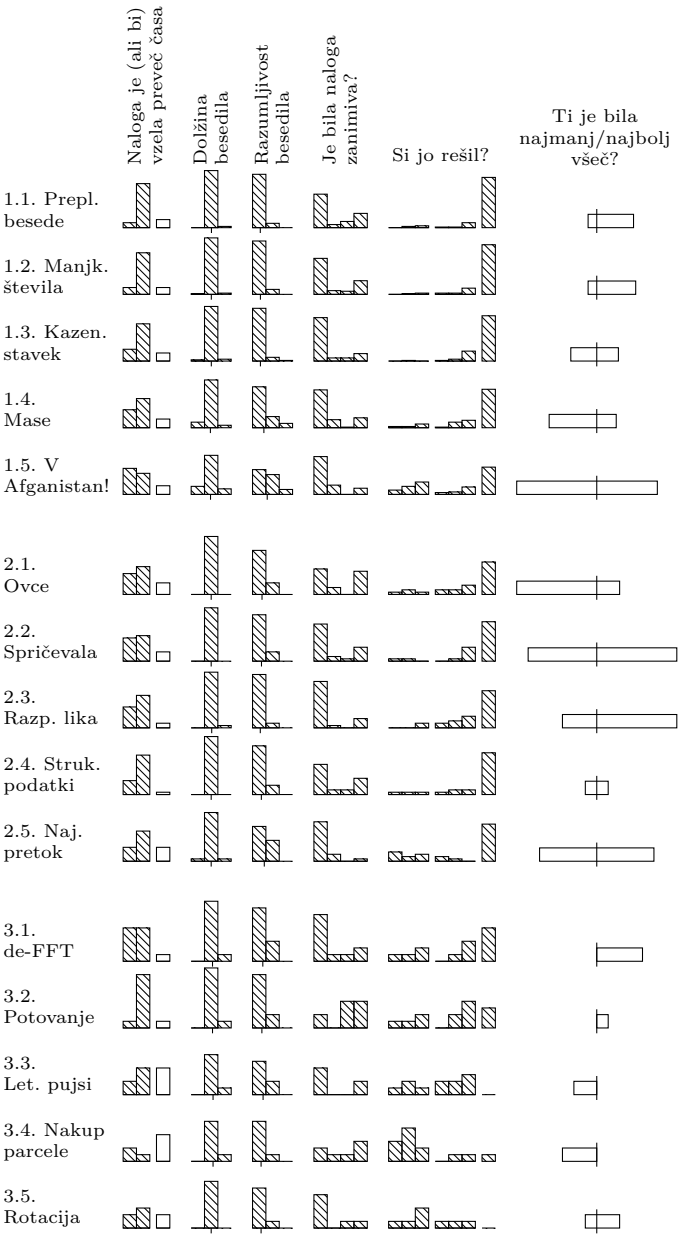
Pomen stolpcev v vsaki vrstici:

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

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

Mnenje tekmovalcev o nalogah

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



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

Naloga je (ali: bi) vzela preveč časa:

- da
- ne
- ne vem

Dolžina besedila:

- prekratko
- primerno
- predolgo

Razumljivost besedila:

- razumljivo
- težko razumljivo
- nerazumljivo

Naloga je bila:

- zanimiva
- dolgočasna
- že znana
- povprečna

Si jo rešil?

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

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

3.4 (nakup parcele). Težava je najbrž predvsem v tem, da so se zdele tekmovalcem te naloge težke (saj drugače same po sebi niso pretirano zamudne).

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, je letos prišlo do zanimivega pojava: v prvi skupini je dobila naloga 1.5 (v Afganistan!) največ glasov pri obeh vprašanjih; nekateri so bili navdušeni, ker je bila težka in jim je predstavljala lep izziv, spet drugim pa se je zdela pretežka in jim zato ni bila všeč. V drugi skupini sta najbolj priljubljeni nalogi 2.2 (ponarejena spričevala) in 2.3 (razpolovišče lika), najbolj nepriljubljena pa 2.1 (ovce). Slednje je nekoliko presenetljivo, saj so lahke naloge, kot je ta, običajno med bolj priljubljenimi. V tretji skupini je bila najbolj priljubljena naloga 3.1 (de-FFT permutacija), najmanj pa 3.4 (nakup parcele), vendar je skupno število glasov v tretji skupini tako nizko, da gre v obeh primerih le za tri ali štiri glasove.

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.

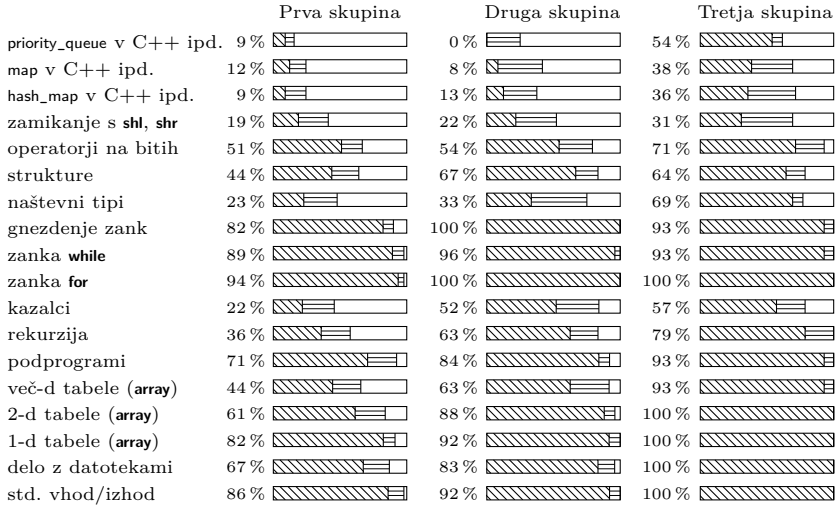


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let; v prvi skupini je pritrdilnih odgovorov še celo znatno več kot v preteklosti. Letos smo vprašanja o uporabi razredov map in priority_queue v C++ razširili in omenili še podobne razrede iz knjižnic drugih programskih jezikov; mogoče je to pripomoglo k temu, da je tudi pri teh vprašanjih letos malo več pritrdilnih odgovorov. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: rekurzija, kazalci, naštevni tipi in operatorji na bitih.

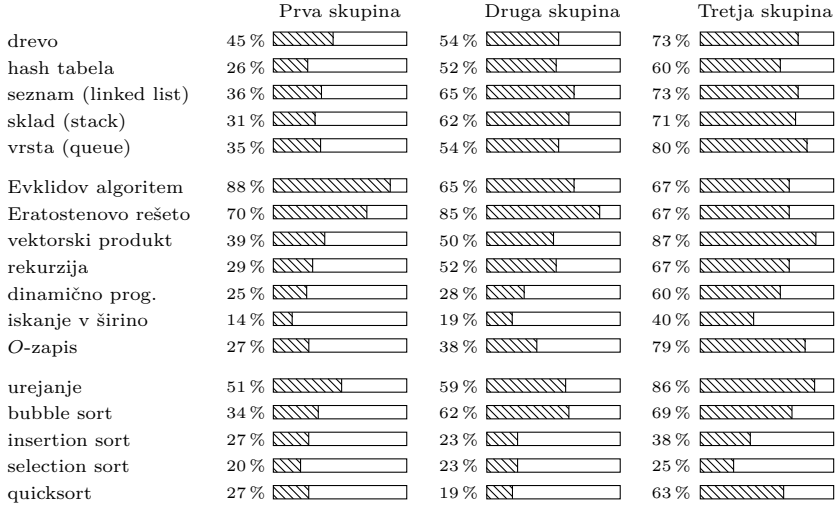


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

Uporaba programskih jezikov

Velika večina tekmovalcev tudi letos uporablja C in C++ (podobno kot zadnja leta je čisti C pravzaprav že redek), še naprej pa se krepi tudi uporaba pythona, ki se je zdaj v prvi skupini že izenačil s C++. Več ljudi kot prejšnja leta uporablja tudi C# in javo. Pascal se drži na približno istem nivoju kot prejšnja leta, basica pa že več let ni uporabljal nihče. Podobno kot prejšnja leta se je tudi letos pojavilo nekaj tekmovalcev (in tekmovalk), ki oddajajo le rešitve v psevdokodi ali pa celo naravnem jeziku, tudi tam, kjer naloga sicer zahteva izvorno kodo v kakšnem konkretnem programskem jeziku. Iz tega bi človek mogoče sklepal, da bi bilo dobro dati več nalog tipa „opiši postopek“ (namesto „napiši podprogram“), vendar se v praksi izkaže, da so takšne naloge med tekmovalci precej manj priljubljene in da si večinoma ne predstavljajo preveč dobro, kako bi opisali postopek.

Podobno kot v prejšnjih letih je v anketi precej tekmovalcev napisalo, da dobro poznajo tudi PHP, vendar ga je na tekmovanju uporabljal le eden.

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže tabela na str. 158. Glede štetja C in C++ v tej tabeli je treba pripomniti, da je razlika med njima majhna in včasih pri kakšnem krajšem kosu izvorne kode že težko rečemo, za katerega od obeh jezikov gre. Je pa po drugi strani videti, da se raba stvari, po katerih se C++ loči od C-ja, sčasoma povečuje (na primer `string` namesto **`char *`** in tip `vector` namesto tradicionalnih tabel).

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, C#, pythonu in javi. Deklaracije v C# so letos novost in smo jih dodali kot odziv na pripombe iz anket v prejšnjih letih. Očitno je to pomagalo, saj je delež tekmovalcev, ki pravijo, da deklaracije razumejo, zdaj še večji kot doslej (83/93 v prvi skupini in 23/24 v drugi). Nenavadno je, da so pri vprašanju, ali bi želeli deklaracije še v kakšnem jeziku, nekateri tekmovalci navedli

	Leto in skupina																	
	2012			2011			2010			2009			2008			2007		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	6	1	4	3	4	3	$4\frac{1}{2}$	5	2	4	2	1	$1\frac{1}{2}$	2	2	$8\frac{1}{2}$	2	1
C	7	2	1	7	2		6	6	1	$9\frac{1}{2}$	$3\frac{1}{2}$	$\frac{1}{2}$	$4\frac{1}{2}$	11	$2\frac{1}{2}$	$5\frac{1}{2}$	11	$6\frac{1}{2}$
C++	26	16	9	$23\frac{1}{2}$	19	8	33	$17\frac{1}{2}$	13	$26\frac{1}{2}$	2	$12\frac{1}{2}$	$17\frac{1}{2}$	11	$9\frac{1}{2}$	7	14	$15\frac{1}{2}$
java	17	$6\frac{1}{2}$	1	6	5	3	5	9	4	8	8	11	$9\frac{1}{2}$	3			$2\frac{1}{2}$	
PHP		1	–	$\frac{1}{2}$		–	1	1	–	2	1	–		2	–	1		–
basic			–			–			–			–			–		1	–
C#	17	1	3	4	2	3		$\frac{1}{2}$	1						3		$\frac{1}{2}$	–
python	25	5	–	20	6	–	12	2	–	4	$\frac{1}{2}$	–	6	1	–			–
NewtonScript		$\frac{1}{2}$	–			–			–			–			–			–
psevdokoda	3		–	6		–	4		–	8		–	–	–	–	–	–	–
nič	2			1	1		1	5		1				1		3		

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. Psevdokoda šteje tekmovalce, ki so pisali le psevdokodo, tudi pri nalogah tipa „napiši (pod)program“; pred letom 2009 takih nismo šteli posebej in če je kdo uporabljal le psevdokodo, je štet pod „nič“.

jezike, v katerih deklaracije že imamo, na primer javo in C#.

V rešitvah nalog zadnja leta objavljamo izvorno kodo le v C-ju; tekmovalce smo v anketi vprašali, če razumejo C dovolj, da si lahko kaj pomagajo s to izvorno kodo, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Večina je s C-jem zadovoljna (51/98 v prvi skupini, 21/26 v drugi, 13/16 v tretji; to je približno enak ali še malo boljši delež kot lani). Med jeziki, ki bi jih radi videli namesto (ali poleg) C-ja, jih največ omenja C++ in C#, v prvi skupini pa še zlasti python.

Letnik

Po pričakovanjih so tekmovalci zahtevnejših skupin v povprečju v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot prejšnja leta. V prvi skupini in drugi skupini sta nastopala tudi po dva osnovnošolca; teh pri izračunu povprečnega letnika v spodnji tabeli nismo upoštevali.

	Št. tekmovalcev po letnikih					Povprečni
Skupina	OŠ	1	2	3	4	letnik
prva	2	16	36	27	22	2,5
druga	2		5	12	14	3,3
tretja			4	4	10	3,3

Druga vprašanja

Podobno kot prejšnja je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!). V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale.

Pri vprašanju, kje so se naučili programirati, podobno kot prejšnja leta prevladujeta odgovora „sam“ in „v šoli“, v prvi skupini pa je skoraj prav toliko tudi tekmovalcev, ki pravijo, da so se programirati naučili na krožkih. Letos je v primerjavi z lanskimi leti delež samoukov nekoliko narastel.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. Podobno kot lani je v prvi skupini tudi precej takšnih tekmovalcev, ki želijo manj nalog ali pa (še raje) več časa (pri nespremenjenem številu nalog).

	Kje si izvedel za tekmovanje				Kje si se naučil programirati					Čas reševanja			Število nalog			Potekmovalne dejavnosti							
Skupina	od mentorja	na spletni strani	od prijatelja/sošolca	drugače	sam	pri pouku	na krožkih	na tečajih	poletna šola	hočem več časa	hočem manj časa	je že v redu	hočem več nalog	hočem manj nalog	je že v redu	izlet v tuji laboratorij	poletna šola	praksa na IJS	predstavitve tehnologij	predavanja o algoritmih	reševanje nalog	iskanje štipendije	iskanje podjetij
I	86	1	7	3	60	41	35	9	6	17	12	59	7	8	72	36	31	27	33	30	31	39	45
II	26	0	0	1	19	8	4	2	0	8	4	14	3	7	16	7	5	9	11	8	3	9	10
III	11	0	1	3	6	1	5	5	4	1	1	8	0	6	4	3	2	4	2	5	4	2	4

Iz odgovorov na vprašanje, kakšne potekmovalne dejavnosti bi jih zanimale, je težko zaključiti kaj posebej konkretnega.

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb. Od 2009 imajo tekmovalci v prvi in drugi skupini možnost pisati svoje odgovore na računalniku namesto na papir (kar so si prej v anketah že večkrat želeli). Velika večina jih je res oddajala odgovore na računalniku, nekaj pa jih je vseeno reševalo na papir. V anketo smo dodali tudi vprašanje o tem, kakšen se jim je zdel sistem za oddajo nalog; z njim so bili večinoma zadovoljni, tehničnih težav s shranjevanjem je bilo precej manj kot lani, je pa veliko tekmovalcev motilo delovanje avtomatskega shranjevanja v spletnem urejevalniku besedila: po avtomatskem shranjevanju je moral uporabnik namreč ponovno klikniti na urejevalno okno, preden je lahko nadaljeval z urejanjem besedila.

Veliko tekmovalcev si je tudi želelo, da bi imeli v prvi in drugi skupini na računalniki prevajalnike in podobna razvojna orodja. Razlog, zakaj se v teh dveh skupinah izogibamo prevajalnikom, je predvsem ta, da hočemo s tem obdržati poudarek tekmovanja na snovanju algoritmov, ne pa toliko na lovljenju drobnih napak; in radi bi tekmovalce tudi spodbudili k temu, da se lotijo vseh nalog, ne pa da se zakopljejo v eno ali dve najlažji in potem večino časa porabijo za testiranje in odpravljanje napak v svojih rešitvah pri tistih dveh nalogah.

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) Izviren način za preverjanje, ali je neka črka velika:

```
if s[i].upper() == s[i]:
   velike += s[i]
```

Nek drug tekmovalec je prišel na podobno idejo, vendar jo je implementiral na precej zgrešen način:

```
strcpy(S2, s);
tolower(S2[i]);
if (strcmp(s, S2) != 0) {
```

(1.1) Čudno, da se ni tekmovalcu zdelo sumljivo, da bi velike črke pokrivale kar 50 mest v tabeli ASCII... No, mogoče se je zatipkal in je mislil napisati 0x40 — tisto bi bilo potem pravilno.

```
if (s[a] < 91 and s[a] > 40) { // pomagamo si z ASCII tabelo znakov, da pogledamo,
                                  če je znak veliki, in ga dodamo k stringu velikih črk
```

(1.1) Rešitev, ki optimizira enostavnost tipkanja imen spremenljivk:

```
string qwe = "", ewq = "", t = "";
```

Nek drug tekmovalec je podobno uporabil imeni asdf in qwer.

(1.1) Rešitev z iracionalnim strahom pred praznimi nizi:

```
d := " ";       (* da se program ne sesuje *)
e := " ";
.
.
.
Del(e, 1, 1);   (* odstrani presledek *)
Del(d, 1, 1);
t := d + e;     (* združi velike in male črke *)
WriteLn(t);     (* izpiše *)
```

V izpuščenem delu programa nizoma `d` in `e` le pritika znake na koncu, tako da ni očitno, zakaj ne bi smela biti tadva niza na začetku prazna.

(1.1) Zakaj bi zgolj staknili dva niza, če lahko mednju vtaknemo še en prazen niz:

```
String t = velike + "" + male;
```

(1.1) Eden od tekmovalcev si je za vsako možno veliko ali malo črko privoščil štiri vrstice dolg blok kode v stilu:

```
else if (string[i] == 'X') {
   velike_crke[n_0] = string[i];
   n_0++;
}
```

Rezultat je 228 vrstic dolg podprogram, ki si zasluži letošnjo nagrado za najdaljšo oddajo.

Nek drug tekmovalec je naredil rešitev v podobnem duhu, le da je obdelal samo tiste črke, ki se pojavljajo v nizu iz primera pri besedilu naloge, torej `P`, `R`, `E`, `L`, `T`, `N`, `O`, `b`, `e`, `s`, `d`, `i`, `l`, `o`.

(1.1) Rešitev s predpostavko, da uporabljamo nek zelo ne-ASCIIjevski nabor znakov:

```
if Ord(s[a]) > 24 then    // če je ord od črke, na katere mestu se nahajamo,
                                                  večje od 24, kar pomeni, da je velika
```

Nič groznega ni, če človek ne ve natančno, kakšne so številske kode posameznih znakov, ampak potem bi lahko namesto s 24 primerjal z `Ord('Z')`; ali pa, kar je še lepše, primerjal znaka neposredno: `if s[a] > 'Z'`.

(1.1) Tudi letos se je pojavil priljubljeni večmestni ||, in to celo pri dveh tekmovalcih:

```
if (c == A || B || C || D || ... || Z) {
```

Pri tem so bile vse črke napisane eksplicitno.

(1.1) Posebna obravnava, ki ni tako zelo posebna:

```
if ((s[i] >= 'A' && s[i] <= 'Z') || (i == s.Length - 1 && s[i] >= 'A' && s[i] <= 'Z'))
   t += s[i];
```

(1.2) Pravilen, vendar neobičajen način za inicializacijo spremenljivke:

```
for (int x(1); x < b - a; x++) {
```

(1.2) Tale tekmovalec si je odločno privoščil nov ustavitveni pogoj, ki ga besedilo naloge ne omenja:

```
do {
   s = input.nextInt();
   for (int i = counter + 1; i < s; i++)
      t += "\n(" + i + ")";
   t += "\n" + s;
   counter = s;
} while (s >0); // zadnja vrednost mora biti 0. Deal with it
```

Bolj moteče je pravzaprav to, da njegova rešitev tisto ničlo, ki naj bi sicer le označevala konec vhodnih podatkov, tudi izpiše...

(1.2) Zanka s predvidljivim številom ponovitev:

```
a = c - 1;
while ((c - a) != 1)
{
   a++;
   printf("(%d)\n", a);
}
```

(1.2) Rešitev z odporom do celoštevilskih konstant v izrazih:

```
int y = 1;
do { //ugotavlja, če je med zapisanim številom in naslednjim več kot 1 razlike
   if (X[b + 1] == X[b] + y)
```

Vrednosti y seveda nikjer ne spreminja. Škoda, da ni napisal še X[b + y] namesto X[b + 1]...

(1.2) Zanimiv pogled na naključna števila:

```
var e: array [1..10000] of integer; (* 10000 je naključna številka, lahko je tudi večja *)
```

(1.2) Zanimiva razširitev sintakse za delo z datotekami:

```
dolzina = (stevila.txt).length # „dolzina“ je spremenljivka, v katero zapišemo število
a = (stevila.txt).readline(pos)                                    # vrstic v datoteki
⋮
(izhod.txt).writelines(vpis)
```

(1.2) Rešitev z veliko z-ji:

```
int z[] = input;
⋮
for (z = 0, z < z[], z++) {
   while (z[] < z[z[] − 1]) {          // če je element array manjši kot njegov predhodnik,
      z[] se zamenja z z[z[] − 1];     // bosta zamenjala mesta
```

(1.2) Rešitev z veliko odvečnega dela:

```
for symbol in lines:        # za znake v vrsticah naj velja
   if symbol == kriterij:
      kriterij += 1         # če je trenutno stevilo enako kriteriju ali pa če ni, naj se kriterij
   else:                    # poveča za ena
      kriterij += 1
```

(1.2) Nov prispevek v boju proti zapostavljanju stavka else:

```
if i not in f:
   f += (i)
else:
   pass
```

(1.2) Človek bi mogoče pričakoval, da kdor zna pretvoriti število v niz, zna tudi obratno, ampak ni vedno tako:

```
for y in range(0, 10**10): # poišče številsko vrednost zadnje vrstice
   if a == str(y):
      k = y
      break
```

(1.2) Rešitev, ki jemlje čiščenje vhodnih podatkov zelo resno:

KAKO SE BOM LOTIL NALOGE:
v neskončni zanki bom pral števila, ki jih vnese uporabnik

(1.2) Prispevek za rubriko „tekmovalci čestitajo in pozdravljajo“:

```
for (int kuli = 0; kuli < n) { // Zdelo se mi je, da sem kot a poimenoval že
   preveč spremenljivk, zato sem to imenoval po mojem zvestem prijatelju, ki mi je stal
   ob strani skozi vse tekmovanje
```

(1.2) Rešitev s prelaganjem dela na uporabnika:

```
int main(int argc, char argv) { // zakaj ne bi uporabili kar človeških možganov za takšno
}                                                                    // štetje?
```

(1.3) Eden od tekmovalcev je predpostavil, da je kazenski stavek vedno `NE BOM VEČ METAL PAPIRČKOV PO TLEH` (ki ima skupaj s presledkom na koncu 35 znakov):

```
public static int kazenskiStavek(String s) {
   int dolzinan = s.length() + 1;
   int dolzinas = 35;
   int rezultat = dolzinan / dolzinas;
   return rezultat;
}
```

Enako predpostavko je naredil tudi nek drug tekmovalec, ki pa se je potem v svoji rešitvi oprl na dejstvo, da se črka `R` v tem stavku pojavi enkrat samkrat, torej je dovolj, če njegov program izpiše število pojavitev črke `R` v nizu `s`.

(1.3) Še ena rešitev z zanašanjem na neupravičene poenostavitve: zapomni si prvi dve besedi in pogleda, kolikokrat se v nizu s še pojavi ta par besed.

> Da bi se ti dve besedi v istem zaporedju pojavili 2-krat v 1-nem kazenskem stavku, je malo verjetno.

Razne podobne neupravičene predpostavke, ki poenostavijo nalogo, je naredilo še več drugih tekmovalcev.

(1.3) Zanimiv nov izraz za spremenljivko:

> Program bi najprej prebral celotno besedilo, in bi ga shranil v eno neznanko.

(1.3) Nov in dekadenten pristop k poimenovanju spremenljivk:

```
char 1stavek[a];
char 2stavek[a];
```

(1.3) Rešitev s ponavljanjem problema naloge:

```
// Najprej moramo v nizu nakov najti podniz znakov, ki se ponavlja.
```

Tega, kaj točno to pomeni in kako bi to naredil, pa seveda ni napisal.

(1.3) Rešitev z odporom do povečevanja števil za več kot 1:

```
for (int c = 0; c < b.length(); c++)
   counter++;
```

(1.4) Rešitev z veliko slovarji:

> Vzamemo vsa števila jim prištejemo in odštejemo vsa števila iz range(a). Za vsako število te vrednosti shranimo v slovar. Nato križno primerjamo. Če se v vseh slovarjih ponovi isto število je izdelek en.

Do sem je še nekako v redu (čeprav neučinkovito in nas omeji na celoštevilske mase); težave se začnejo, če je izdelkov več vrst (v tem primeru nam slovarji ne koristijo kaj posebej).

(1.4) Rešitev z teorijo števil in tokom zavesti:

> Najhitrejše iskanje je iskanje skupnih večkratnikov števil. Torej vsakemu številu odštevamo oziroma prištevamo $+a$ in $-a$ toliko časa, dokler ne najdemo 1 skupnega večkratnika med vsemi meritvami. V list dodamo vse kombinacije večkratnike prvega števila. Naslednje drugo število bo imelo neujemajoč večkratnik mu nato v list dodamo vse neujemajoče večkratnike, a če se ujema vsaj en s prejšnjim večkratnikom, prištejemo številu $+1$, ampak isto storimo, če se jih ujema več, saj tako ne bomo šteli v prazno, le dodajali bomo nove ujemajoče večkratnike. [...]

(1.4) Klasičen primer rešitve, ki si navodilo „opiši postopek“ razlaga tako, da v bistvu opisuje program:

> Program najprej sprejme število meritev in to vrednost shrani v spremenljivko „st_meritev“, tipa **int**. [...] Deklariramo spremenljivko „stevec_vrst“, tipa **int** in jo nastavimo na začetno vrednost 1.

Tako se nadaljuje še približno pol strani. Še en podoben primer pri prvi nalogi:

> Nato s funkcijo cout vprašam uporabnika, koliko črk bo imela tabela, in vrednost, ki jo on vnese, v kvadratni oklepaj ([]) spremenljivke **char** vhodni[], s tem sem definiral velikost tabele. [...] Nato naredim še eno čisto isto **for** zanko, kot sem jo naredil malce prej, in z **if** stavkom preverjam, ali je v tabeli vhodni[a] prva črka večja ali enaka mali črki a in če je hkrati manjša ali enaka mali črki z. [...] S funkcijo **return** 0; končam program.

To je še posebej nenavadno, ker navodilo pri prvi nalogi pravi „napiši podprogram“ in ne „opiši postopek“ ali kaj podobnega. Ta tekmovalec je oddal takšne dolgovezne prozne opise izvorne kode (namesto izvorne kode same) kar pri vseh nalogah.

(1.4) Rešitev z omejitvijo na trivialne primere:

> Prvič preberemo podatke o odstopanju (a) ter zaporedje meritev. Z **if** stavkom preverimo naslednji pogoj (najmanjši + 2 * A <= največji), če je ta izpolnjen, potem je pravilni odgovor 1.

O tem, kaj storiti, če ta pogoj ni izpolnjen, pa nič.

(1.4) Rešitev z modularno aritmetiko:

> Seštejem vse meritve skupaj in jih delim po modulu z naključen izdelek $-a$, ostanek pri deljenju mi vrne število izdelkov.

To je cel odgovor tega tekmovalca pri tej nalogi. Težko si je predstavljati, kakšen razmislek je mogel pripeljati do tako zelo zgrešene rešitve.

(1.4) Ena od pogostejših napak pri tej rešitvi je, da poiščejo razliko med maksimalno in minimalno meritvijo ter jo delijo z $2a$ ali čim podobnim. V resnici tako dobimo le neko zgornjo mejo za minimalno število izdelkov.

(1.4) Ta naloga sicer ni bila mišljena kot realnočasovna, ampak eden od tekmovalcev jo je vendarle rešil v tem smislu:

> Če 60 s ni izdelka, prekini program in izpiši system.out.print StIZ.

(1.5) Precej ljudi si je napisalo rekurziven podprogram za iskanje najvišjega nadrejenega; neobičajno veliko pa jih je pri tem pozabilo na en **return**:

```
def NajvisjiNadrejeni(a):
    b = Nadrejeni(a)
    if b != −1:
        NajvisjiNadrejeni(b)
    else:
        return a
```

Podobno napako so naredili še vsaj trije drugi.

(1.5) Včasih se sliši o kremenitih vojakih, ampak tudi druge kamnine pridejo kdaj na vrsto:

```
// sedaj smo določili vse najvišje vojake, ki imajo pod seboj še kakega gnajstnega.
```

(1.5) Eden od tekmovalcev je predpostavil, da obstaja nek vojak, ki je (posredno ali neposredno) nadrejen vsem ostalim, zato lahko problem rešimo tako, da pošljemo ukaz le njemu:

```
program Afganistan;
begin
    WriteLn('1');
    ReadLn;
end.
```

Nenavadno pri tem je, da smo že v besedilu naloge lepo opisali primer, pri katerem takšnega vojaka ni in so potrebni vsaj trije ukazi...

(1.5) Še en komentar, ki odraža težko razložljivo nerazumevanje naloge:

```
# Najmanjše število ukazov je seveda, da pošljemo vse vojake v Afganistan
```

Zanimivo je tudi, da postopek, ki ga je v nadaljevanju opisal, ne dela tega, ampak se celo res trudi ugotavljati, kdaj ima več gajstnih vojakov skupnega nadrejenega in podobno. Je pa gornjo napačno idejo implementiral nek drug tekmovalec, torej lepo v stilu **for** (**int** i = 0; i < n; i++) **if** (Nadrejeni(i) == −1) ukaz++.

(1.5) Cela rešitev nekega tekmovalca pri tej nalogi:

> Najprej izdamo ukaz vojakom z najvišjim činom, nato preverimo, koliko gajstnih vojakov je ostalo.
> A GRE

No, kot je pripisal eden od ocenjevalcev: NE GRE.

(1.5) Rešitev za ljubitelje Lenina:[17]

```
while (Nadrejeni(c) != −1)
  if (Nadrejeni(c) != −1)
    c = Nadrejeni(c);
```

(1.5) Nekaj odlomkov iz ene od najbolj nekoherentnih rešitev:

```
Nadrejeni(i) = n − (n − 1)
c = število vodij
m = število vojakov
c − m = n
for vnos = n
  n /= a
  b = a − 1
print "b"
```

Pred tem spremenljivke a nikjer ni inicializiral.

(2.1) Rešitev s kombinatorično eksplozijo:

> ŠteviloOvac razdelimo na seznam vseh možnih seštevkov naravnih števil, ta seznam poimenujemo Kombinacije. s tem določimo vse možne kombinacije delavcev. [...] za vsak element v Kombinacije izračunamo skupni strošek striženja: [...]

Videti je tudi, da je pozabil preverjati, če je določeno kombinacijo sploh mogoče izvesti, ne da bi kateri od strižcev prekoračil časovno omejitev. Mimogrede, če to omejitev odmislimo, se izkaže, da je možnih razporedov ovac med strižce kar $\binom{n+m-1}{m-1}$.

Na podoben način je razmišljal še nek drug tekmovalec in se zato v komentarjih celo pritoževal, kako težka da je ta naloga.

(2.1) Iz komentarja na koncu ene od rešitev:

```
// Strošek pa bi lahko znižal tudi tako, da bi lastnik kmetije pomagal ostriči kakšno ovco :)
```

(2.1) Nekaj drobnih jezikovnih opažanj pri tej nalogi: nihče ne piše „ovac“, vsi „ovc“; precej tekmovalcev strižce imenuje „brivci“; eden pa v približno polovici primerov piše „štrižci“.

(2.1) Rešitev, ki je pripravljena na veliko strižcev:

```
private static int[][] man = new int[2147000000][2];
```

Pri peti nalogi ima isti tekmovalec še bolj ekstremno tabelo:

```
private static int[][] rac = new int[2147000000][2147000000];
```

Kar je še bolj zabavno: te tabele kasneje sploh ne uporablja.

(2.2) Letošnjo nagrado za najglobljo indentacijo dobi:

[17] „Zaupanje je dobro, kontrola še boljša.“

```
int main() {
   for (int n = 0; ...) {
      for (int g = n + 1; ...) {
         if (zap[n].L == zap[g].L) {
            for (int j = 0; ...) {
               if (rZapis[j] == zap[g].R) {
                  if (rZapis[j + 1] != zap[g].S) {
                     ⋮
                     for (int i = 0; ...) {
                        if (izpis[i] == st) {
                           obstaja = true;
```

(2.2) Rešitev z odporom do oglatih oklepajev:

```
*(i + trenutnasola) = *(sifrasole + i);
⋮
if ((*(trenutnasola + j + j) == *(sifrasole + i + j)) &&
   (*(sifraravnatelja + i + j) != *(trenutniravnatelj + i + j)))
```

Pri tem so sifrasole, trenutnasola ipd. tipa **int** *, spremenljivki i in j pa sta tipa **int**. Običajnega naslavljanja z oglatimi oklepaji ne uporablja nikjer.

(2.3) Naloga pravi, da je funkcija Znotraj(x, y) že podana, ampak naslednji tekmovalec jo je vendarle implementiral:

```
def Znotraj(x, y): # jemanje koordinat je v tem primeru brezpredmetno, ker imamo itak
   crn = random.randint(0, 1)                                    # random funkcijo
   return crn
```

Kraljestvo se tako navdihuje pri idejah kvantne fizike — s tem, ko ga pogledaš, ga že tudi spremeniš :)

(2.3) Ena od rešitev ima spremenljivki part1 in part2 tipa **int**, v katerih prešteje kvadratke nad in pod delilno črto; potem pa z njima počne naslednje:

```
if ((((float) part1 / (float) part2) <= 1 && ((float) part2 / (float) part1) >= 1) ||
    (((float) part1 / (float) part2) >= 1 && ((float) part2 / (float) part1) <= 1))
   // preverimo, ali je (1/2 manjša od 1 in 2/1 večja) ALI (1/2 večja in 2/1 manjša)
{
   if (((float) ((float) part1 / (float) part2)) <= 1) // preverimo, kateri ima manj
                                                       // kot 1 in povemo rezultat
```

Če odmislimo težave, do katerih bo prišlo, če bo eden od obeh delov prazen, je pogoj v prvem **if**u v bistvu tavtologija. Oba **if**a pa kažeta neznansko močan odpor do neposrednega primerjanja dveh **int**ov...

(2.4) Tabele in kazalci imajo v C/C++ sicer veliko skupnega, ampak tako veliko pa spet ne:

```
char line[100];
⋮
while (line[0] == ' ') /* Odstranimo oklepaje */
{
   line++;
}
```

(2.4) Rešitev z nezaupanjem do standardne knjižnice (Stack je tipa **char**[][]):

```
void IzpisiIme(unsigned int Niveau)
{
   // Izpiše ime nivoja na konzolo.
   // Ja, ja, printf to zmore (%s?), briga me, ziher je ziher.
   unsigned int i;
   while (Stack[Niveau][i])
      putchar(Stack[Niveau][i++]);
}
```

Stvar sicer ni tako zelo ziher, saj je pozabil inicializirati spremenljivko i.

(2.4) Odlična ideja za nov operator nad nizi. Če += pritakne niz na koncu, naj ga −= pač pobriše:

```
for i in seznamZnack: # pripravimo izpis značk pred besedilom
   izpisZnack += i
   izpisZnack += ".";
izpisZnack −= ".";
```

(2.4) Spodnja rešitev skuša v neskončni zanki brati vhodne podatke po vrsticah, dokler ne naleti na EOF. Žal uporabljena zanka ni tako zelo neskončna:

```
while (0)
{
   ⋮
}
```

(3.5) Lep prispevek na temo nepotrebnega kompliciranja s stavki **if** in **else**:

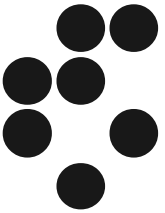
```
for (int i = 0; i < n; i++) {
   if (a[i] == b[i]) {
      continue;
   } else {
      if (a[i] > b[i]) {
         return(true);
      } else {
         return(true);
      }
   }
```

Lahko bi napisal preprosto:

```
for (int i = 0; i < n; i++)
   if (a[i] != b[i]) return true;
```

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 sta podprla naslednja odseka 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.

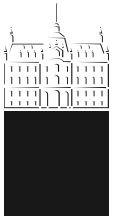
Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi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

E3 — Laboratorij za umetno inteligenco
Področje dela Laboratorija za umetno inteligenco so informacijske tehnologije s poudarkom na tehnologijah umetne inteligence. Najpomembnejša področja raziskav in razvoja so: (*a*) analiza podatkov s poudarkom na tekstovnih, spletnih, večpredstavnih in dinamičnih podatkih, (*b*) tehnike za analizo velikih količin podatkov v realnem času, (*c*) vizualizacija kompleksnih podatkov, (*d*) semantične tehnologije, (*e*) jezikovne tehnologije.

Laboratorij za umetno inteligenco posveča posebno pozornost promociji znanosti, posebej med mladimi, kjer v sodelovanju s Centrom za prenos znanja na področju informacijskih tehnologij (CT3) razvija izobraževalni portal VideoLectures.NET in vrsto let organizira tekmovanja ACM v znanju računalništva.

Laboratorij tesno sodeluje s Stanford University, University College London, Mednarodno podiplomsko šolo Jožefa Stefana ter podjetji Quintelligence, Cycorp Europe, LifeNetLive, Modro Oko in Envigence.

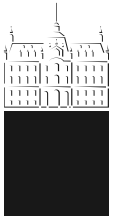
*



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.



Fakulteta za računalništvo in informatiko
Glavna dejavnost Fakultete za računalništvo in informatiko Univerze v Ljubljani je vzgoja računalniških strokovnjakov različnih profilov. Oblike izobraževanja se razlikujejo med seboj po obsegu, zahtevnosti, načinu izvajanja in številu udeležencev. Poleg rednega izobraževanja skrbi fakulteta še za dopolnilno izobraževanje računalniških strokovnjakov, kot tudi strokovnjakov drugih strok, ki potrebujejo znanje informatike. Prav posebna in zelo osebna pa je vzgoja mladih raziskovalcev, ki se med podiplomskim študijem pod mentorstvom univerzitetnih profesorjev uvajajo v raziskovalno in znanstveno delo.

Fakulteta za elektrotehniko, računalništvo in informatiko

Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. K temu bo fakulteta v prihodnjih letih (2009–2013) z razvojem kakovostnega oblikovanja in izvajanja naravoslovnih študijskih programov tudi stremela. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija

ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

REPUBLIKA SLOVENIJA
MINISTRSTVO ZA ŠOLSTVO IN ŠPORT

Ministrstvo za šolstvo in šport

Ministrstvo za šolstvo in šport opravlja upravne in strokovne naloge na področjih predšolske vzgoje, osnovnošolskega izobraževanja, osnovnega glasbenega izobraževanja, nižjega in srednjega poklicnega ter srednjega strokovnega izobraževanja, srednjega splošnega izobraževanja, višjega strokovnega izobraževanja, izobraževanja otrok in mladostnikov s posebnimi potrebami, izobraževanja odraslih ter športa.

BRONASTI POKROVITELJI

arnes