6. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 26. marca 2011

Bilten

Bilten 6. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2012

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Andrej Bauer, Luka Bradeško, Andrej Brodnik, Primož Gabrijelčič, Boris Gašperin, Tomaž Hočevar, Jurij Kodre, Mitja Lasič, Mark Martinec, Mojca Miklavec, Klemen Simonič, Andraž Tori, Mitja Trampuš, Miha Vuk, Klemen Žagar, Janez Brank.

Tisk: Present d. o. o., Ljubljana
Naklada: 350 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 (6 ; 2011 ; Ljubljana)
Bilten / 6. tekmovanje ACM v znanju računalništva, Ljubljana, 26. marca 2011 ; [avtorji nalog Nino Bašić ... [et al.] ; uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2012

Dostopno tudi na: `http://rtk.ijs.si/2011/rtk2011-bilten.pdf`

ISBN 978-961-264-039-2
ISBN 978-961-264-040-8 (PDF)
1. Bašić, Nino 2. Brank, Janez, 1979–
260809728

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 lani in predlani 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 24.) 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. 23–25 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 114–117.

Podobno kot v zadnjih dveh letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 28. januarja 2011. 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. 103–106). 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 215 tekmovalcev s 27 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. Vsota

Dano je zaporedje n celih števil, recimo jim $a_1, a_2, \ldots, a_{n-1}, a_n$. Vsa števila so večja od 0. Poleg tega je dano tudi celo število m. **Opiši postopek** (algoritem), ki poišče v tem zaporedju najkrajše tako strnjeno podzaporedje, pri katerem je vsota števil v podzaporedju večja od m. Poišče naj torej tak začetni indeks z in dolžino podzaporedja d, da bo $a_z + a_{z+1} + a_{z+2} + \ldots + a_{z+d-2} + a_{z+d-1} > m$ in da bo d čim manjši.

Tvoj postopek naj bo čim bolj učinkovit, tako da bo hitro našel pravi odgovor tudi pri zelo dolgih zaporedjih (takih z nekaj milijoni členov). Bolj učinkovite rešitve dobijo več točk.

Primer: recimo, da imamo zaporedje

$$5, 6, 5, 6, 9, 9, 9, 6$$

in $m = 24$. Najkrajše strnjeno podzaporedje z vsoto, večjo od m, je dolgo 3 člene: $9 + 9 + 9 = 27$, kar je večje od 24. (Do vsote 25 pri tem primeru ne moremo priti z nobenim strnjenim podzaporedjem, do vsote 26 pa sicer lahko, vendar potrebujemo za to vsaj štiri člene: $6 + 5 + 6 + 9 = 26$.) Pravilni rezultat pri tem primeru je torej $z = 5$, $d = 3$.

Pri tej nalogi ti ni treba pisati programa ali podprograma v kakšnem konkretnem programskem jeziku (lahko pa, če ti je lažje). Tvoj postopek sme predpostaviti, da so števila že podana v nekakšni tabeli ali seznamu, na primer takole:

```
const int n = ...;              /* v C/C++ */
int a[n];

const n = ...;                  { v pascalu }
var a: array [0..n − 1] of integer;

int[] a = ...;                  // v javi
int n = a.length;

a = [...]                       # v pythonu
n = len(a)
```

2. Lego kocke

Zoran dela v končni kontroli v tovarni legokock. Vrečke s kockami potujejo po tekočem traku, kjer je vsaka vrečka stehtana, predno se zapakira v škatlo. To se zgodi tako, da medtem ko vrečka potuje po tekočem traku, čitalnik črtne kode prebere črtno kodo z vrečke, tehtnica pa jo stehta na miligram natančno. Težo vrečke potem računalnik primerja z minimalno in maksimalno dopustno težo in če ne leži med njima, vrečko odstrani s tekočega traku. Tekoči trak se med tem ne ustavlja in tehtnica neprestano deluje, zato mora program poiskati, kdaj je teža te vrečke na tehtnici največja in potem to težo primerjati z referenčno težo.

Na voljo imamo:

- Tabeli **int** MinTeza[n], MaxTeza[n], ki vsebujeta minimalno in maksimalno dovoljeno težo za vsako vrsto izdelka — indeks je vrednost črtne kode.

Funkcije:

- **int** CrtnaKoda(), ki vrne črtno kodo z vrečke, ki je na tehtnici; če ni vrečke, vrne -1;
- **int** Tehtaj(), ki vrne težo, ki jo trenutno zaznava tehtnica, v miligramih; če na tehtnici trenutno sploh ni vrečke, vrne 0;
- **void** OdstraniVrecko(), ki odstrani vrečko, ki je na tehtnici, s tekočega traku. Ta funkcija se vrne iz klica šele, ko je vrečka popolnoma odstranjena.

(Za te funkcije torej predpostavi, da že obstajajo in jih lahko kličeš iz svojega programa.) **Napiši program**, ki se bo izvajal v neskončni zanki in nadzoroval obnašanje tega dela tekočega traku. Predpostaviš lahko, da vrečke potujejo izredno počasi v primerjavi s hitrostjo delovanja računalnika; da ko vrečka potuje po traku, se teža, ki jo zaznava tehtnica, najprej nekaj časa le povečuje, nato se nekaj časa le zmanjšuje, nato pa je nekaj časa enaka 0 (ko se je dosedanja vrečka že odpeljala mimo tehtnice in nanjo še ni prišla naslednja vrečka); in da je črtna koda vrečke vidna ves čas, ko je ta vrečka na tehtnici.

Še deklaracije v drugih jezikih:

```
{ v pascalu: }
var MinTeza, MaxTeza: array [1..n] of integer;
function CrtnaKoda: integer;
function Tehtaj: integer;
procedure OdstraniVrecko;

# v pythonu:
MinTeza = [...]; MaxTeza = [...]
def CrtnaKoda: ... # vrne int
def Tehtaj: ... # vrne int
def OdstraniVrecko: ...
```

3. Majevska števila

Maji so ljudstvo, živeče v južni Mehiki in severni Srednji Ameriki s tritisočletno zgodovino. Majevska pisava je bila v rabi vse do prihoda Evropejcev in je dolgo predstavljala veliko zagonetko.

Manj zagoneten pa je njihov sistem zapisa števil. Poznali so ničlo in podobno kot mi (za razliko od rimskih številk) uporabljali mestni zapis števil, vendar s številsko osnovo 20 (namesto naše bolj običajne 10). To jim je omogočalo zapisati zelo velika števila, kar je prišlo prav pri obvladovanju astronomije in koledarja.

Posamezne številke torej predstavljajo števila med 0 in 19, zapisana pa so kot skupek pik in črt, pri čemer vsaka pika predstavlja 1 in vsaka črta predstavlja vrednost 5. Maji so sicer običajno pisali črte ležeče in pike naložene na njih, vendar so lahko črte tudi pokončne in pike lahko ležijo tudi pred ali za njimi — da dobimo vrednost, moramo le pošteti vse črte in pike. Za potrebe te naloge bomo zapisali črto (vrednost 5) kot znak „|“, eno piko kot znak „.“, dve piki pa zaradi lepšega izgleda lahko zapišemo kot dvopičje „:“ ali pa kot dve piki „..“. Številko 0 so Maji narisali kot školjko, mi pa si bomo pomagali kar z našo ničlo „0“.

Nekaj primerov, kako lahko zapišemo posamezne številke:

```
 0 = 0
 1 = .
 2 = .. ali :
 3 = ... ali :. ali .:
 5 = | ali :.. ali :.: ali .:: ali ..... }
 8 = .:| ali :|.                          } ali še z nekaj drugimi
12 = :||                                  } kombinacijami znakov
19 = ::|||                                } „|“, „:“ in „.“, ki tu
                                          } niso naštete
```

Številke v mestnem zapisu številke ločimo med seboj s presledkom. Osnova je 20, na zadnjem mestu je torej faktor 1, na predzadnjem 20, na naslednjem z zadnje strani $20 \cdot 20 = 400$, na naslednjem $20 \cdot 20 \cdot 20 = 8000$, itd. Torej povsem enako, kot smo navajeni zapisovati v desetiškem sistemu, le da je faktor 20 namesto 10.

Nekaj primerov večjih števil:

Majevsko število	Števke	Vrednost
. 0	1 0	$1 \cdot 20 + 0 \cdot 1 = 20$
: ..	2 2	$2 \cdot 20 + 2 \cdot 1 = 42$
| 0 |:|	5 0 12	$5 \cdot 20 \cdot 20 + 0 \cdot 20 + 12 \cdot 1 = 2012$
. | :| |	1 5 7 5	$1 \cdot 20 \cdot 20 \cdot 20 + 5 \cdot 20 \cdot 20 + 7 \cdot 20 + 5 \cdot 1 = 10145$

Napiši program, ki bo prebral eno majevsko število, zapisano v eni vrstici, kot smo opisali zgoraj (pike, dvopičja in pokončne paličice, presledek loči števke) in izpisal vrednost števila, kot smo navajeni.

4. Kemijske formule

Podana je strukturna formula kemijske spojine (npr.: C_2H_5OH, H_2O, C_{60}). Elementi so podani z eno veliko tiskano črko angleške abecede, formule ne vsebujejo oklepajev. **Napiši podprogram** `void IzpisiAtome(char* s)`, ki bo izpisal, koliko atomov katere vrste je v spojini (vsak element naj izpiše v svoji vrstici, ni pa važno, v kakšnem vrstnem redu jih izpiše).

Primer: če pokličemo

```
IzpisiAtome("C2H5OH");
IzpisiAtome("H2O");
IzpisiAtome("C60");
```

naj se izpiše:

```
C 2
H 6
O 1

H 2
O 1

C 60
```

Še deklaracije v drugih jezikih:

```
void IzpisiAtome(string s);                  // v C++
public static void izpisiAtome(String s);    // v javi
procedure IzpisiAtome(s: string);            { v pascalu }
def IzpisiAtome(s): ...                      # v pythonu
```

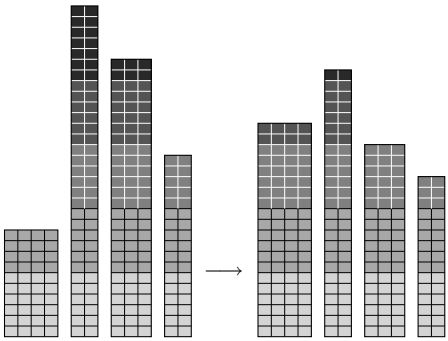
5. Okoljevarstveni ukrepi

V Butalah je začelo primanjkovati elektrike. Da bi vaščane motiviral k varčevanju z njo, je župan odredil, da bo elektrika za tiste, ki je porabijo veliko, dražja kot za tiste, ki je porabijo manj. Na koncu meseca pošljejo vsakemu vaščanu račun za elektriko, ki ga izračunajo po naslednjih pravilih:

- Recimo, da je ta vaščan porabil v tem mesecu x sodčkov elektrike[1] (x bo vedno celo število) in da ima ta mesec d dni.
- Za potrebe izračuna se predpostavi, kot da je vsak dan porabil natanko x/d sodčkov elektrike (ne glede na to, kako je bila v resnici razporejena poraba po mesecu).
- Za vsak dan se določi ceno elektrike takole: prvih 6 sodčkov tisti dan stane po 10 dinarjev vsak; naslednjih 6 sodčkov stane po 11 dinarjev vsak; naslednjih 6 sodčkov stane po 12 dinarjev vsak; naslednjih 6 sodčkov stane po 13 dinarjev vsak; vsa preostala poraba se zaračuna po 18 dinarjev na sodček.
- Ceno na dan potem pomnožimo s številom dni v mesecu (torej d) in tako dobimo znesek, ki ga bo moral ta vaščan ta mesec plačati za elektriko.

Primer: recimo, da ima mesec $d = 30$ dni in da smo porabili $x = 819$ sodčkov elektrike. Povprečna dnevna poraba je torej $x/d = 819/30 = 27{,}3$ sodčkov. Prvih 6 sodčkov na dan stane po 10 dinarjev, naslednjih 6 po 11, naslednjih 6 po 12, naslednjih 6 po 13, preostalih 3,3 sodčkov na dan (ker je $27{,}3 - 4 \cdot 6 = 3{,}3$) pa stane po 18 dinarjev. Cena na dan je torej $6 \cdot 10 + 6 \cdot 11 + 6 \cdot 12 + 6 \cdot 13 + 3{,}3 \cdot 18 = 335{,}4$ dinarjev, za ves mesec pa $335{,}4 \cdot d = 10\,062$ dinarjev.

V praksi porabijo Butalci pozimi več elektrike kot poleti in jo zato v mesecih, ko je porabijo veliko, tudi dražje plačujejo. Toda Cefizelj se je domislil, da bi lahko prihranil kar nekaj denarja, če bi porabo prerazporedil po mesecih (pri čemer bi skupna poraba ostala enaka).



Leva slika kaže primer porabe v štirimesečnem obdobju; sodčki, ki jih plačamo po višji ceni, so predstavljeni s temnejšimi odtenki sive. Širina stolpcev ponazarja dolžino meseca (v praksi seveda razlike v dolžini mesecev niso tako dramatične kot na naši sliki). Desna slika kaže, kako bi lahko isto skupno porabo razporedili med te štiri mesece tako, da bi bila skupna cena manjša (čeprav ne nujno tudi najmanjša možna).

Napiši program, ki prebere s standardnega vhoda 12 celih števil, ki predstavljajo porabo elektrike po mesecih od januarja do decembra, in

[1] Vsi vemo, da se električno energijo meri v kilovatnih urah, vendar je mestni starešina protestiral, češ da elektrike že ne morejo računati v urah, ker ima mesec za vse vaščane enako ur, pa jo odtlej računajo v sodčkih.

(*a*) izpiše skupno ceno elektrike, ki bi jo v tem letu pri takšni porabi po mesecih plačali; in

(*b*) izpiše, kako bi morali elektriko razporediti po mesecih, da bi bila skupna letna poraba enaka kot v vhodnih podatkih, skupna cena elektrike pa najmanjša možna. Izpiše naj tudi skupno ceno elektrike za celo leto pri tako preprazporejeni porabi. Pri tem pa upoštevaj omejitev, da mora biti mesečna poraba za vsak mesec še vedno celo število sodčkov. Če je možnih več enako dobrih rešitev, je vseeno, katero izpišeš.

Program, ki reši le podnalogo (*a*), dobi največ 8 točk (od 20 možnih). Pri vseh izračunih lahko predpostaviš, da ima februar 28 dni.

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. Ve,jic,e

Tomija je učiteljica v šoli čisto zastrašila z vejicami: ker so mu v spisih kar naprej manjkale, jih zdaj za vsak slučaj raje napiše malo več. Da jih ja ne bi bilo premalo, postavi kakšno celo sredi besede. Primer njegovega spisa:[2]

```
Na t,em vl,a,ku stre,žejo so,uv,laki, ne pa tudi hra,ne,
bogate z vla,kni,n,ami. Vp,liv lako,,te je, hud!
```

Ko napiše tak spis v *Mikromehki Besedi* ali drugem urejevalniku besedil, je zoprno, da ne more enostavno zamenjati vseh pojavitev nekega niza w_1 z drugim (enako dolgim) nizom w_2. Če se recimo želi prikupiti svoji učiteljici z Notranjskega in vse pojavitve niza `vlak` zamenjati z `wlah`, bi moral na koncu dobiti takšno besedilo:

```
Na t,em wl,a,hu stre,žejo so,uw,lahi, ne pa tudi hra,ne,
bogate z wla,hni,n,ami. Vp,liv lako,,te je, hud!
```

Njegov urejevalnik besedila pa tega ne zna, saj zgago delajo vejice, razmetane znotraj w_1.

Pomagaj ubogemu Tomiju. **Napiši podprogram** `Zamenjaj(s, w1, w2)`, ki izpiše besedilo, kakršno nastane, če v besedilu `s` zamenjamo vse pojavitve niza `w1` z enako

[2] Souvlaki je grška različica ražnjičev.

dolgim nizom w2. Pri teh zamenjavah mora ohraniti vse originalne položaje vejic, kot je tudi prikazano v zgornjem primeru. Tvoj podprogram naj bo takšne oblike:

```
void Zamenjaj(char *s, char *w1, char *w2);                    /* v C/C++ */
void Zamenjaj(string s, string w1, string w2);                 // v C++
public static void zamenjaj(String s, String w1, String w2);  // v javi
procedure Zamenjaj(s, w1, w2: string);                         { v pascalu }
def Zamenjaj(s, w1, w2): ...                                   # v pythonu
```

2. (Opomba)

Pri pisanju besedila uporabimo oklepaje, če želimo zapisati kakšno opombo (ali kaj drugega manj pomembnega). Včasih tudi znotraj opombe povemo kaj še manj pomembnega (torej manj pomembnega od opombe (ki že sama ni pomembna), a še vedno zanimivega); v takem primeru lahko uporabimo gnezdene oklepaje.

Gnezdenje oklepajev lahko načeloma neomejeno stopnjujemo — možno je imeti oklepaje v oklepajih v oklepajih v oklepajih ... v oklepajih. Če se nam pri branju takšnega teksta mudi, lahko preskočimo vse dele besedila, ki so gnezdeni vsaj k korakov globoko, pri čemer je vrednost k odvisna od tega, kako hudo se nam mudi.

Napiši podprogram SamoPomembno(s, k), ki sprejme niz s s tovrstnim besedilom in naravno število k, nato pa na standardni izhod izpiše to, kar ostane od besedila s, če iz njega pobrišemo vso vsebino, ki je vgnezdena več kot k nivojev globoko. Iz besedila naj bodo pri tem pobrisani tudi prazni oklepajski pari, t.j. takšni, ki ne vsebujejo med oklepajem in zaklepajem nobenega drugega znaka, niti presledka. Pozor, oklepajski par je lahko prazen tudi zato, ker smo iz njegove notranjosti brisali druge prazne ali pregloboko gnezdene oklepajske pare.

Predpostaviš lahko, da so oklepaji v besedilu smiselno postavljeni; tako na primer niz „`foo)bar(baz)qux`“ ni veljaven vhodni podatek.

Primer: ob klicu

```
SamoPomembno("ja (ja(ja(a b)(tri)) r((c d)(ef ghi))tk() ena) nič", 2);
```

se mora izpisati:

```
ja (ja(ja) rtk ena) nič
```

Tvoj podprogram naj bo takšne oblike:

```
void SamoPomembno(char *s, int k);                    /* v C/C++ */
void SamoPomembno(string s, int k);                   // v C++
public static void samoPomembno(String s, int k);     // v javi
procedure SamoPomembno(s: string; k: integer);        { v pascalu }
def SamoPomembno(s, k): ...                           # v pythonu
```

Če ti je ta naloga pretežka, lahko rešiš naslednjo lažjo različico: pobriši vso vsebino, zapisano v k ali več oklepajih, ne briši pa praznih oklepajskih parov. Rešitev te lažje različice dobi 10 točk (namesto 20).

3. Kozarci

Smo v proizvodnji večstranih kozarcev (imajo n stranic, pri čemer je n najmanj 4), kjer moramo nadzorovati stroj za barvanje stranic teh kozarcev. Na voljo imamo več funkcij za nadzorovanje tega stroja:

- **void** Naslednji() — vzame naslednji kozarec s tekočega traku (nekatere stranice na njem so lahko že pobarvane). Tega, kako je kozarec na začetku obrnjen, ne vemo.
- **void** Koncaj() — odloži ravnokar pobarvan kozarec na tekoči trak
- **void** Zavrzi() — odloži kozarec v zaboj s pokvarjenimi oziroma nepravilno pobarvanimi kozarci
- **void** Obrni() — obrne kozarec okoli vertikalne osi za $1/n$ obrata (torej za eno stranico naprej)
- **int** Preveri() — preveri, katera barva je na trenutno sprednji stranici; če trenutna stranica ni pobarvana, funkcija vrne 0
- **void** Pobarvaj(**int** barva) — pobarva trenutno sprednjo stranico z navedeno barvo. To funkcijo smeš poklicati le, če trenutna stranica še ni pobarvana.

Zahtevani vrstni red barv po stranicah kozarca je podan v tabeli **int** Barve[n]. (Število stranic, n, je tudi podano in je za vse kozarce enako.) **Napiši program**, ki v neskončni zanki jemlje kozarce s tekočega traku, poskrbi, da so pravilno pobarvani (pri tem pregleda že pobarvane stranice in pobarva še nepobarvane stranice), in jih odlaga na trak; če pa ugotovi, da kozarca ni mogoče pravilno pobarvati, naj ga odvrže.

Še deklaracije v drugih jezikih:

```
procedure Naslednji;
procedure Koncaj;
procedure Zavrzi;
procedure Obrni;
function Preveri: integer;
procedure Pobarvaj(Barva: integer);
```

```
def Naslednji(): ...
def Koncaj(): ...
def Zavrzi(): ...
def Obrni(): ...
def Preveri(): ... # vrne int
def Pobarvaj(barva): ...
```

4. Telefonske številke

Imamo n telefonskih številk, $s_1, \ldots, s_n$ (vse so k-mestne in v njih nastopajo števke od 0 do 9), ki jih pogosto kličemo. Katerikoli dve od njih se razlikujeta v vsaj štirih istoležnih števkah. Pri klicanju se pogosto zmotimo v eni števki (in torej pokličemo napačno številko), nikoli pa v več kot eni. Zdaj smo dobili seznam številk, ki smo jih dejansko poklicali, recimo $t_1, \ldots, t_m$ (tudi to so k-mestne številke, niso pa nujno vse različne), pri čemer je m precej večji od n. Originalnih $s_1, \ldots, s_n$ nimamo več, vemo pa, da se vsaka od njih pojavi vsaj enkrat nespremenjena v zaporedju $t_1, \ldots, t_m$. Iz zaporedja $t_1, \ldots, t_m$ torej ne moremo nujno ugotoviti, katere so originalne številke $s_1, \ldots, s_n$, lahko pa ugotovimo, katere skupine t-jev se nanašajo na isto originalno številko; **opiši postopek**, ki ugotovi velikost največje take skupine.

Primer: če imamo naslednje zaporedje 15 štirimestnih številk $t_1, \ldots, t_m$ (torej je $k = 4$ in $m = 15$):

1234, 2234, 4322, 3234, 2121, 1334, 4352, 1214, 5545,
2123, 4312, 4512, 5445, 4445, 5444, 5145, 5345

se dá ugotoviti, da so bile originalne številke štiri (torej $n = 4$) in da se na posamezne originalne številke nanašajo naslednje klicane številke:

- 1234, 2234, 3234, 1334, 1214 so vse nastale iz iste originalne številke;
- 4322, 4352, 4312, 4512 so vse nastale iz iste originalne številke;
- 5445, 5545, 5145, 5345, 4445, 5444 so vse nastale iz iste originalne številke;
- 2121, 2123 sta obe nastali iz iste originalne številke.

Za prve tri od teh štirih skupin lahko celo ugotovimo, kakšne so bile originalne številke s_i: to so bile 1234, 4312 in 5445; pri četrti skupini pa ne moremo ugotoviti, ali je bila originalna številka 2121 ali 2123. Kakorkoli že, vidimo lahko, da ima največja od teh skupin šest številk, tako da bi moral tvoj postopek pri tem primeru kot rezultat vrniti število 6.

5. Assembler

Pri tej nalogi bomo namesto običajnih programskih jezikov, kot so C/C++, pascal, java in podobni, uporabljali zelo preprost zbirni jezik (*assembly language*) za nek izmišljen, zelo preprost procesor.

Program je zaporedje ukazov (vsi možni ukazi so opisani spodaj); pred vsakim ukazom lahko stoji tudi oznaka (labela), na katero se lahko sklicujemo pri pogojnih skokih. Razen v primeru pogojnih skokov pa se ukazi izvajajo po vrsti.

Program lahko uporablja poljubno število celoštevilskih spremenljivk (kot tip **int** oz. integer).

Poleg tega obstajata še dve logični spremenljivki, E in L, ki ju ne moremo spreminjati ali brati neposredno, pač pa z njima delajo nekatere inštrukcije (CMP nastavi njuno vrednost, JE in JL pa njuno vrednost bereta).

Poleg tega obstaja tudi sklad, ki lahko hrani poljubno dolgo zaporedje (seznam) celoštevilskih vrednosti. Do vsebine sklada ne moremo dostopati drugače kot prek ukazov PUSH (ki doda nov element na vrh sklada) in POP (ki pobere element z vrha sklada).

Oglejmo si zdaj seznam vseh ukazov, ki jih podpira naš namišljeni procesor:

- ADD x, y — prišteje vrednost spremenljivke y spremenljivki x (rezultat shrani v x);
- SUB x, y — odšteje vrednost spremenljivke y od spremenljivke x (rezultat shrani v x);
- CMP x, y — primerja vrednosti spremenljivk x in y ter nastavi logični vrednosti (zastavici) E in L: E dobi vrednost **true**, če je $x = y$, sicer dobi vrednost **false**; L dobi vrednost **true**, če je $x < y$, sicer dobi vrednost **false**;
- SET x, c — nastavi spremenljivko x na vrednost c (ki mora biti neka celoštevilska konstanta);
- JE lbl — nadaljuje izvajanje za oznako lbl, če ima E vrednost **true**;

- JL lbl — nadaljuje izvajanje za oznako lbl, če ima L vrednost **true**;
- PUSH x — doda vrednost spremenljivke x na vrh sklada;
- POP x — pobere vrednost z vrha sklada in jo vpiše v spremenljivko x; če je bil sklad prazen, se procesor sesuje.

V tem zbirnem jeziku **napiši zaporedje ukazov**, ki izračuna zmnožek števil v spremenljivkah x in y; ob koncu izvajanja mora biti ta zmnožek shranjen v spremenljivki x. Predpostavi, da sta na začetku izvajanja vrednosti spremenljivk x in y večji ali enaki 0 in da sta dovolj majhni, da pri množenju ne bo prišlo do težav zaradi prekoračitve obsega celih števil ali česa podobnega. Poleg spremenljivk x in y lahko uporabiš še poljubno mnogo svojih pomožnih spremenljivk, poleg tega pa lahko uporabljaš tudi sklad, vendar mora biti sklad ob koncu izvajanja tvojega zaporedja ukazov vedno v enakem stanju kot na začetku izvajanja.

Zaželeno je, da je tvoja rešitev čim bolj učinkovita, torej da ob računanju zmnožka $x \cdot y$ izvede čim manj ukazov.

Za primer si oglejmo naslednje zaporedje ukazov, ki rešuje malo drugačen problem — s sklada pobere najprej n in nato še n števil ter na koncu v spremenljivki vsota izračuna vsoto tistih n števil.

```
   POP n
   SET vsota, 0
zacetek:
   SET temp, 0
   CMP n, temp
   JE konec
   POP x
   ADD vsota, x
   SET temp, 1
   SUB n, temp
   CMP temp, temp
   JE zacetek
konec:
```

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 1.6 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 1.6), Visual Studio 2005/2010 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 prva 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. Reklama (`reklama.in`, `reklama.out`)

Stanovanjsko naselje je razdeljeno na pravokotno mrežo gospodinjstev s h vrsticami in w stolpci; v vsaki celici mreže je eno gospodinjstvo. Oglaševalska agencija želi razširiti novico o novem izdelku. V preteklosti je bila opravljena anketa, s pomočjo katere so določili k potencialnih kupcev. Ker bi obiskovanje vsakega gospodinjstva posebej vzelo preveč časa, so se odločili izvesti predstavitev izdelka samo v enem gospodinjstvu (ne nujno v gospodinjstvu potencialnega kupca!). Vedo namreč, da se bo novica o izdelku hitro razširila po naselju. Iz gospodinjstva, ki je že prejelo novico (neposredno s predstavitvijo ali posredno preko svojih sosedov), se v enem dnevu novica razširi na vsa štiri sosednja gospodinjstva, ki imajo z njim v mreži skupno eno stranico. Širjenje reklame je neodvisno od zainteresiranosti članov gospodinjstva za nakup izdelka. **Napiši program**, ki ugotovi, v najmanj kolikšnem času lahko oglaševalci obvestijo vseh k potencialnih kupcev o novem izdelku, če izvedejo predstavitev v najbolj primernem gospodinjstvu.

Vhodna datoteka: v prvi vrstici so podana cela števila h, w in k, ločena s po enim presledkom. Sledi seznam lokacij potencialnih kupcev, ki jih je potrebno obvestiti o izdelku. Vsak kupec je opisan v svoji vrstici s številko vrstice in stolpca gospodinjstva. Veljalo bo $1 \leq h \leq 1000$, $1 \leq w \leq 1000$ in $1 \leq k \leq 1000$. Vrstice so oštevilčene od 1 do h, stolpci pa od 1 do w.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število dni, v katerem lahko novica doseže vseh k potencialnih kupcev, če izvedemo začetno predstavitev v najbolj primerno izbranem gospodinjstvu.

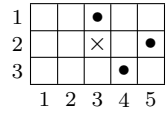
Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
2
```

Slika tega primera:



Pike • označujejo potencialne kupce. Če izvedemo predstavitev v drugi vrstici in tretjem stolpcu (kjer je znak ×), bo novica dosegla vse potencialne kupce v dveh dneh (tistega v vrstici 1 bo dosegla že prvi dan, ostala dva pa drugi dan).

2. Kompresija slike (`slika.in`, `slika.out`)

Da bi prihranili na količini prostora, potrebnega za zapis sivinske slike, želimo zmanjšati število sivinskih nivojev na največ k odtenkov. Izbrati je torej treba največ k odtenkov in vsako točko na sliki pobarvati z enim izmed k izbranih odtenkov. Vsi sivinski odtenki morajo biti cela števila med vključno 0 in 1000. Napako tako zapisane slike definiramo kot vsoto absolutnih vrednosti razlik med vrednostmi slikovnih točk na začetni in končni sliki. **Napiši program**, ki ugotovi, kolikšna je najmanjša napaka, ki jo lahko dosežemo pri taki kompresiji slike.

Vhodna datoteka: v prvi vrstici vhodnih podatkov so podana cela števila h, w in k, ločena s po enim presledkom. Sledi opis sivinske slike v obliki tabele s h vrsticami in w stolpci. Vrednosti slikovnih elementov so ločene s presledki in so po velikosti med vključno 0 in 1000. Veljalo bo $1 \leq w \leq 100$, $1 \leq h \leq 100$ in $1 \leq k \leq 20$.

Pri 60 % testnih primerov so vrednosti vseh slikovnih elementov med vključno 0 in 50.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer minimalno napako pri kompresiji slike na k sivinskih odtenkov.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
3
```

Možen končni videz te slike po kompresiji bi bil takšen (tega ne piši v izhodno datoteko — tu je podan le kot ilustracija):

```
1 1 1 1
3 3 1 1
```

3. Konstrukcija grafa (`graf.in`, `graf.out`)

Urbanist se ukvarja s posebno idejo ureditve glavnega mesta. Raziskave so pokazale, da večina turistov prispe v mesto preko letališča in nadaljuje svojo pot z železniške postaje na drugem koncu mesta. V času med prihodom letala in odhodom vlaka pa si krajšajo čas z ogledom nekaterih izmed 30 mestnih znamenitosti. Letališče in železniška postaja sta izjemna arhitekturna dosežka in se uvrščata v omenjenih 30 znamenitosti. Cilj urbanista je narediti mesto čim privlačnejše za turiste. V ta namen bo izdelal načrt enosmernih ulic, ki bodo povezovale znamenitosti. Zanimivost in prepoznavnost mesta pa želi poudariti s tem, da bo možno priti z letališča do železniške postaje na točno n različnih načinov (pri tem so dovoljene tudi take poti, ki obiščejo kakšno znamenitost po večkrat; niso pa dovoljene poti, ki bi šle po kakšni ulici v napačno smer). Župan mu je prepovedal uporabo vzporednih ulic, torej lahko neposredno od znamenitosti A do znamenitosti B vodi največ ena ulica. **Napiši program**, ki prebere n in izpiše načrt ulic, ki ustreza tem zahtevam.

Vhodna datoteka: vsebuje eno samo celo število, n, ki predstavlja želeno število različnih poti od letališča do železniške postaje. Veljalo bo $n \leq 10^8$, v 70 % testnih primerov pa bo veljalo tudi $n \leq 5000$.

Izhodna datoteka: za vsako enosmerno ulico izpiši po eno vrstico, v njej pa naj bosta števili A in B (ločeni z enim presledkom), ki povesta, da ulica pelje od znamenitosti A do znamenitosti B. Znamenitosti so oštevilčene s števili od 1 do 30. Letališče je označeno s številom 1, železniška postaja pa z 2.

Če obstaja več različnih pravilnih rešitev, je vseeno, katero od njih izpišeš.

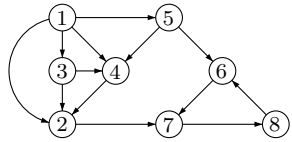
Primer vhodne datoteke:

```
5
```

Ena od možnih pripadajočih izhodnih datotek:

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

Slika tako dobljenega omrežja ulic:



4. Mušji drekci (`musji.in`, `musji.out`)

V predalu sameva kuverta; nanjo se sčasoma poserje n mušic. Končno jo iz predala potegne tajnica, ki se ji grozno mudi oddati pošto. Kuverta je edina, ki jo ima, po drugi strani pa mora z njo odposlati strašno ugleden dopis, zato s kuverto nikakor ne želi razkazovati prebave lokalnih mušic. Edina možna rešitev je, da neomiko prikrije z dvema znamkama.

Napiši program, ki prebere opis drekcev na kuverti ter velikost znamk in pove, ali je možno z dvema znamkama prekriti vso mušjo nesnago. Drekce obravnavamo kot točke. Znamki sta enako veliki, imata kvadratno obliko ter ravne robove in morata biti nalepljeni vzporedno s stranicami kuverte. Znamki se lahko prekrivata in lahko gledata čez rob kuverte. Drekec, ki leži točno pod robom znamke, šteje kot pokrit.

Vhodna datoteka: vhodni podatki so sestavljeni iz več testnih primerov. Na vhodu bo zato v prvi vrstici napisano število testnih primerov c (to je celo število in zanj velja $1 \leq c \leq 10$). Sledi c blokov podatkov; njihovo strukturo opisuje naslednji odstavek.

Vsak posamezen blok se začne s prazno vrstico. Sledi ji vrstica z n ($0 \leq n \leq 10^5$), številom drekcev. V naslednji vrstici se nahaja celo število a, dolžina stranice znamke v mikrometrih. Vsaka od naslednjih n vrstic vsebuje dve s presledkom ločeni celi števili, x_i in y_i; to sta koordinati i-tega drekca, prav tako v mikrometrih. Vsa števila a, x_i in y_i so večja ali enaka 1 in manjša ali enaka 10^9.

V 50 % testnih primerov bo dodatno veljalo $n \leq 5000$; v 20 % testnih primerov bo veljalo $n \leq 300$.

Izhodna datoteka: za vsakega od testnih primerov izpiši v samostojni vrstici `DA`, če je drekce mogoče prekriti z znamkama na opisani način, sicer `NE`.

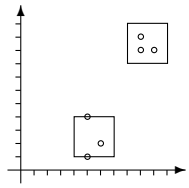
Primer vhodne datoteke:

```
2

6
3
5 1
5 4
6 2
9 10
9 9
10 9

3
10
1 1
100 1
200 1
```

Pripadajoča izhodna datoteka:

```
DA
NE
```

Komentar: v prvem primeru lahko z eno znamko pokrijemo prve tri naštete drekce in z drugo preostale tri, na primer takole:



5. Podajanje žoge (`podaje.in`, `podaje.out`)

Otroci, razdeljeni v dve ekipi, se postavijo na križišča kariraste mreže. Med seboj si podajajo žogo po naslednjih pravilih: en otrok lahko poda žogo drugemu le, če oba pripadata isti ekipi in stojita v isti vrstici ali stolpcu in med njima ni nobenega drugega otroka. **Napiši program**, ki prebere koordinate vseh otrok in izračuna najmanjše potrebno število podaj, v katerem lahko pride žoga od določenega otroka do določenega drugega otroka.

Vhodna datoteka: v prvi vrstici je število otrok, n. Sledi n vrstic, ki po vrsti opisujejo posamezne otroke. Vsaka od teh vrstic vsebuje tri cela števila, ločena s po enim presledkom: najprej x-koordinato tega otroka, nato y-koordinato tega otroka in končno še številko ekipe, ki ji ta otrok pripada (1 ali 2). Nikoli se ne zgodi, da bi dva ali več otrok stalo na isti točki.

Na koncu pride še vrstica z dvema različnima celima številoma, s in t, ločenima z enim presledkom. To sta številki otrok (velja torej $1 \leq s \leq n$ in $1 \leq t \leq n$) in sta pri vseh testnih primerih izbrani tako, da otroka pripadata isti ekipi.

Veljalo bo $1 \leq n \leq 100\,000$, vse koordinate otrok pa so večje ali enake 0 in manjše ali enake 10^6. Pri 50 % testnih primerov bo $n \leq 1000$, koordinate otrok pa bodo manjše ali enake 1000.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število podaj, v katerih je mogoče žogo spraviti od otroka s do otroka t (ob upoštevanju prej navedenih pravil podajanja). Če žoga od s do t sploh ne more priti, pa izpiši `-1`.

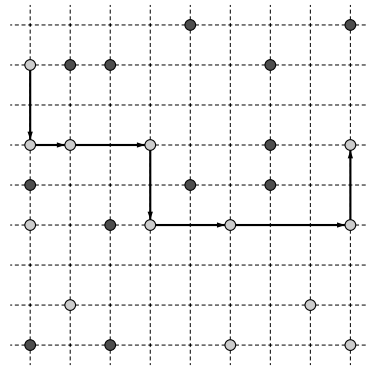
Primer vhodne datoteke:

```
25
0 0 1
2 0 1
5 3 2
8 3 2
0 5 2
4 4 1
6 4 1
6 5 1
1 7 1
2 7 1
8 0 2
1 1 2
2 3 1
0 4 1
0 7 2
0 3 2
3 3 2
7 1 2
6 7 1
4 8 1
8 8 1
5 0 2
8 5 2
1 5 2
3 5 2
15 23
```

Pripadajoča izhodna datoteka:

```
7
```

Slika tega primera:



NALOGE ZA ŠOLSKO TEKMOVANJE

28. januarja 2011

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. Primerjanje oklepajev

Komisija za računalniško tekmovanje ugotavlja, ali sta dve rešitvi naloge morda prepisani, tako, da iz njiju pobriše vse znake *razen* oklepajev `(`, `)`, `[`, `]`, `{` in `}` in nato primerja preostanka. Če sta enaka, šteje rešitvi za sumljivo podobni.

Napiši podprogram Prepisovanje(a, b), ki vrne **true**, če niza a in b predstavljata dve rešitvi naloge, ki sta sumljivo podobni. V nasprotnem primeru naj podprogram vrne **false**.

Tvoj podprogram naj bo takšne oblike:

```
bool Prepisovanje(char *a, char *b);                /* v C/C++ */
public static bool Prepisovanje(String a, String b); // v javi
function Prepisovanje(a, b: string): boolean;       { v pascalu }
def Prepisovanje(a, b): ...                         # v pythonu
```

2. Globalno segrevanje

Za naslednjih n let imamo po letih dane ocene, kako se bo zaradi globalnega segrevanja dvigovala gladina morij (predpostaviš lahko, da bo gladina morja vsako leto vsaj tako visoka kot prejšnje leto). Imamo tudi seznam trenutnih nadmorskih višin m obmorskih krajev po vsem svetu. Predpostaviš lahko, da je seznam krajev že urejen naraščajoče po nadmorski višini.

Opiši postopek, ki bo po letih izpisal vrstni red krajev, ki naj bi se vsako leto potopili zaradi dviga gladine morij. Tvoj postopek naj bo učinkovit, tako da bo hitro deloval tudi za velike n in m.

3. Riziko

Dva igralca se igrata naslednjo igro. Prvi igralec vrže tri kocke in jih uredi v nenaraščajočem vrstnem redu po številu pik; označimo jih z a_1, a_2 in a_3 (velja torej $a_1 \geq a_2 \geq a_3$). Drugi igralec vrže dve kocki in ju tudi uredi v nenaraščajočem vrstnem redu pik; označimo ju z b_1 in b_2 (velja torej $b_1 \geq b_2$). Nato igralca primerjata število pik na svojih kockah. Če je $a_1 > b_1$, dobi prvi igralec eno točko, sicer jo dobi drugi. Podobno, če je $a_2 > b_2$, dobi prvi igralec eno točko, sicer jo dobi drugi.

Met petih kock se lahko izide na $6 \cdot 6 \cdot 6 \cdot 6 \cdot 6 = 7776$ različnih načinov. **Napiši program**, ki ugotovi, koliko od teh 7776 izidov pripelje do tega, da dobi prvi igralec

dve točki, koliko do tega, da dobi drugi igralec dve točki, in koliko do tega, da dobi vsak od igralcev po eno točko.

4. Preglednice

Programi za delo s preglednicami (angl. *spreadsheet*) pogosto omogočajo izvoz v razne preproste tekstovne formate. Pri tej nalogi predpostavimo, da imamo celo tabelo zapisano kot en sam dolg niz znakov, pri čemer so posamezne vrstice tabele ločene s podpičji („;“), posamezne celice znotraj vsake vrstice pa z vejicami („,“).

Napiši funkcijo NaslednjiZnak(**char** c), ki ji podajamo tako zapisano tabelo znak za znakom (v parametru c), ta funkcija pa naj kliče funkcijo

```
Celica(int vrstica, int stolpec, char *vsebina)
```

čim prebere celotno vsebino celice (poleg vsebine celice naj poda tudi številko vrstice in stolpca, v kateri se nahaja; številčenje vrstic in stolpcev se prične pri 1). Za funkcijo Celica torej predpostavi, da že obstaja; tebi ni treba pisati nikakršne implementacije te funkcije, pač pa jo moraš le poklicati iz svoje funkcije NaslednjiZnak.

Primer: če je tabela predstavljena z nizom

```
Ime,Priimek;Janez,Novak;Ena,,Tri;...
```

lahko pričakuješ naslednje klice funkcije NaslednjiZnak:

```
NaslednjiZnak('I');
NaslednjiZnak('m');
NaslednjiZnak('e');
NaslednjiZnak(',');
NaslednjiZnak('P');
NaslednjiZnak('r');
⋮
```

ki jo moraš napisati tako, da bo klicala funkcijo Celica:

```
Celica(1, 1, "Ime");
Celica(1, 2, "Priimek");
Celica(2, 1, "Janez");
Celica(2, 2, "Novak");
Celica(3, 1, "Ena");
Celica(3, 2, "");
Celica(3, 3, "Tri");
```

Predpostaviš lahko, da je vsebina celice vedno krajša od 100 znakov in da ne vsebuje vejic ali podpičij. Uporabljati smeš tudi globalne spremenljivke, ki jih lahko pred prvim klicem funkcije NaslednjiZnak po svoje inicializiraš.

Še deklaracije v drugih jezikih:

```
procedure NaslednjiZnak(c: char);
procedure Celica(Vrstica, Stolpec: integer; Vsebina: string);

public static void naslednjiZnak(char c);
public static void celica(int vrstica, int stolpec, String vsebina);

def NaslednjiZnak(c): ...
def Celica(vrstica, stolpec, vsebina): ...
```

5. Smučarji

Dan je seznam m otrok in n parov smuči. Smuči so dovolj močne, da lahko zdržijo vsakega otroka; problem je v tem, da lažji otroci ne morejo peljati pretežkih smuči. Za vsakega otroka imamo njegovo težo in za vsak par smuči imamo minimalno težo otroka, ki ju še lahko pelje. **Opiši postopek**, ki iz teh podatkov ugotovi, koliko največ otrok lahko smuča. Poleg tega tudi utemelji, zakaj je tvoja rešitev pravilna (torej zakaj je rezultat, ki ga poišče tvoj postopek, res največje možno število otrok, ki jim je mogoče razdeliti smuči tako, da nihče ni prelahek za dobljeni par smuči).

NEUPORABLJENE NALOGE IZ LETA 2009

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 4. tekmovanjem IJS v znanju računalništva (leta 2009), 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. 77–102) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Avtobusi

Avtobusno podjetje „Vozi se ceneje“ bo podražilo cene na svojih progah. Za vsako relacijo so določili ceno prevoza med obema končnima postajama, težave pa imajo z izračunom cen prevoza za vmesne postaje. Njihovi ekonomisti so namreč določanje cen pošteno zapletli.

Vse njihove proge imajo skupno začetno postajo, zato bodo cene določili glede na razdaljo od te postaje. Postopek bo enak za vse proge, zato se pri tej nalogi ukvarjamo le z eno progo.

1. Izhodiščna cena prevoza naj bo sorazmerna razdalji od začetne postaje. Če označimo razdaljo med začetno in končno postajo z L, ceno prevoza med tema postajama pa s C, potem naj bo izhodiščna cena prevoza do postaje na razdalji L_k enaka $C \cdot L_k/L$.

2. Da bodo imeli šoferji lažje delo, naj bodo vse cene izražene v celem številu evrov. Da pa bo prevoznik v teh težkih časih zaslužil kakšen dodaten cent, naj bodo vse cene zaokrožene navzgor na najbližji evro. Da pa ljudi ne bi preveč prinašali naokoli, so sklenili, da naj bo cena prevoza kar najbližja izhodiščni ceni (zaokroženo navzgor, seveda).

3. Na koncu želijo še, da bi bile vse cene različne. Če bi po zgornjih dveh pogojih za dve postaji na razdaljah L_k in L_m, pri čemer je $L_k < L_m$, izračunali enako ceno, je treba prevoz do bolj oddaljene postaje podražiti za en evro.

4. Če bo zaradi razpostavitve postaj potrebno podražiti ceno vožnje od začetne do končne postaje, je to dovoljeno, a nova vrednost ne vpliva na izračun izhodiščne cene prevoza.

Napiši podprogram IzracunajCene, ki bo dobil na vhodu:

- želeno ceno prevoza na celi progi (PolnaCena; v evrih, že zaokroženo navzgor);
- tabelo razdalj do vseh vmesnih postaj in do končne postaje (v metrih);
- tabelo Cene, v kateri naj vrne cene prevozov;
- število postaj v zgornjih dveh poljih.

Tvoj podprogram naj izračuna cene prevozov do vseh postaj in jih shrani v tabelo Cene. Primer deklaracije takšnega podprograma:

```
const MaxPostaj = ...;
type TabelaT = array [1..MaxPostaj] of integer;
procedure IzracunajCene(PolnaCena: integer; Razdalje: TabelaT;
                        var Cene: TabelaT; StPostaj: integer);

void IzracunajCene(int polnaCena, const int razdalje[], int cene[], int stPostaj);

public static void IzracunajCene(int polnaCena, int[] razdalje, int[] cene, int stPostaj);

def IzracunajCene(polnaCena, razdalje, cene, stPostaj): ...
```

2. Podnizi

Napiši podprogram Kolikokrat, ki prešteje, v koliko besedah danega besedila se pojavi dani podniz. Če se dani podniz pojavi večkrat v isti pojavitvi besede, se to šteje le enkrat. Štejejo le strnjene pojavitve. Besedilo vsebuje le male črke angleške abecede in presledke, podniz pa le male črke angleške abecede. Tvoj podprogram naj bo takšne oblike:

```
function Kolikokrat(Besedilo, Podniz: string);
int Kolikokrat(char* besedilo, char* podniz);
int Kolikokrat(String besedilo, String podniz);
def Kolikokrat(besedilo, podniz): ...
```

3. Latovščina

Besedo predelamo v latovsko besedo tako, da poiščemo v njej prvi samoglasnik; soglasnike, ki so bili pred njim, premaknemo na konec besede; dodamo na konec še eno kopijo tistega prvega samoglasnika; in nato na začetek besede vrinemo *l*, na konec besede pa *te*. Primeri: *soba* → *lobasote*, *granit* → *lanitgrate*, *lata* → *latalate*. **Napiši podprogram**, ki kot parameter dobi neko besedo in izpiše latovsko različico te besede. Tvoj podprogram naj bo takšne oblike:

```
procedure Latovscina(Beseda: string);
void Latovscina(char* beseda);
public static void latovscina(String beseda);
def Latovscina(beseda): ...
```

Predpostaviš lahko, da vsebuje dana beseda zgolj male črke angleške abecede. Če dana beseda ne vsebuje nobenega samoglasnika, naj jo tvoj podprogram izpiše nespremenjeno.

4. Disk

Dan je trdi disk, razdeljen na m cilindrov, ki so oštevilčeni od 1 do m. Podane so tudi številke cilindrov, s katerih bi radi prebrali neke podatke: to so cilindri $a_1, a_2, \ldots, a_n$ (da bo lažje, lahko predpostaviš, da so urejeni v naraščajočem vrstnem redu). Bralnopisalna glava diska se trenutno nahaja nad cilindrom g. Podatke bi radi brali po načelu *shortest seek time first*, torej si za naslednji cilinder, ki ga bomo prebrali, vedno izberemo tistega (med cilindri, s katerih bi še radi kaj prebrali), ki je najbližje

trenutnemu položaju glave. **Opiši postopek**, ki ugotovi, v kakšnem vrstnem redu moramo prebrati cilindre $a_1, \ldots, a_n$, da bomo ustregli tej omejitvi. (Razdaljo med cilindri definiramo preprosto kot absolutno vrednost razlike med številkama cilindrov. Če sta v nekem trenutku dva neprebrana cilindra na enaki razdalji od trenutnega položaja glave, je vseeno, katerega se lotiš brati v nadaljevanju; ni ti torej treba minimizirati skupne dolžine vseh premikov glave ali česa podobnega.)

5. Železnica

(To je težja različica naloge, ki smo jo na tekmovanju leta 2009 uporabili kot peto nalogo v prvi skupini.) Po nekem železniškem tiru vozijo vlaki v obe smeri. Hitrost vlakov je različna, od 10 do 100 km/h. Vlaki so sestavljeni iz vagonov in lokomotive, ki so vsi dolgi po 10 m, med njimi pa je 1 m prostora.

Na tiru je zapornica, 1 km pred in za njo pa je na vsaki strani zapornice po en par optičnih senzorjev. Razdalja med senzorjema v paru je 1 m. Ko lokomotiva in vagoni potujejo mimo senzorjev, prekinejo svetlobni žarek, sicer pa je žarek neprekinjen (tudi npr. v presledku med dvema vagonoma). (Senzorja zaznavata lokomotivo povsem enako kot vagone, zato bomo v preostanku te naloge tudi lokomotive imenovali vagoni.)

Na zgornji sliki imamo na primer vlak s tremi vagoni; žarek senzorja 1 je trenutno prekinjen, senzorja 2 pa ne. Če bi se vlak premaknil malo proti levi, žarek senzorja 1 ne bi bil več prekinjen, pač pa bi se prekinil žarek senzorja 2.

Napiši podprogram SenzorSprememba(**int** stevilkaSenzorja, **bool** prekinjen), ki ga bo sistem poklical ob vsaki spremembi na katerem koli senzorju. Predpostaviš lahko, da se metoda izvede v trenutku. Če se istočasno zgodita dve spremembi senzorjev, se najprej izvede prvi klic metode, šele nato se metoda ponovno pokliče.

Takoj ko ugotoviš, da bo vlak peljal mimo zapornic, pokliči Zapornice(Dol), ko pa ugotoviš, da je že mimo, kliči Zapornice(Gor). Če ugotoviš, da se zapornicam bližata dva vlaka iz različnih strani, kliči Sirena(), da morda še preprečiš trčenje.

Če hočeš, lahko poleg svojega podprograma deklariraš tudi kakšne globalne spremenljivke in jih tudi po svoje inicializiraš.

Predpostavi, da se vlaki med vožnjo mimo senzorjev ne ustavljajo in ne spreminjajo smeri vožnje.

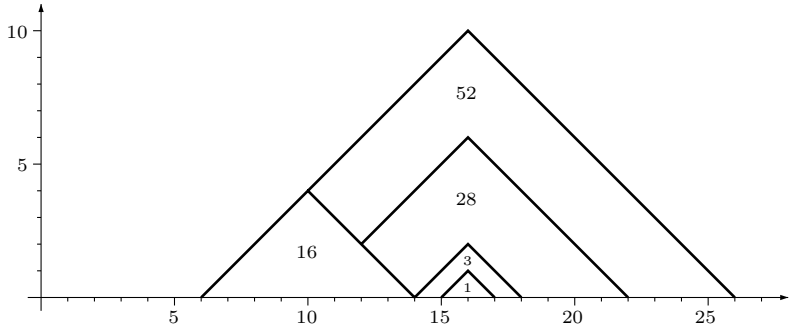
6. Peščene piramide

Finci pogosto hodijo v savno, kjer v potu svojega obraza opazujejo peščeno uro. Ob tem so se domislili naslednje naloge.

Na vodoravno premico z določenim koordinatnim izhodiščem z višine na različnih mestih vsipaš pesek. Sipni kot peska naj bo natanko 45°. Če torej na x-koordinati $x = 10$ vsujemo 16 enot peska enoto peska, dobimo piramido (pravzaprav trikotnik — naloga je v dveh dimenzijah) višine 4 s spodnjo stranico dolžine 8 (njegova ploščina je zato 16). Tako piramido bi opisal z zaporedjem točk

$(6,0),(10,4),(14,0)$. Če nato vsujemo na primer 1 enoto peska pri $x = 16$, nastane tam še ena manjša piramida in imamo zdaj dve piramidi z zaporedjem točk $(6,0),(10,4),(14,0),(15,0),(16,1),(17,0)$. Če z nasipanjem pri $x = 16$ nadaljujemo, se po nadaljnjih treh enotah peska naša druga piramida že toliko poveča, da se stakne s prvo: zdaj imamo zaporedje točk $(6,0),(10,4),(14,0),(16,2),(18,0)$. Po še nadaljnjih 28 enotah peska pri $x = 16$ imamo $(6,0),(10,4),(12,2),(16,6),(22,0)$. Po še nadaljnjih 52 enotah peska pri $x = 16$ pa druga piramida že popolnoma pogoltne prvo in imamo $(6,0),(16,10),(26,0)$. Za preizkus se lahko prepričamo, da ima tako dobljeni trikotnik ploščino 100, to pa je tudi vsota vseh količin doslej nasutega peska $(16 + 1 + 3 + 28 + 52 = 100)$.

Naslednja slika prikazuje stanje piramid v zgoraj opisanih trenutkih; števila 16, 1 itd. povedo količino peska v posamezni plasti:



Opiši postopek, ki na podlagi podatkov o tem, pri katerih x-koordinatah in v kakšnih količinah smo nasipali pesek (in v kakšnem vrstnem redu), izračuna zaporedje točk, ki opisujejo nastalo skupino piramid, tako kot smo to videli v zgornjem primeru.

7. Enkratno prirejanje

Recimo, da imamo tabelo a z n celicami, v vsaki od njih pa je celo število. Radi bi ugotovili, kolikokrat se v tej tabeli pojavi njen najmanjši element. (Na primer: če so v tabeli elementi $[5,3,7,3,5,4,5,7]$, je pravilni odgovor 2 — najmanjši element v tabeli je 3, ki se v njej pojavi dvakrat.) To lahko naredimo na primer s takšnim podprogramom:

```
int Kolikokrat(int a[], int n)
{
  int i, m, k = 0;
  for (i = 0; i < n; i++) {
    if (i == 0 || a[i] < m) { m = a[i]; k = 0; }
    if (a[i] == m) k = k + 1; }
  return k;
}
```

Napiši podprogram, ki reši ta isti problem (torej poišče število pojavitev najmanjšega elementa v tabeli) z naslednjo dodatno omejitvijo: nobeni spremenljivki

ne priredi vrednosti več kot enkrat. Z drugimi besedami, po tistem, ko neki spremenljivki prvič priredíš neko vrednost, je ne smeš več spreminjati. (To med drugim tudi pomeni, da (v pascalu) ne smeš uporabiti zanke **for**, ki bi šla z nekim števcem i od 0 do n − 1 ali kaj podobnega, saj bi tam prevajalnik poskrbel, da se bo i-ju v vsaki iteraciji zanke priredila nova vrednost. V C-ju in njemu sorodnih jezikih pa ne moreš uporabljati operatorjev, kot sta += in ++.) Poleg nove različice podprograma Kolikokrat lahko napišeš še več drugih podprogramov, ki jih bo tvoj Kolikokrat klical (ali pa se bodo oni klicali med sabo).

Težja različica: z enako omejitvijo (torej da lahko vsaki spremenljivki priredíš vrednost največ enkrat) napiši podprogram, ki uredi tabelo števil. Podprogram naj bo takšne oblike:

```
void Uredi(const int a[], int n, int b[]);
```

Pri tem je a vhodna tabela n števil, b pa je izhodna tabela, v katero moraš prepisati elemente tabele a v urejenem vrstnem redu. Vsakemu elementu tabele b lahko priredíš vrednost le enkrat, pa tudi morebitnim drugim spremenljivkam, ki jih tvoj podprogram uporablja, lahko priredíš vrednost le enkrat. Podobno kot prej lahko napišeš še več drugih podprogramov, ki jih bo klical podprogram Uredi ali pa se bodo klicali med sabo.

8. Poravnavanje desnega roba

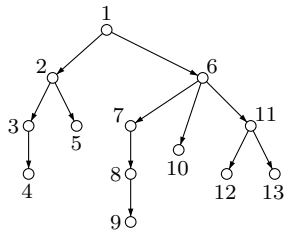
Neko besedilo bi radi izpisali s pisavo, v kateri so vsi znaki (tudi presledki, ločila itd.) enako široki. Da bo izpis videti lepši, bi radi, da bi bila vsaka vrstica (razen mogoče zadnje vrstice) dolga natanko n znakov. Pri tem posamezne besede ne smemo razdeliti med dve vrstici, lahko pa med besede vrivamo poljubno število dodatnih presledkov. Ob izpisu se nobena vrstica ne sme začeti ali končati na presledek. Dodatni presledki naj bodo razporejeni čim bolj enakomerno. **Napiši podprogram** Izpisi, ki kot parametra dobi niz z besedilom in želeno dolžino vrstice n ter izpiše besedilo na standardni izhod v skladu z zgoraj opisanimi omejitvami. Tvoj podprogram naj bo take oblike:

```
procedure Izpisi(Besedilo: string; n: integer);
void Izpisi(char* besedilo, int n);
public static void izpisi(String besedilo, int n);
def Izpisi(besedilo, n): ...
```

Predpostaviš lahko, da bo vhodno besedilo sestavljeno le iz presledkov in malih črk angleške abecede, da na začetku ali koncu besedila ni presledka, da nikjer v njem ne stojita dva presledka skupaj in da nobena beseda v njem ni daljša od n črk.

9. Lačni mravljinčar

Na sončnem travniku stoji veliko mravljišče, katerega rovi so drevesno razvejani: čisto pri vrhu je edina vhodna luknja, v kateri se začenja en ali več rovov. Vsak od rovov se lahko konča bodisi s kamrico bodisi s križiščem, v katerem se spet začenja najmanj en rov. Primer takšnega mravljišča je na sliki na str. 42 — vhodna luknja je označena s številko 1, prav tako pa imajo številke prirejena vsa križišča in kamrice. Za vhodno luknjo, križišča in kamrice bomo uporabljali skupno besedo



„točke“. Poznano je tudi, kako so mravlje razporejene znotraj mravljišča: v vsakem d_{ij} centimetrov dolgem rovu med dvema točkama i in j je enakomerno razporejenih $d_{ij}g_{ij}$ mravelj. Število g_{ij} je torej nekakšna gostota mravelj, ki jo merimo v mravljah na centimeter.

Do mravljišča pride požrešni mravljinčar, ki bi rad z enim samim potiskom svojega jezika v mravljišče polizal kar čim več mravelj. Mravljinčarjev jezik je dolg l centimetrov, rovi, kamrice in križišča pa so tako ozki, da mravljinčar jezika ne more zasukati za 180 stopinj.

Opiši postopek, ki na podlagi opisa mravljišča ter dolžine mravljinčarjevega jezika izračuna največje število mravelj, ki jih mravljinčar lahko naenkrat poliže.

10. Prepogibanje traku

Dan je niz znakov, $s = s_1 s_2 \ldots s_n$. Dovoljena operacija na njem je, da odrežemo prvih k znakov, kar pa lahko naredimo le, če velja $s_1 s_2 \ldots s_k = s_{2k+1} s_{2k} \ldots s_{k+2}$. Na podoben način smemo odrezati tudi zadnjih k znakov. Vprašanje je, kako dolg je najkrajši niz, ki se ga da z zaporedjem takšnih operacij dobiti iz danega začetnega s.

11. Tajna pošta

Tajna služba uporablja interno pošto, kjer kurirji prenašajo sporočila med pošiljateljem in sprejemnikom. Kurirja sta vsaj dva, in sicer „sprejemni kurir“ ter „oddajni kurir“.

Zakon o javnih službah zahteva, da se vsa pošta zabeleži, in sicer mora:

- sprejemni kurir, ki je pošto prejel od pošiljatelja, zabeležiti, kdo je pošto poslal, komu jo pošilja, in na kateri dan;
- oddajni kurir, ki je pošto dostavil prejemniku, zabeleži, komu je pošto oddal (prejemnik), od koga pošta prihaja, in na kateri dan je bila pošta poslana.

Ker pa tajna služba ne sme kot pošiljatelje in prejemnike navajati kar imen in priimkov tajnih agentov, vsakemu tajnemu agentu pripišejo enolično številko agenta; to so cela števila od 1 do n, pri čemer je n število vseh agentov. Da bi bilo razvozlavanje teh podatkov še težje, oba kurirja svoje zapisnike delno tudi šifrirata: sprejemni kurir šifrira številke prejemnikov s funkcijo g, oddajni kurir pa šifrira številke pošiljateljev s funkcijo h. Pri tem sta funkciji g in h permutaciji števil od 1 do n; po šifriranju so torej oznake še vedno števila od 1 do n in to različna za različne agente.

Torej, če na dan d pošlje agent A pismo agentu B, se v zapisniku sprejemnega kurirja znajde zapis $\langle d, A, g(B)\rangle$, v zapisniku oddajnega kurirja pa zapis $\langle d, h(A), B\rangle$.

Da bo naloga jasnejša, si oglejmo primer. Recimo, da si tajni agentje 1, 2, 3 in 4 pošiljajo pošto, in sicer takole:

Dan	Pošiljatelj	Prejemnik
1	1	2
1	2	1
2	1	3
2	1	4
3	3	4
3	4	1

In recimo, da kurirja uporabljata takšni šifrirni funkciji:

x	$g(x)$	$h(x)$
1	4	3
2	3	2
3	2	4
4	1	1

Potem sta njuna zapisnika videti takole:

Zapisnik sprejemnega kurirja			Zapisnik oddajnega kurirja		
Dan	Pošiljatelj	Prejemnik	Dan	Pošiljatelj	Prejemnik
1	1	3	1	3	2
1	2	4	1	2	1
2	1	2	2	3	3
2	1	1	2	3	4
3	3	1	3	4	4
3	4	4	3	1	1

Recimo zdaj, da sta nam po srečnem naključju prišla v roke oba zapisnika, poznamo pa tudi število n. Radi bi rekonstruirali šifrirni funkciji g in h. Tega se lahko lotimo na naslednji način: v vsakem zapisniku za vsakega pošiljatelja poglejmo, na katere dneve je kaj pošiljal in koliko pisem je poslal tisti dan; če te sezname dni primerjamo med sabo, lahko izvemo kaj o funkciji h. V zgornjem primeru iz zapisnika sprejemnega kurirja vidimo, da je agent 1 poslal tri pisma, in sicer eno na dan 1 in dve na dan 2; to lahko krajše zapišemo kot $\{1, 2, 2\}$. Podobno za agenta 2 dobimo spisek $\{1\}$, za agenta 3 spisek $\{3\}$ in za agenta 4 tudi spisek $\{3\}$. Iz zapisnika oddajnega kurirja podobno vidimo, da agent s šifro 1 dobi spisek $\{3\}$, agent s šifro 2 spisek $\{1\}$, agent s šifro 3 spisek $\{1, 2, 2\}$ in agent s šifro 4 spisek $\{3\}$. Če zdaj te spiske primerjamo, vidimo, da morata biti agent 1 iz prvega zapisnika in agent s šifro 3 iz drugega zapisnika ena in ista oseba, saj ni nihče drug poslal treh pisem na točno te dneve; torej je $h(1) = 3$. Podobno vidimo, da je $h(2) = 2$. Za ostala dva agenta pa iz te primerjave ne moremo zanesljivo ugotoviti, čigava je katera šifra, saj imata oba enak spisek dni pošiljanja (namreč $\{3\}$) — lahko bi bilo $h(3) = 1$ in $h(4) = 4$, lahko pa obratno, $h(3) = 4$ in $h(4) = 1$.

Na podoben način lahko namesto pošiljateljev gledamo prejemnike in za vsakega pripravimo spisek dni, ko je kaj prejel (in koliko pisem je na tisti dan prejel); s primerjavo teh spiskov bomo izvedeli kaj o funkciji g, podobno kot smo v prejšnjem odstavku o funkciji h.

Napiši program, ki prebere n in oba zapisnika in rekonstruira funkciji g in h po zgoraj opisanem postopku. Če kakšne od teh dveh funkcij po tem postopku ni mogoče enolično rekonstruirati, je vseeno, katero od možnih rešitev izpišeš.[3]

12. Mehurčki v slamici

Božiček je združil sponzorske obveznosti z ženinimi željami in tako po novem hujša s Coca Colo Zero. Ta pa je močno gazirana: če bi lahko pogledali v notranjost slamice, po kateri Božiček srka svoj zvarek, bi v njej videli n mehurčkov. Pri tem ima i-ti mehurček premer d_i in plava na višini h_i. Pri tem določimo, da $h = 0$ pomeni dno slamice, $h = 1$ pa vrh, torej velja $0 < h_i < 1$ za vsak i. Za potrebe naše naloge je slamica povsem ravna in stoji povsem navpično, mehurčki pa so ploščati (zasedajo torej točno samo prostor na višini h_i). Vsak mehurček plava navzgor s konstantno hitrostjo v_i, ki je odvisna le od njegovega premera: $v_i = f(d_i)$ in funkcija f je znana. Ko nek mehurček i dohiti kak večji mehurček j, se zaletita (slamica je namreč zelo ozka) in združita v nov mehurček premera $\sqrt{d_i^2 + d_j^2}$.

Opiši postopek, ki izračuna, koliko mehurčkov bo priplavalo do vrha slamice.

13. Kitka

Iz n trakov pletemo kitko. Na začetku vsi trakovi visijo drug ob drugem, obešeni na nosilno palico. Po vrsti jih označimo z zaporednimi številkami od 1 do n, nato pa jih v k korakih prepletemo. Na i-tem koraku pograbimo trak številka a_i in ga položimo desno od traku številka b_i. Oznak (številk) trakov pri tem ne spreminjamo. Na vsakem koraku seveda velja $1 \leq a_i \leq n$ in $1 \leq b_i \leq n$.

Opiši postopek, ki prebere zaporedje korakov spletanja kitke in naravno število m, $1 \leq m \leq n$, ter izpiše, kateri trak je po koncu pletenja m-ti po vrsti.

Veljalo bo $1 \leq n \leq 10^6$ in $1 \leq k \leq 10^6$.

14. Tovor

Muzej naprav USB je nekega dne dobil pošiljko USB-ključkov (mediji za shranjevanje podatkov). Nekateri ključki so že pokvarjeni, spet drugi so pripadali ljudem dvomljivega slovesa, zato jih nihče ne sme več priklapljati na računalnike. Vsak ključek ima tako neko količino prostora — recimo ji velikost — ki pa je ne moremo več ugotoviti, zaradi prej omenjenih razlogov. (Različni ključki so lahko enako veliki.)

V pošiljki je bil vsak ključek oštevilčen z zaporedno številko med 1 in n ($n \leq 10000$). Velikost ključka i imenujmo a_i. Pošiljki je priložen tudi nepopoln seznam parov številk ključev, kjer vsak par (a_i, a_j) pomeni, da je a_i po velikosti zagotovo manjši od a_j. Seznam so sestavili na Institutu IJS, zato vemo, da v njem zagotovo ni napak in nesmislov (npr. $a_1 < a_2$, $a_2 < a_3$, $a_3 < a_1$).

Ker v muzeju vedo, da so najbolj redki tisti ključki, ki imajo tudi najbolj redko velikost, bi radi razstavili ključek, za katerega lahko pri danih podatkih zaključimo, da je različen od čimveč drugih ključkov. **Opiši postopek**, ki ugotovi, kateri ključek je to (če je več enako dobrih, poišči vse).

[3] Zelo zanimivo nalogo dobimo tudi, če se vprašamo, kako lahko gornji postopek izboljšamo, da bo rekonstruiral funkciji g in h tudi v kakšnem od takih primerov, kjer ju sedanja različica postopka ne more rekonstruirati.

V izogib dvoumnostim zapišimo nalogo še malo bolj formalno. Imamo torej n spremenljivk $a_1, \ldots, a_n \in \mathbb{R}$ in m neenačb oblike $a_i < a_j$. Naj bo A množica vseh takih n-teric $\mathbf{a} = (a_1, \ldots, a_n)$, ki ustrezajo vsem m neenačbam (zagotovljeno je, da A ni prazna). Za vsak $k \in 1..n$ in $\mathbf{a} \in A$ naj bo $r(k, \mathbf{a}) = \{i \in 1..n : a_i \neq a_k\}$ množica indeksov spremenljivk, ki so v $\mathbf{a}$ različne od a_k. Naj bo $S(k) := \cap_{\mathbf{a} \in A}\, r(k, \mathbf{a})$ množica spremenljivk, ki so od a_k različne pri vseh dopustnih rešitvah, in $s(k) := |S(k)|$ naj bo število teh spremenljivk. Naloga sprašuje po tem, kateri k ima največjo vrednost $s(k)$.[4]

Primer: če imamo neenačbi $a_1 < a_3$ in $a_2 < a_3$, je odgovor a_3, kajti od nje sta gotovo različni a_1 in a_2. Pri a_1 (in podobno pri a_2) pa vemo, da je a_3 različna od nje, medtem ko je a_2 lahko tudi enaka a_1.

[4] Zelo zanimivo, vendar težjo nalogo dobimo, če si razlagamo besedilo malo drugače: definirajmo $s'(k) := \min_{\mathbf{a} \in A} |r(k, \mathbf{a})|$ in se vprašajmo, kateri k ima največjo vrednost $s'(k)$. Funkcija $s'(k)$ nam torej pove, da se pri vsaki dopustni rešitvi $\mathbf{a} \in A$ od a_k razlikuje vsaj $s'(k)$ spremenljivk (vendar so to lahko pri različnih $\mathbf{a}$ različne spremenljivke). Očitno je, da vedno velja $s'(k) \geq s(k)$, najti pa se dá primere, kjer so te neenakosti vsaj pri nekaterih k stroge in kjer je tudi $\max_k s'(k)$ strogo večji od $\max_k s(k)$. (Primer: pet spremenljivk in štiri neenačbe, $a_1 > a_2 < a_3 > a_4 < a_5$; hitro se lahko prepričamo, da je tu $\max_k s(k) = 2$ in $\max_k s'(k) = 3$.)

REŠITVE NALOG ZA PRVO SKUPINO

1. Vsota

Premikajmo se po zaporedju z dvema indeksoma: i kaže na prvi člen našega podzaporedja, j pa na zadnji člen podzaporedja. Na začetku postavimo i na 1 in se z j zapeljimo tako daleč, da bo vsota podzaporedja presegla m. Nato počasi povečujmo i. Vsakič, ko i povečamo za 1, izpade en člen iz vsote; če je ta zdaj manjša ali enaka m, moramo dodati v podzaporedje nekaj novih členov, torej povečajmo j (in popravimo vsoto; to ponavljajmo, dokler vsota ne preseže m).

Dolžino najkrajšega doslej najdenega podzaporedja si zapomnimo v `d`, njegov začetek pa v `z`. Spremenljivko `d` na začetku postavimo na $n+1$, kar je gotovo daljše od katerega koli podzaporedja; če bo imela na koncu še vedno to vrednost, bomo vedeli, da sploh ni nobenega podzaporedja z vsoto, večjo od m.

```
void Vsota(int a[], int n, int m)
{
  int z, d = n + 1, i = 0, j = -1, vsota = 0;
  while (i < n) {
    /* Dodajajmo člene v podzaporedje (in povečujmo j),
       dokler njegova vsota ne preseže m. */
    while (j + 1 < n && vsota <= m)
      vsota += a[++j];
    /* Če smo našli najkrajše zaporedje doslej, si ga zapomnimo. */
    if (vsota > m && j - i + 1 < d)
      z = i, d = j - i + 1;
    /* Premaknimo levi rob zaporedja za en člen naprej. */
    vsota -= a[i++]; }
  if (d > n) printf("primernega podzaporedja ni\n");
  else printf("%d %d\n", z, d);
}
```

2. Lego kocke

Spodnja rešitev hrani zadnje tri izmerjene teže v spremenljivkah `t1`, `t2` in `t3`. Ker naloga pravi, da teža najprej nekaj časa le narašča, nato pa nekaj časa le pada, bomo maksimalno težo prepoznali po tem, da je `t1 < t2` in `t2 > t3`. (Če bi se lahko zgodilo, da bi bila teža pri več zaporednih meritvah enaka, bi se to preverjanje malo zapletlo.) Takrat lahko pogledamo črtno kodo vrečke in preverimo, če njena teža leži na predpisanem območju.

```
#include <stdbool.h>
int main()
{
  int t1 = 0, t2 = 0, t3 = 0, koda;
  while (true)
  {
    t1 = t2; t2 = t3; t3 = Tehtaj();
    if (t1 < t2 && t2 > t3)
    {
      /* Našli smo lokalni maksimum teže. Preberimo črtno kodo
         vrečke in preverimo, če teža ustreza omejitvam. */
```

```
        koda = CrtnaKoda();
        if (t2 < MinTeza[koda] || t2 > MaxTeza[koda])
          OdstraniVrecko();
      }
    }
}
```

3. Majevska števila

Spodnja rešitev bere vhodne podatke znak po znak, vrednost trenutne številke računa v spremenljivki `stevka`, vrednost vseh predhodnih številk pa v `n`. Če je trenutno prebrani znak še del številke (torej pika, dvopičje, črta ali 0), prištejemo njegovo vrednost k trenutni številki; ko pa pridemo do konca številke (presledek, konec vrstice ali konec datoteke), pomnožimo dosedanji `n` z 20, mu prištejemo novo številko in postavimo spremenljivko `stevka` na 0, da bo pripravljena za naslednjo številko.

```
#include <stdio.h>
int main()
{
  int n = 0, stevka = 0, c;
  do {
    c = fgetc(stdin);
    if (c == '.') stevka += 1;
    else if (c == '0') stevka = 0;
    else if (c == ':') stevka += 2;
    else if (c == '|') stevka += 5;
    else n = 20 * n + stevka, stevka = 0;
  } while (c != EOF && c != '\n');
  printf("%d\n", n);
}
```

Slabost te rešitve je, da ne deluje pravilno, če sta dve zaporedni številki ločeni z več kot enim presledkom (ali pa če so odvečni presledki na koncu vrstice ipd.); obnašala bi se, kot da bi bila med dvema sosednjima presledkoma številka z vrednostjo 0. Temu se lahko izognemo z dodatno spremenljivko, ki pove, ali se je nova številka sploh že začela:

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
  int n = 0, stevka = 0, c; bool jeStevka = false;
  do {
    c = fgetc(stdin);
    if (c == '.') stevka += 1, jeStevka = true;
    else if (c == '0') stevka = 0, jeStevka = true;
    else if (c == ':') stevka += 2, jeStevka = true;
    else if (c == '|') stevka += 5, jeStevka = true;
    else {
      if (jeStevka) n = 20 * n + stevka;
      stevka = 0; jeStevka = false; }
  } while (c != EOF && c != '\n');
  printf("%d\n", n);
}
```

Bolj za šalo kot zares si oglejmo še naslednjo dekadentno enovrstično rešitev v pythonu. Prebrano majevsko število lahko predelamo v niz, ki predstavlja aritmetični izraz, in uporabimo funkcijo eval, da izračunamo njegovo vrednost. Spodnja rešitev predpostavlja, da po dve zaporedni majevski števki nista ločeni z več kot enim presledkom.

```
print (lambda s: eval("(" * s.count(" ") + s.replace(".", "1+").
   replace(":", "2+").replace("|", "5+").replace("0", "0+").
   replace(" ", "0)*20+") + "0"))(__import__("sys").stdin.readline().strip())
```

Kot zanimivost razmislimo še o tem, koliko je različnih majevskih zapisov posamezne števke; označimo z a_n število zapisov števke n. Vsak zapis števke n pripada eni od naslednjih treh skupin: (1) lahko se začne na `|` in nadaljuje s poljubnim zapisom števke $n-5$ (teh pa je a_{n-5}); (2) lahko se začne na `:` in nadaljuje s poljubnim zapisom števke $n-2$; (3) lahko pa se začne na `.` in nadaljuje s poljubnim zapisom števke $n-1$. Tako dobimo rekurzivno zvezo $a_n = a_{n-1} + a_{n-2} + a_{n-5}$; da bomo dobili prave rezultate, moramo za robne primere vzeti $a_n = 0$ za $n \leq 0$ (četudi je sicer pri $n = 0$ en zapis vendarle mogoč, namreč „0“). Tako dobimo:

n	a_n	n	a_n	n	a_n	n	a_n
0	1	5	9	10	128	15	1843
1	1	6	15	11	218	16	3 142
2	2	7	26	12	372	17	5 357
3	3	8	44	13	634	18	9 133
4	5	9	75	14	1 081	19	15 571

Skupaj je teh zapisov kar 37 660.

4. Kemijske formule

Naloga pravi, da je vsak element predstavljen z eno veliko tiskano črko angleške abecede, tako da je možnih kvečjemu 26 različnih elementov. Imejmo torej tabelo koliko s 26 celicami, ki za vsak element pove število doslej videnih atomov tega elementa. Na začetku postavimo vse vrednosti v tabeli na 0. Niz s pregledujmo od leve proti desni; prvi znak je črka, ki predstavlja element, nato pa pride nič ali več števk, ki skupaj tvorijo število atomov tega elementa. To število počasi računamo v spremenljivki n, na koncu (ko pregledamo že vse števke in pridemo do znaka, ki ni števka) pa za n povečamo ustrezno celico tabele koliko. Posebej moramo paziti na primer, ko za oznako elementa ni nobene števke, kajti to pomeni en atom tega elementa in ne 0. Na koncu se moramo le še sprehoditi po tabeli koliko in izpisati rezultate.

```
#include <stdio.h>

void IzpisiAtome(char *s)
{
  int koliko[26], element = -1, n;
  /* Inicializirajmo tabelo. */
  for (element = 0; element < 26; element++) koliko[element] = 0;

  while (*s)
  {
    /* Prvi naslednji znak mora biti črka, ki predstavlja nek element. */
    element = *s++ - 'A'; n = 0;
```

```
    /* Preberimo število, ki ga tvorijo števke za simbolom elementa. */
    while ('0' <= *s && *s <= '9') n = 10 * n + (*s++ - '0');
    /* Če števk sploh ni bilo (in je n = 0), imamo 1 atom tega elementa.
       To bi sicer dalo napačne rezultate pri formuli C0, ampak saj taka
       formula tako ali tako nima smisla. */
    if (n == 0) n = 1;
    /* Zabeležimo teh n atomov v tabeli koliko. */
    koliko[element] += n;
  }
  /* Izpišimo rezultate. */
  for (element = 0; element < 26; element++)
    if (koliko[element]) printf("%c %d\n", 'A' + element, koliko[element]);
  printf("\n");
}
```

5. Okoljevarstveni ukrepi

Najprej si pripravimo podprogram Cena, ki izračuna ceno elektrike za en mesec v odvisnosti od porabe in števila dni v mesecu. Če ima mesec d dni, nas bo prvih $6 \cdot d$ sodčkov stalo po 10 dinarjev, naslednjih $6 \cdot d$ sodčkov po 11 dinarjev, naslednjih $6 \cdot d$ sodčkov po 12 dinarjev, naslednjih $6 \cdot d$ sodčkov po 13 dinarjev in vsi preostali sodčki po 18 dinarjev. V spremenljivki cena se nam nabira skupna cena doslej obdelanih sodčkov, spremenljivko poraba pa počasi zmanjšujemo za število že obdelanih sodčkov.

```
int Cena(int poraba, int dni)
{
  int c, p, cena = 0;
  for (c = 10; c <= 13; c++) {
    p = 6 * dni; if (p > poraba) p = poraba;
    cena += p * c; poraba -= p; }
  cena += 18 * poraba; return cena;
}
```

Glavni blok programa najprej prebere porabo po mesecih, izračuna ceno po mesecih (tu kliče podprogram Cena) ter skupno ceno in porabo za celo leto. S tem smo že rešili podnalogo (a).

Malo več dela je s podnalogo (b). Elektrika bo najcenejša, če bo povprečna dnevna poraba po vseh mesecih približno enaka (če si predstavljamo porabo ilustrirano s stolpci, tako kot v besedilu naloge, je najugodnejši razpored porabe po mesecih takrat, ko so vsi stolpci enako visoki). Povprečno dnevno porabo za celo leto dobimo tako, da skupno letno porabo delimo s številom dni v letu; toda kaj če se deljenje ne izide? Če imamo na primer 10 000 sodčkov letne porabe in 365 dni v letu, je povprečje približno 27,4. Če vsak dan porabimo 27 sodčkov, bo to naneslo v celem letu 9 855 sodčkov; do 10000 nam jih manjka še 145. Primeren razpored je torej ta, da 145 dni v letu porabimo en sodček več, torej 28 sodčkov, ostale dni pa po 27 sodčkov. Načeloma je vseeno, kako te dneve z višjo porabo razporedimo po letu; spodnja rešitev jih stlači vse v prve mesece leta.

```
#include <stdio.h>
int main()
{
```

```
  const int DniVMesecu[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  int cena = 0, poraba = 0, mesec, povp, p, dni = 0;

  /* Preberimo porabo po mesecih; za vsak mesec izračunajmo ceno;
     v spremenljivkah cena in poraba hranimo skupno ceno in porabo. */
  for (mesec = 0; mesec < 12; mesec++) {
    scanf("%d", &p);
    poraba += p;
    cena += Cena(p, DniVMesecu[mesec]);
    dni += DniVMesecu[mesec]; }
  printf("Prvotna cena: %d\n", cena);

  printf("Poraba po mesecih, ki da najnižjo ceno:\n");
  /* Izračunajmo povprečno dnevno porabo čez celo leto (zaokroženo navzdol). */
  povp = poraba / dni;
  poraba -= povp * dni;

  /* Zdaj moramo v prvih „poraba“ dneh porabiti po povp + 1 sodčkov, v ostalih
     dneh pa po povp sodčkov. */
  for (mesec = 0, cena = 0; mesec < 12; mesec++) {
    p = DniVMesecu[mesec]; if (p > poraba) p = poraba;

    /* V tem mesecu bomo imeli p dni z večjo porabo (povp + 1),
       ostale dni pa bomo porabili po povp sodčkov. */
    poraba -= p;

    /* Izračunajmo zdaj skupno porabo (in ceno) za ta mesec in jo izpišimo. */
    p += DniVMesecu[mesec] * povp;
    printf("%d\n", p); cena += Cena(p, DniVMesecu[mesec]); }

  printf("Cena pri tej porabi: %d\n", cena); return 0;
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Ve,jic,e

S kazalcem p se premikajmo po nizu s in pri vsakem položaju p-ja preverimo, ali se na tem mestu v s začne pojavitev niza w1. V ta namen se z dvema kazalcema hkrati premikajmo po nizih — s pp gremo po s od položaja p naprej, z ww pa gremo po w1 od začetka naprej. Istoležne znake primerjamo in se ustavimo, ko bodisi opazimo neujemanje bodisi pridemo do konca niza w1. Razlika v primerjavi z običajnim primerjanjem nizov je le ta, da se po s ne premaknemo vedno le za en znak naprej, ampak še preskočimo vse vejice. Če smo na ta način prišli do konca niza w1, ne da bi opazili kakšno neujemanje, potem vemo, da smo našli pojavitev w1 v s. To pojavitev moramo zdaj zamenjati z w2, torej le skopiramo znake iz w2 v s (spet od položaja p) naprej, pri tem pa v s-ju preskočimo vejice, tako da bodo ostale tam, kjer so bile.

```
#include <stdio.h>
void Zamenjaj(char *s, char *w1, char *w2)
{
  char *p, *pp, *ww;
  for (p = s; *p; )
  {
    for (pp = p, ww = w1; *ww && *ww == *pp; ww++)
      do { pp++; } while (*pp == ',');
    if (*ww) { p++; continue; }
    for (ww = w2; *ww; p++)
      if (*p != ',') *p = *ww++;
  }
  printf("%s\n", s);
}
```

2. (Opomba)

Za začetek si oglejmo rešitev lažje različice naloge, pri kateri moramo zgolj pobrisati tiste dele niza, ki so vgnezdeni vsaj k nivojev globoko. Niz s glejmo od leve proti desni in si v spremenljivki globina zapomnimo trenutno globino gnezdenja oklepajev (torej koliko oklepajev je trenutno odprtih); ko naletimo na znak (, globino povečamo za 1, ko pa naletimo na znak), jo zmanjšamo za 1. Znake na globini, manjši od k, izpišemo. Paziti moramo le na to, da globino povečamo še pred izpisom znaka (, zmanjšamo pa jo po izpisu znaka), tako da se pri k-tem vgnezdenem oklepajskem paru ne bo izpisal niti oklepaj niti zaklepaj.

```
void SamoPomembno(char *s, int k)
{
  int globina = 0;
  for ( ; *s; s++)
  {
    if (*s == '(') globina++;
    if (globina <= k) fputc(*s, stdout);
    if (*s == ')') globina--;
  }
}
```

Če hočemo zdaj v to rešitev dodati še brisanje praznih oklepajev, se spremeni predvsem to, da ko naletimo na nek znak (, še ni takoj jasno, ali ga bo sploh treba izpisati ali ne (torej ali bo ta oklepaj na koncu zaradi brisanja praznih oklepajskih parov izpadel iz niza). Zato poleg spremenljivke globina vpeljimo še eno spremenljivko, globIzpisa, ki pove, do katere globine smo oklepaje že izpisali. Tisti na globinah od globIzpisa + 1 do globina pa še čakajo na to, da ugotovimo, ali jih bo treba izpisati ali pa bodo mogoče izpadli zaradi brisanja oklepajskih parov. Ko naletimo na znak, ki ni niti oklepaj niti zaklepaj in ki je na globini manj kot k, vemo, da ga bo treba izpisati, in ob tej priliki najprej izpišemo še vse doslej neizpisane oklepaje (za globine od globIzpisa + 1 do globina. Zaklepaj izpišemo le, če je bil izpisan njegov pripadajoči oklepaj (torej če je globIzpisa == globina); če ni bil, je to znak, da gre za prazen oklepajski par in torej tudi zaklepaja ne smemo izpisati.

```
void SamoPomembno2(char *s, int k)
{
  int globina = 0, globIzpisa = 0;
  for ( ; *s; s++)
  {
    if (*s == '(') globina++;
    else if (*s == ')') {
      if (globIzpisa == globina) {
        /* Pripadajoči oklepaj smo očitno izpisali, torej izpišimo tudi zaklepaj. */
        fputc(*s, stdout); globIzpisa--; }
      globina--; }
    else {
      if (globina > k) continue;
      while (globIzpisa < globina) /* Najprej izpišimo še neizpisane odprte oklepaje. */
        fputc('(', stdout), globIzpisa++;
      fputc(*s, stdout); }
  }
}
```

3. Kozarci

Naloga pravi, da ne vemo, kako je kozarec zasukan, ko ga poberemo s tekočega traku. Torej ne vemo, ali bi bilo treba prvo stranico pobarvati z barvo Barve[0] ali z Barve[1] ali Barve[2] itd. Če bi bil kozarec še popolnoma nepobarvan, bi bilo to tako ali tako vseeno; težava pa je, da so nekatere stranice mogoče že pobarvane. Ugotoviti moramo torej, za koliko stranic bi bilo treba kozarec zasukati, da bi se ga dalo od tam naprej barvati z barvami od Barve[0] naprej. Zasuk za j stranic pride v poštev, če je prva stranica kozarca ravno barve Barve[j] (ali pa nepobarvana), druga stranica barve Barve[j + 1] (ali pa nepobarvana) in tako naprej. Možni zasuki so od 0 do $n - 1$ in da ne bomo za vsakega od njih obračali kozarca za poln krog, lahko vse zasuke preverjamo istočano: v tabeli kand vodimo podatke o tem, kateri zasuki sploh še pridejo v poštev (torej: kateri so konsistentni s tem, kar smo doslej videli o trenutnem kozarcu); ko kozarec zasukamo za eno stranico naprej, pogledamo, če je ta stranica pobarvana; če je pobarvana, se sprehodimo po tabeli kand in če bi kateri od zasukov v njej zahteval za trenutno stranico neko drugo barvo namesto te, ki smo jo na kozarcu dejansko opazili, potem vemo, da tisti zasuk ni pravi. Na koncu vzamemo poljubnega od zasukov, ki je prestal to testiranje, in kozarec pobarvamo v skladu z njim; če pa ni primernega zasuka, moramo kozarec zavreči.

```
#include <stdbool.h>
int main()
{
  bool kand[n]; int i, j, barva;
  while (true)
  {
    Naslednji();
    for (i = 0; i < n; i++) kand[i] = true;
    for (i = 0; i < n; i++, Obrni()) {
      barva = Preveri();
      if (barva == 0) continue;
      /* Trenutna, i-ta stranica kozarca je pobarvana z barvo „barva“.
         Poglejmo, če je kateri od možnih zasukov nekonsistenten s tem. */
      for (j = 0; j < n; j++)
        if (kand[j] && Barve[(i + j) % n] != barva) kand[j] = false; }
    /* Poiščimo prvi primerni zasuk; če ni nobenega, kozarec zavržemo. */
    for (i = 0; i < n && ! kand[i]; i++) ;
    if (i >= n) { Zavrzi(); continue; }
    /* Kozarec pobarvamo v skladu z izbranim zasukom i. */
    for (j = 0; j < n; j++, Obrni())
      if (Preveri() == 0) /* Že pobarvane stranice pustimo pri miru. */
        Pobarvaj(Barve[(i + j) % n]);
    Koncaj();
  }
}
```

4. Telefonske številke

Nalogo lahko elegantno rešimo tako, da za vsako t_j pogledamo, koliko je v zaporedju $t_1, \ldots, t_m$ takih številk, ki se od t_j razlikujejo v eni števki, v vseh ostalih pa se ujemajo z njo. Recimo, da je $n(j, r)$ takih številk, ki se od t_j razlikujejo le v r-ti števki, v ostalih pa se ujemajo z njo.

Če je t_j ravno ena od originalnih številk s_i (naloga pravi, da se tudi vse s_i pojavljajo vsaj enkrat v $t_1, \ldots, t_m$), bomo na ta način prešteli vse številke, ki smo jih dobili, ko smo poskušali klicati s_i (pa smo se mogoče zmotili v eni števki). Vsota $1 + \sum_{r=1}^{k} n(j, r)$ je tedaj ravno velikost cele skupine (1 smo prišteli zato, ker je tudi t_j oz. s_i sama del te skupine).

Če pa t_j ni enaka nobeni od originalnih številk, je morala nastati tako, da smo poskušali klicati neko s_i in smo se pri tem zmotili v eni števki, recimo na r-tem mestu. Potem sledi, da je $n(j, r') = 0$ za vsak $r' \neq r$. Zakaj? Če bi obstajala neka $t_{j'}$, ki bi se od t_j razlikovala le na r'-tem mestu, bi se ta $t_{j'}$ razlikovala od s_i že na dveh mestih (r in r'), torej ni mogla nastati pri poskusu klica s_i; torej je nastala pri poskusu klica neke druge originalne številke, recimo $s_{i'}$; toda ker se $s_{i'}$ (kot pravi naloga) razlikuje od s_i v vsaj štirih mestih, se $t_{j'}$ razlikuje od $s_{i'}$ v vsaj dveh mestih, torej vendarle ni mogel nastati pri poskusu klica številke $s_{i'}$. Tako smo prišli v protislovje, torej vidimo, da je res $n(j, r') = 0$ za vse $r' \neq r$. Če torej za takšno t_j izračunamo vsoto $1 + \sum_{r=1}^{k} n(j, r)$, smo s tem prešteli le tiste številke, ki so nastale ob poskusu klica s_i in se od nje razlikujejo na r-tem mestu (ali pa še tam ne); prešteli smo torej nek manjši del skupine številk, ki je nastala ob poskusih klica številke s_i.

Če torej izračunamo vsoto $1 + \sum_{r=1}^{k} n(j,r)$ za vse j in vzamemo največjo izmed tako dobljenih vsot, bomo dobili ravno velikost največje skupine števil, ki so nastale iz kakšne s_i, ravno to pa je rezultat, po katerem sprašuje naloga.

Razmislimo še malo podrobneje o tem, kako za vsako t_j prešteti, koliko drugih števil v zaporedju $t_1, \ldots, t_m$ se od nje razlikuje na r-tem mestu in se z njo ujema povsod drugod. To lahko elegantno naredimo tako, da vse števile $t_1, \ldots, t_m$ leksikografsko uredimo, le da pri primerjanju števil preskočimo r-to števko. Tako bodo prišle skupaj števile, ki se ujemajo povsod razen na r-tem mestu. Še ena možnost je, da števile $t_1, \ldots, t_m$ (brez r-te števke) zložimo v razpršeno tabelo (*hash table*) ali pa v drevo (*trie*), pri čemer bodo spet prišle skupaj tiste, ki se ujemajo povsod razen na r-tem mestu.

Zapišimo postopek še s psevdokodo:

(* *v* v_j *se bo nabirala vsota* $1 + \sum_{r=1}^{k} n(j,r)$ *)
for $j := 1$ **to** m **do** $v_j := 1$;

for $r := 1$ **to** k:
  pripravi skupine števil (iz $t_1, \ldots, t_m$), ki se ujemajo
    povsod razen mogoče v r-ti števki (kot smo videli zgoraj, lahko
    to naredimo z urejanjem, razpršeno tabelo, drevesom ipd.);
  za vsako tako skupino:
    naj bo N število števil v tej skupini;
    za vsako t_j iz te skupine:
      (* *vemo, da je* $n(j,r) = N - 1$ *)
      $v_j := v_j + N - 1$;

vrni $\max\{v_1, v_2, \ldots, v_m\}$;

5. Assembler

Preprosta, a neučinkovita rešitev je, da zmnožek $x \cdot y$ računamo kot $x+x+\ldots+x+x$, torej vsoto y členov z vrednostjo x. Spodnji program to vsoto računa v spremenljivki z, ki jo v vsaki iteraciji zanke poveča za x; števec zanke pa je spremenljivka yy, ki se začne pri y in se v vsaki iteraciji zmanjša za 1, dokler ne pade na 0. Na koncu imamo torej v z ravno vrednost $x \cdot y$ in jo moramo le še skopirati v x (saj naloga pravi, da mora biti zmnožek na koncu v x).

```
   SET yy, 0
   ADD yy, y          /* yy = y */
   SET z, 0           /* z = 0 */
zacetek:
   SET temp, 0
   CMP yy, temp       /* ali je yy == 0? */
   JE konec           /* če da, končajmo */
   ADD z, x           /* z = z + x */
   SET temp, 1
   SUB yy, temp       /* yy = yy - 1 */
   CMP temp, temp
   JE zacetek         /* skočimo nazaj na začetek zanke */
konec:
   SET x, 0
   ADD x, z           /* x = z */
```

Učinkovitejša ideja je tale: y lahko vsekakor zapišemo v dvojiškem zapisu, torej kot vsoto potenc števila 2, na primer

$$1234 = 1024 + 128 + 64 + 16 + 2 = 2^{10} + 2^7 + 2^6 + 2^4 + 2^1.$$

Iz tega sledi tudi

$$x \cdot 1234 = 2^{10}x + 2^7x + 2^6x + 2^4x + 2^1x.$$

Torej, če bi znali y nekako razbiti na potence števila 2 (z drugimi besedami, če bi znali preverjati, kateri biti v dvojiškem zapisu števila y so prižgani) in izračunati še x-kratnike teh potenc, bi morali potem le še sešteti te x-kratnike. Težava te ideje je, da zbirni jezik naše naloge ne ponuja tradicionalnih operacij za delo z biti (SHL, AND in podobne). Pomagamo pa si lahko z naslednjim dejstvom: recimo, da leži y na območju $2^k \leq y < 2^{k+1}$. Potem vemo, da je bit k v dvojiškem zapisu y gotovo prižgan; vsi nižji biti skupaj imajo namreč vrednost le $2^{k-1} + \ldots + 4 + 2 + 1 = 2^k - 1$, kar je manj od y, torej bomo do y lahko prišli le, če je bit k prižgan. Ker torej vemo, da je bit k prižgan, lahko k našemu zmnožku (ki ga bomo spet računali v spremenljivki z) prištejemo $2^k x$, nato pa bit k v y ugasnemo preprosto tako, da od y odštejemo 2^k. Zdaj imamo pred sabo neko manjše število, ki ga obdelujemo naprej na enak način z manjšimi potencami števila 2.

Z operacijami, ki jih imamo na voljo, ni težko računati potenc števila 2 v naraščajočem vrstnem redu: $1 + 1 = 2$, $2 + 2 = 4$, $4 + 4 = 8$, $8 + 8 = 16$ in tako naprej. Enako velja tudi za x-kratnike teh potenc. Toda postopek, ki smo ga pravkar zasnovali, bi želel pregledovati potence v padajočem vrstnem redu. Ker nimamo pri roki inštrukcije za deljenje z 2 ali kaj podobnega, si pomagamo s tem, da števila 2^k in $2^k x$ sproti, ko jih računamo, odlagamo na sklad. Tako bodo nižje potence pri dnu sklada, višje pa pri vrhu in ko jih bomo kasneje pobirali s sklada, jih bomo dobivali v padajočem vrstnem redu, torej točno tako, kot jih potrebujemo.

Oglejmo si zdaj najprej zanko, ki izračuna dovolj potenc števila 2 (in njihovih x-kratnikov) ter jih odloži na sklad:

```
   SET p, 1          /* p = 1 */
   SET px, 0
   ADD px, x         /* px = x */
zacetek1:
   PUSH p            /* Zdaj je p neka potenca števila 2 in px = p * x. */
   PUSH px           /* Odložimo ju na sklad. */
   CMP y, p
   JL konec1         /* Če je p > y, končajmo. */
   ADD p, p          /* Podvojimo p in px. */
   ADD px, px
   CMP y, y
   JE zacetek1
konec1:
```

V resnici bi se lahko ustavili že, čim je $p \geq y$, ne šele pri $p > y$; vendar je pogoj $p > y$ malo lažje preverjati. Kakorkoli že, zdaj pride na vrsto glavna zanka, ki ugaša bite v y in računa zmnožek $x \cdot y$:

```
   SET z, 0          /* z = 0 */
   SET yy, 0
```

```
   ADD yy, y          /* yy = y */
zacetek2:
   POP px             /* Poberimo s sklada naslednjo potenco */
   POP p              /* števila 2 in njen x-kratnik. */
   CMP yy, p
   JL preskok         /* Če je yy < p, ta bit ni prižgan. */
   SUB yy, p          /* Sicer ugasnimo bit p v yy. */
   ADD z, px          /* z = z + p * x */
preskok:
   SET tmp, 1
   CMP p, tmp
   JG zacetek2        /* Če je p > 1, moramo še nadaljevati. */
   SET x, 0           /* Zdaj je z = x * y; skopirajmo */
   ADD x, z           /* ta zmnožek v x. */
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Reklama

Označimo položaj, na katerem smo izvedli predstavitev, z (x_0, y_0). Iz besedila naloge sledi, da po t dneh izvejo za naš izdelek vsa gospodinjstva (x, y), katerih koordinate ustrezajo pogoju

$$|x - x_0| + |y - y_0| \le t.$$

Upoštevajmo, da je $|a| = \max\{a, -a\}$; gornji pogoj je zato enakovreden naslednjemu:

$$\max\{x - x_0, x_0 - x\} + \max\{y - y_0, y_0 - y\} \le t,$$

tega pa lahko razbijemo na štiri dele, odvisno od predznaka števil $x - x_0$ in $y - y_0$:

$$\begin{aligned} x - x_0 + y - y_0 &\le t \\ x - x_0 - y + y_0 &\le t \\ x_0 - x + y - y_0 &\le t \\ x_0 - x + y_0 - y &\le t \end{aligned}$$

Vidimo, da tu pogosto nastopajo vsote in razlike x- in y-koordinat; da si bomo te reči lažje predstavljali, pišimo $u = x + y$, $v = x - y$ in podobno za u_0 in v_0. Tako dobimo:

$$\begin{aligned} u - u_0 &\le t \\ v - v_0 &\le t \\ v_0 - v &\le t \\ u_0 - u &\le t, \end{aligned}$$

kar je isto kot

$$|u - u_0| \le t \quad \text{in} \quad |v - v_0| \le t.$$

Mi bi seveda radi izbrali u_0, v_0 in t tako, da bodo ti pogoji izpolnjeni za vsa gospodinjstva naših potencialnih kupcev (in da bo t pri tem čim manjši). Naj bo (x_i, y_i) položaj i-tega izmed teh gospodinjstev in naj bo $u_i = x_i + y_i$, $v_i = x_i - y_i$. Ko si enkrat izberemo u_0 in v_0 (torej položaj predstavitve), lahko najmanjši primeren t dobimo tako, da poiščemo $d_u(u_0) := \max_i |u_i - u_0|$ in $d_v(v_0) := \max_i |v_i - v_0|$ in vzamemo večjega od njiju: $t(u_0, v_0) := \max\{d_u(u_0), d_v(v_0)\}$.

Načeloma bi lahko najmanjšo vrednost $d_u(u_0)$ dobili, če poiščemo najmanjši in največji u_i (recimo jima u_{min} in u_{max}) ter postavimo u_0 ravno na pol poti med njima, torej $u_0 = (u_{min} + u_{max})/2$. Zanj je $d_u(u_0)$ enako ravno $(u_{max} - u_{min})/2$. Če tako dobljeni u_0 ni celo število, je vseeno, ali ga zaokrožimo navzdol ali navzgor, saj bomo v vsakem primeru dobili $d_u(u_0) = \lceil (u_{max} - u_{min})/2 \rceil$. (Primer: če imamo $u_{min} = 5$, $u_{max} = 10$, lahko $(5 + 10)/2 = 7{,}5$ zaokrožimo bodisi na 7 bodisi na 8, pa bomo v vsakem primeru do enega od krajišč, bodisi u_{min} bodisi u_{max}, oddaljeni za $\lceil (10 - 5)/2 \rceil = 3$.) Enak razmislek lahko seveda ponovimo tudi za v-koordinate in tako dobimo še v_0.

Težava pa je, da je naša prvotna mreža (x, y) diskretna, koordinate x in y so cela števila, zato ni vsak par koordinat (u, v) veljaven. Iz $u - v = (x + y) - (x - y) = 2y$ sledi, da je razlika $u - v$ vedno soda, torej sta u in v enake parnosti. Pri naši ideji, da bi vzeli $u_0 = (u_{min} + u_{max})/2$ in $v_0 = (v_{min} + v_{max})/2$, pa se čisto lahko zgodi, da

dobimo u_0 in v_0 različne parnosti. V prvotnem koordinatnem sistemu ustreza taki točki par ne-celih koordinat (x, y). (Na primer: če je $u_{min} = 3$, $u_{max} = 7$, $v_{min} = 0$, $v_{max} = 4$, dobimo $u_0 = 5$, $v_0 = 2$, kar ustreza $x_0 = 3{,}5$, $y_0 = 1{,}5$.)

V primerih, ko smo imeli neceloštevilski $(u_{min} + u_{max})/2$, se tej težavi zlahka izognemo: odvisno od tega, ali to vrednost zaokrožimo navzdol ali navzgor, dobimo enkrat sodo število, enkrat pa liho (obe izbiri pa nam dasta enako $d_u(u_0)$, torej sta v tem smislu obe enako dobri); od njiju torej za u_0 vzemimo tisto, ki se po parnosti ujema z v_0. Podobno lahko razmišljamo, če imamo takšno možnost izbire glede zaokrožanja pri v_0 namesto pri u_0.

Ostane nam še primer, ko sta tako $u_0 = (u_{min} + u_{max})/2$ kot $v_0 = (v_{min} + v_{max})/2$ že celi števili, vendar različne parnosti. V tem primeru nam ne ostane drugega, kot da eno od njiju povečamo ali zmanjšamo za 1. Recimo, da to naredimo z u_0; njegova vrednost $d_u(u_0)$ zdaj ni več enaka $(u_{max} - u_{min})/2$, pač pa $(u_{max} - u_{min})/2 + 1$. Podobno je, če za 1 povečamo ali zmanjšamo vrednost v_0. Ker bi radi, da je večja od vrednosti $d_u(u_0)$ in $d_v(v_0)$ čim manjša (to je namreč $t(u_0, v_0)$ — število dni, v katerem bodo obveščeni vsi potencialni kupci), popravimo raje tisto koordinato, pri kateri je bila ta vrednost doslej manjša; torej, če je bilo $d_u(u_0) < d_v(v_0)$, popravimo raje u_0, koordinato v_0 pa pustimo pri miru; tako bo $d_v(v_0)$ po novem še vedno $\geq d_u(u_0 \pm 1)$, torej bo $t(u_0 \pm 1, v_0) = \max\{d_u(u_0 \pm 1), d_v(v_0)\} = d_v(v_0) = \max\{d_u(u_0), d_v(v_0)\} = t(u_0, v_0)$; z drugimi besedami, rešitev se ne bo zaradi tega popravka celo nič poslabšala. Podobno lahko razmišljamo, če je bilo $d_u(u_0) > d_v(v_0)$, le da takrat pač raje premaknemo v_0 namesto u_0. Če pa je veljalo $d_u(u_0) = d_v(v_0)$, je zdaj vseeno, ali premaknemo u_0 ali v_0, saj se bo $t(u_0, v_0)$ v vsakem primeru povečal za 1.

```
#include <stdio.h>
int main()
{
  int h, w, k, x, y, u, v, uMin, uMax, vMin, vMax, i, t;
  /* Preberimo velikost mreže in število točk. */
  FILE *f = fopen("reklama.in", "rt");
  fscanf(f, "%d %d %d", &h, &w, &k);
  /* Preberimo koordinate točk, jih preračunajmo v koordinatni sistem (u, v)
     in poiščimo največjo in najmanjšo koordinato na vsaki osi. */
  for (i = 0; i < k; i++)
  {
    fscanf(f, "%d %d", &y, &x);
    u = x + y; v = x - y;
    if (i == 0) uMin = u, uMax = u, vMin = v, vMax = v;
    if (u < uMin) uMin = u; if (u > uMax) uMax = u;
    if (v < vMin) vMin = v; if (v > vMax) vMax = v;
  }
  fclose(f);
  /* Čas obveščanja je načeloma ceil(max(uMax - uMin, vMax - vMin)/2), ki je dosežen,
     če predstavitev izvedemo v u0 = (uMax - uMin) / 2, v0 = (vMax - vMin) / 2. */
  t = uMax - uMin; if (vMax - vMin > t) t = vMax - vMin;
  /* Posebej pa moramo paziti na primere, ko tako dobljen par (u0, v0) ni veljaven,
     vsi njegovi veljavni sosedje pa dajo za 1 dan daljši čas obveščanja. */
  if (uMax - uMin == vMax - vMin &&
      abs(uMax - uMin) % 2 == 0 && abs(uMin - vMin) % 2 == 1) t++;
```

```
  /* Izpišimo rezultat. */
  f = fopen("reklama.out", "wt");
  fprintf(f, "%d\n", (t + 1) / 2); fclose(f); return 0;
}
```

2. Kompresija slike

Recimo, da smo si naših k odtenkov sive že nekako izbrali in jih oštevilčili v naraščajočem vrstnem redu: $0 \le b_1 < b_2 < \ldots < b_k < B$, pri čemer je $B = 1001$ število vseh odtenkov v našem barvnem prostoru. Naloga pravi, da moramo minimizirati vsoto absolutnih vrednosti napak, torej bo najbolje, če barvo vsakega piksla zamenjamo z najbližjo b_i. Če leži neka barva b točno na pol poti med b_i in b_{i+1}, je vseeno, ali piksle barve b zamenjamo z b_i ali z b_{i+1}; lahko bi celo zamenjali nekatere z eno in nekatere z drugo, ampak od tega ne bi bilo nobene posebne koristi;[5] dogovorimo se na primer, da bomo v takem primeru vedno uporabili temnejšega od obeh odtenkov, torej b_i.

Vidimo torej, da se lahko pri naši kompresiji omejimo tako, da (za vsak b) vse piksle barve b prebarvamo z istim barvnim odtenkom, namreč s tistim b_k, ki minimizira $|b - b_i|$. Ker se vsi piksli barve b preslikajo v isti odtenek b_i, je za potrebe našega razmisleka pravzaprav vseeno, kateri piksli so to oz. kje na sliki ležijo; pomembno je le to, koliko jih je. Iz vhodne slike je zato koristno izračunati histogram h — to je tabela z B celicami, v kateri element h_b pove, koliko pikslov na sliki je barve b.

Iz dosedanjega razmisleka tudi vidimo, da bo v komprimirani sliki vsak odtenek b_i pokril nek strnjen interval barv iz prvotne slike (načeloma vse tiste b, za katere je $(b_{i-1} + b_i)/2 < b \le (b_i + b_{i+1})/2$; posebej moramo paziti le na spodnjo mejo pri b_1 in na zgornjo mejo pri b_k). Označimo te meje med intervali s c_i; imamo torej $0 = c_0 \le c_1 \le c_2 \le \ldots \le c_{k-1} \le c_k = B$ in ta števila nam povedo, da se pri kompresiji v odtenek b_i preslikajo vsi piksli, katerih barva b leži na intervalu $c_{i-1} \le b < c_i$.

Namesto da iščemo najboljši nabor odtenkov $(b_1, \ldots, b_k)$, bi lahko torej rekli, da iščemo najboljše razbitje intervala $0, \ldots, B-1$ na k podintervalov; oz. z drugimi besedami, da iščemo najboljše zaporedje mej $(c_0, c_1, \ldots, c_{k-1}, c_k)$. Med vsemi temi razbitji na podintervale je gotovo tudi tisto, ki ustreza najboljšemu možnemu naboru odtenkov, tako da, če bomo našli najboljše razbitje, bomo s tem dobili najboljšo rešitev naloge.

Recimo zdaj, da imamo v mislih neko konkretno razbitje na podintervale, in si oglejmo v njem nek konkreten interval, na primer kar tistega najsvetlejšega, od c_{k-1} do $c_k - 1$ (ki se bo pri kompresiji preslikal v odtenek b_k). Za vsako barvo b s tega intervala (torej $c_{k-1} \le b < c_k$) imamo na sliki h_b pikslov te barve in vsak od njih prispeva k napaki naše komprimirane predstavitve slike vrednost $|b - b_k|$; skupaj to nanese $\sum_{b=c_{k-1}}^{c_k-1} h_b|b - b_k|$. Vidimo torej, da na napako pri teh pikslih čisto nič ne vplivajo ostali odtenki $(b_1, \ldots, b_{k-1})$; še več, na napako pri teh pikslih ne vpliva niti to, kako smo na podintervale razbili preostale barve (tiste od 0 do $c_{k-1}-1$). Podobno tudi izbor odtenka b_k nič ne vpliva na napako pri pikslih barv od 0 do $c_{k-1} - 1$; z drugimi besedami, ko si enkrat izberemo c_{k-1} (drugo krajišče pa je tako ali tako

[5] Vsaj ne z vidika minimizacije takšne mere napake, kot jo določa besedilo naloge. Lahko pa bi bilo to koristno v smislu tega, da bi bila za človeškega opazovalca slika po kompresiji videti čim bolj podobna tisti pred kompresijo.

fiksirano: $c_k = B$), lahko izberemo zanj najboljši b_k neodvisno od tega, kako bomo izbrali $c_1, \ldots, c_{k-2}$. Podobno tudi na napako pri pikslih barv od 0 do $c_{k-1} - 1$ nič ne vpliva to, kaj smo si mi izbrali za b_k. Ker vnaprej ne vemo, kateri c_{k-1} bo pripeljal do najmanjše napake, bomo morali pač preizkusiti vse možnosti. Tako smo prišli do naslednje rekurzivne rešitve:

algoritem NajboljšeRazbitje(c_k, k):
(* *Izračuna napako najboljšega razbitja intervala* $0, \ldots, c_k - 1$ *na* k *podintervalov.* *)
1 za vsak c_{k-1} od 0 do $c_k - 1$:
2 $E[c_{k-1}] := \infty$;
3 za vsak b_k od c_{k-1} do $c_k - 1$:
4 izračunaj napako $E[c_{k-1}, b_k] := \sum_{b=c_{k-1}}^{c_k-1} h_b|b - b_k|$;
5 če je ta napaka manjša od $E[c_{k-1}]$, jo shrani v $E[c_{k-1}]$;
6 $E[c_{k-1}] := E[c_{k-1}] +$ NajboljšeRazbitje($c_k, k - 1$);
7 med vsemi tako dobljenimi $E[c_{k-1}]$ vrni najmanjšo;

Posebej bi morali paziti še na robni primer: ko je $k = 1$, torej ko hočemo en sam interval, ne res razbitja na več podintervalov, pride v poštev le $c_{k-1} = 0$, saj takrat ni nobenih drugih podintervalov, ki bi jim lahko prepustili barve pod c_{k-1}.

Ta rešitev torej deluje tako, da preizkusi vse možnosti glede levega krajišča zadnjega (najsvetlejšega) podintervala, torej c_{k-1}; pri vsaki c_{k-1} poišče najmanjšo napako za dobljeni podinterval (v ta namen preizkusi vse možne b_k in vzame tistega z najmanjšo napako) in najmanjšo napako za preostale barve (od 0 do $c_{k-1} - 1$, če se jih razbije na $k - 1$ podintervalov; v ta namen uporabimo rekurzivni klic); na koncu pa uporabi tisto c_{k-1}, ki je dala najmanjšo skupno napako.

Opisana rešitev je pravilna, vendar neučinkovita; na srečo pa je ni pretežko predelati v učinkovito rešitev. Za začetek opazimo, da če se naš podprogram kliče večkrat za isti par (B, k), bo vračal vedno enake rezultate, torej bi si bilo koristno že izračunane rezultate shranjevati v neki tabeli, da ne bomo računali istih stvari po večkrat. Opazimo lahko celo, da ko računamo rezultate za nek k, potrebujemo le rezultate za $k - 1$, ne pa tudi tistih za $k - 2$, $k - 3$ in tako naprej — tiste lahko torej sproti pozabljamo in tako prihranimo še nekaj pomnilnika. Tako imamo zdaj rešitev, ki izvede izračun za vsak par (c_k, k) le enkrat, pri tem pa ima $O((c_k)^3)$ dela (dve gnezdeni zanki v vrsticah 1 in 3 ter še tretja, ki je skrita v vsoti v vrstici 4); ker gre lahko c_k do B, je skupna časovna zahtevnost te rešitve $O(B^4 k)$.

Naslednji pomembni prihranek pa je povezan z iskanjem najboljšega b_k. Recimo, da je na našem intervalu od c_{k-1} do $c_k - 1$ skupaj N pikslov, od česar jih je n svetlejših od b_k in n' temnejših od b_k. Imamo torej $N = \sum_{b=c_{k-1}}^{c_k-1} h_b$ in $n = \sum_{b=b_k+1}^{c_k-1} h_b$ in seveda $n' = N - n - h_{b_k}$. Kaj se zgodi z napako $\sum_b h_b|b - b_k|$, če povečamo b_k za 1? Pri vseh pikslih, ki so svetlejši od (stare) b_k, se napaka zmanjša za 1; pri vseh, ki so bili prej barve b_k ali temnejše, pa se poveča za 1. Napaka se torej skupno zmanjša za $n - (N - n) = 2n - N$; če je to > 0, je nova vrednost b_k boljša od prejšnje. Podoben razmislek pokaže, da če b_k zmanjšamo za 1, se napaka skupno zmanjša za $n' - (N - n') = 2n' - N$. Najboljši možni b_k bomo seveda prepoznali po tem, da noben od teh dveh premikov ne zmanjša njegove napake, saj je imel že prej najmanjšo možno napako. Za najboljši b_k torej velja, da je $2n - N \leq 0$, pa

tudi $2n' - N \le 0$. Iz prvega dobimo $n \le N/2$, iz drugega pa $n' \le N/2$, kar je (če vstavimo $n' = N - n - h_{b_k}$) isto kot $n + h_{b_k} \ge N/2$. Z drugimi besedami, najboljši b_k je ravno mediana barve vseh pikslov z intervala od c_{k-1} do $c_k - 1$: kvečjemu polovica teh pikslov sme biti svetlejša od b_k, kvečjemu polovica sme biti temnejša od b_k.

Zanko v vrsticah 3 in 4 gornjega postopka (pravzaprav dve zanki, ker se ena skriva še v izračunu vsote v vrstici 4) bi torej lahko nadomestili s preprostejšo in učinkovitejšo zanko, ki bi poiskala mediano. Lepo pri tem je, da če se na primer b_k premakne za 1, se n in n' le malo spremenita (če se b_k zmanjša za 1, se n poveča za h_b), tako da ju ni treba računati vsakič znova. Vrstice 2–5 moramo zamenjati z nečim takšnim:

$N := 0$; za vsak b od c_{k-1} do $c_k - 1$: $N := N + h_b$;
$b_k := c_k - 1$; $n := 0$;
while $b_k \ge c_{k-1}$:
  $n := n + h_{b_k}$;
  if $n + h_{b_k} \ge N/2$ then (* *našli smo mediano* *) break;
  $b_k := b_k - 1$;
(* *Zdaj poznamo mediano b_k in lahko izračunamo njeno napako.* *)
$E[c_{k-1}] := \sum_{b=c_{k-1}}^{c_k-1} h_b |b - b_k|$;

Naša nova zanka ima časovno zahtevnost $O(B)$; zunanja zanka (tista iz vrstice 1) pa ostane in postopek kot celota (ko ga izvedemo za vse pare (c_k, k)) ima zdaj časovno zahtevnost $O(B^3 k)$.

Dobljeno rešitev lahko še malo izboljšamo. Spomnimo se, da v zunanji zanki pregledamo vse možne c_{k-1}, od 0 do $c_k - 1$. Pri vsakem izračunamo b_k kot mediano barve vseh pikslov z intervala $c_{k-1}, \ldots, c_k - 1$. Toda mediane, ki jih dobimo pri različnih c_{k-1}, so si med seboj tesno povezane. Recimo, da že poznamo pravo mediano pri neki vrednosti c_{k-1}; kaj se zgodi, če c_{k-1} zmanjšamo za 1? Z drugimi besedami, kaj se zgodi, če v interval barv, ki naj bi ga pokril odtenek b_k, dodamo še eno malo temnejšo barvo? Nova mediana po tem seveda ne more biti svetlejša od stare — lahko je kvečjemu enaka ali temnejša. Vsota N se poveča na $N' := N + h_{c_{k-1}-1}$, vsota n pa se nič ne spremeni (če ne spremenimo b_k). Ker je bila b_k prej prava mediana, velja zanjo $n + h_{b_k} \ge N/2$ in $n \le N/2$; zdaj bi želeli, da bi veljalo isto za N' namesto N. Ker je $n \le N/2$ in $N' \ge N$, velja tudi $n \le N'/2$. Za $n + h_{b_k}$ pa ni nujno, da je $\ge N'/2$; in če ni, moramo b_k malo zmanjšati: pri tem se bo n malo povečal in prej ali slej bo pogoj $n + h_{b_k} \ge N'/2$ izpolnjen. Tako smo dobili naslednji postopek:

$b_k := c_k$; $n := 0$; $N := 0$;
za vsak c_{k-1} od $c_k - 1$ do 0 (v padajočem vrstnem redu):
  $N := N + h_{c_{k-1}}$;
  while $2(n + h_{b_k}) < N$:
    $n := n + h_{b_k}$; $b_k := b_k - 1$;
  (* *Zdaj je b_k mediana barve vseh pikslov z intervala od c_{k-1} do $c_k - 1$.* *)

Obe zanki — zunanja, ki zmanjšuje c_{k-1}, in notranja, ki zmanjšuje b_k, sta zdaj prepleteni; ko zunanja opravi vseh c_k iteracij, opravi tudi notranja največ toliko iteracij, saj se b_k pri njej ves čas zmanjšuje (od začetne vrednosti c_k do neke končne

vrednosti, ki je nekje od 0 do c_k). Zato imata obe zanki skupaj časovno zahtevnost le $O(c_k)$.

Toda račun mediane nam še ni dovolj, mi bi radi pri vsakem c_{k-1} izračunali tudi vsoto absolutnih vrednosti napak, ki jo dobimo, če vse barve z intervala $c_{k-1}, \ldots, c_k - 1$ predstavimo z njihovo mediano. Če bomo za to pri vsakem c_{k-1} izvedli še eno vgnezdeno zanko, ki bo računala vsoto $E[c_{k-1}] = \sum_{b=c_{k-1}}^{c_k-1} h_b|b - b_k|$, bomo pristali pri enaki časovni zahtevnosti kot prej: $O(B^2)$ za vsak par (c_k, k) in zato $O(B^3k)$ za celoten postopek. Na srečo lahko tudi to počnemo učinkoviteje. Recimo, da že poznamo napako za neko konkretno vrednost c_{k-1} in b_k; v našem postopku se ti dve spremenljivki občasno zmanjšata za 1 in ugotoviti moramo, kako takrat čim ceneje izračunati novo napako. Če se c_{k-1} zmanjša za 1 (njegovi novi vrednosti recimo c'_{k-1}, dobi ta vsota nov člen z vrednostjo $h_{c'_{k-1}}|c'_{k-1} - b_k|$. Če pa se b_k zmanjša za 1, se zgodi naslednje: napaka pri vseh n pikslih, ki so bili že prej svetlejši od mediane, se zdaj poveča za 1 (ker je nova mediana še temnejša od prejšnje); napaka pri vseh n' pikslih, ki so bili prej temnejši od mediane, pa se zdaj zmanjša za 1. Napaki moramo torej preprosto prišteti $n - n'$. Tako torej vidimo, da nam računanje nove napake po vsaki spremembi c_{k-1} in b_k vzame le $O(1)$ časa; pri vsakem paru (c_k, k) porabimo torej le $O(c_k)$ časa in postopek kot celota reši nalogo v $O(B^2k)$ časa.

Zapišimo dobljeni postopek še v C-ju:

```
#include <stdio.h>
#define B 1000
int main()
{
  int w, h, k, b, bb, i, kk, N, n, m, d, kand;
  int hist[B + 1], ff[2][B + 1], *f0, *f1;
  /* Preberimo vhodno datoteko. */
  FILE *f = fopen("slika.in", "rt"); fscanf(f, "%d %d %d", &h, &w, &k);
  for (b = 0; b <= B; b++) hist[b] = 0;
  for (i = 0; i < w * h; i++) { fscanf(f, "%d", &b); hist[b]++; }
  fclose(f);
  for (b = 0; b <= B; b++) ff[0][b] = 0;

  for (kk = 1; kk <= k; kk++) {
    f0 = &ff[(kk - 1) % 2][0]; f1 = &ff[kk % 2][0];
    /* Zdaj bomo reševali podprobleme s kk odtenki. Rezultate bomo
       shranjevali v tabelo f1, rešitve podproblemov s kk - 1 odtenki
       pa imamo že izračunane v tabeli f0. */
    for (b = 0; b <= B; b++)
    {
      /* Podproblem: radi bi predstavili barve 0, ..., b s samo kk odtenki.
         Pri tem bo odtenek kk pokril barve bb, ..., b. */
      m = b; /* Mediana barv pikslov z območja bb, ..., b. */
      N = 0; /* Število pikslov na območju od bb, ..., b. */
      n = 0; /* Število pikslov na območju m + 1, ..., b. */
      d = 0; /* Vsota absolutnih vrednosti napak za območje od bb do b. */
      f1[b] = (w + 1) * (h + 1) * (B + 1);
      for (bb = b; bb >= 0; bb--)
      {
        N += hist[bb]; d += (m - bb) * hist[bb];
        while (2 * (n + hist[m]) < N) { /* Popravimo mediano. */
```

```
            n += hist[m]; m--;
            d = d - (N - n) + n; }
          if (bb > 0 && kk == 1) continue;
          kand = (bb == 0 ? 0 : f0[bb - 1]) + d;
          if (kand < f1[b]) f1[b] = kand;
        } /* for bb */
      } /* for b */
    } /* for kk */

    /* Izpišimo rezultat. */
    f = fopen("slika.out", "wt"); fprintf(f, "%d\n", ff[k % 2][B]); fclose(f); return 0;
}
```

3. Konstrukcija grafa

Radi bi sestavili usmerjen graf, v katerem je od točke s do točke t natanko n različnih sprehodov. Za začetek se prepričajmo, da ni nobene koristi od tega, da bi imeli v grafu kakšne cikle. Če je v grafu kakšen tak cikel, ki vsebuje kakšno točko, ki leži na neki poti od s do t, lahko v to pot vključimo še enega ali več obhodov po tem ciklu in tako dobimo neskončno različnih sprehodov od s do t; naloga pa zahteva le n takih sprehodov. Če pa v grafu sicer je nek cikel, vendar nobena njegova točka ne leži na kakšni poti od s do t, potem je z vidika našega problema vseeno, če tega cikla sploh ne bi bilo (pravzaprav lahko iz grafa pobrišemo vse točke, ki niso dosegljive iz s ali pa iz njih ni dosegljiva t).

Omejimo se torej na aciklične grafe. Naj bo $f(u)$ število različnih poti od u do t; očitno je $f(t) = 1$ (pot dolžine 0 od t do t) in $f(u) = \sum_v f(v)$, pri čemer gre vsota po vseh takih v, za katere obstaja neposredna povezava $u \to v$.

Največ poti bomo torej dobili, če vključimo v graf kar vse možne povezave. Poleg točke t imejmo še zaporedje točk $u_0, u_1, u_2, \ldots, u_k$ in povezave $u_i \to u_j$ za vsak par točk, pri katerem je $0 \le j < i \le k$. Poleg tega imejmo še povezavo $u_i \to t$ za vsak $i = 0, \ldots, k$.

Tako smo dobili graf s $k + 2$ točkami in hitro se lahko prepričamo, da je v njem natanko 2^i poti od točke u_i do točke t. Število n lahko zapišemo kot vsoto nekaj različnih potenc števila 2 — to je pravzaprav isto, kot če bi ga pretvorili v dvojiški zapis. Na primer:

$$1234 = 1024 + 128 + 64 + 16 + 2 = 2^{10} + 2^7 + 2^6 + 2^4 + 2^1.$$

Torej, če ustanovimo novo točko s in iz nje potegnemo neposredne povezave do u_{10}, u_7, u_6, u_4 in u_1, bo od s do t natanko 1234 različnih poti. Pri naši nalogi gre lahko n do 10^8, kar je med 2^{26} in 2^{27}, torej nam bodo zadoščale potence do 2^{26}. Tako dobimo graf s točkami $t, u_0, u_1, \ldots, u_{26}$ in s; to je skupaj 29 točk, kar je še v okviru predpisanih omejitev.

Paziti moramo še na številčenje točk; naloga predpisuje, da mora biti $s = 1$ in $t = 2$. Za ostale točke je vseeno, v kakšnem vrstnem redu jih oštevilčimo; spodnji program točki u_i pripiše številko $3 + i$.

```
#include <stdio.h>

#define MaxN 100000000
#define m 26 /* floor(log2 MaxN) */
```

```
int main()
{
  int n, i, j;
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("graf.in"); fscanf(f, "%d", &n); fclose(f);
  /* Dodajmo povezave u_i → t in u_i → u_j za i > j. */
  f = fopen("graf.out", "wt");
  for (i = 3; i <= 3 + m; i++) for (j = 2; j < i; j++) fprintf(f, "%d %d\n", i, j);
  /* Zdaj je od u_i = 3 + i do t = 2 možnih natanko 2^i poti. */
  if (n > 0) {
    /* Sestavimo n iz potenc števila 2 in dodajmo povezave od s = 1 do ustreznih u_i. */
    for (i = 0; i <= m; i++) if (n & (1 << i)) fprintf(f, "%d %d\n", 1, 3 + i); }
  fclose(f); return 0;
}
```

4. Mušji drekci

Imamo torej n točk in radi bi jih pokrili z dvema kvadratoma $a \times a$, katerih stranice so vzporedne koordinatnima osema. Poiščimo med temi točkami najbolj levo in ji recimo L (s koordinatama (x_L, y_L); če je več najbolj levih, torej takih z minimalno x-koordinato, je vseeno, katero od njih vzamemo za L); podobno poiščimo tudi najbolj desno točko D, najbolj spodnjo S in najbolj zgornjo Z.

Če se našo množico točk dá pokriti z dvema kvadratoma, mora eden od teh kvadratov seveda pokriti tudi točko L; recimo mu *prvi kvadrat*. (Če oba kvadrata pokrivata L, si pač izberimo za prvi kvadrat poljubnega od njiju.)

(1) Mogoče pokrije prvi kvadrat tudi točko S. Nobene koristi ni od tega, da bi bil levi rob kvadrata levo od x_L (ker tam ni nobene točke) ali da bi bil spodnji rob kvadrata nižje od y_S (ker tudi tam ni nobene točke). Torej se lahko omejimo na primere, ko je levi rob prvega kvadrata natanko na x_L, njegov spodnji rob pa natanko na y_S. S tem je položaj prvega kvadrata že popolnoma natančno določen in ni težko preveriti, katere točke bi tak kvadrat dejansko pokril. Ob tem bomo tudi videli, katerih točk ne pokrije; med temi lahko spet poiščemo minimalno in maksimalno x- in y-koordinato in vidimo, če bi se dalo vse te preostale točke pokriti z enim samim kvadratom (veljati mora $x_{max} - x_{min} \leq a$ in $y_{max} - y_{min} \leq a$).

(2) Mogoče prvi kvadrat ne pokrije točke S, pokrije pa točko Z. Zdaj lahko razmišljamo analogno kot v prejšnjem odstavku; zgornji rob prvega kvadrata postavimo na y_Z, pogledamo, katere točke pokrije, in preverimo, če je mogoče vse preostale točke pokriti z enim samim kvadratom.

(3) Mogoče prvi kvadrat ne pokrije niti S niti Z, pokrije pa točko D. Torej mora točki S in Z pokriti drugi kvadrat. Ker prvi kvadrat pokrije L in D, mora biti $x_D - x_L \leq a$; in ker drugi kvadrat pokrije S in Z, mora biti $y_Z - y_S \leq a$; in ker so x-koordinate vseh točk na intervalu $[x_L, x_D]$, njihove y-koordinate pa na intervalu $[y_S, y_Z]$, sledi, da bi se dalo vse te točke pokriti že z enim samim kvadratom. Torej lahko primer (3) ignoriramo, ker se bo dalo takšne razporede točk pokriti tudi tako, da bo prvi kvadrat pokril L in S (in sploh vse točke), kar smo že obravnavali pod (1).

(4) Mogoče prvi kvadrat ne pokrije niti S niti Z niti D. Torej mora te tri točke pokriti drugi kvadrat. S podobnim razmislekom kot pri (1) vidimo, da lahko

postavimo desni rob drugega kvadrata na x_D, njegov spodnji rob na y_S in nato preverimo, katere točke pokrije in ali je mogoče vse preostale točke pokriti z enim samim kvadratom.

Tako torej vidimo, da če je našo množico točk sploh mogoče pokriti z dvema kvadratoma, jo je mogoče pokriti na enega od načinov (1), (2) ali (4), in videli smo tudi, kako lahko za vsakega od teh treh načinov preverimo, ali je pri dani množici točk res mogoč ali ne. Če nam na nobenega od teh treh načinov ne uspe pokriti vseh točk, lahko zaključimo, da jih s samo dvema kvadratoma ni mogoče pokriti.

```
#include <stdio.h>
#include <stdbool.h>
#define MaxN 1000000
#define MaxKoord 1000000000
int n, a, xs[MaxN], ys[MaxN];

/* Preveri, ali je mogoče vse točke pokriti z dvema kvadratoma,
   pri čemer ima prvi kvadrat levi rob x0 in spodnji rob y0. */
bool Test(int x0, int y0)
{
  int x1 = MaxKoord, y1 = MaxKoord, x2 = 0, y2 = 0, i;
  for (i = 0; i < n; i++) {
    /* Če točko i pokrije že prvi kvadrat, jo preskočimo. */
    if (x0 <= xs[i] && xs[i] - x0 <= a)
      if (y0 <= ys[i] && ys[i] - y0 <= a) continue;
    /* Ostanejo točke, ki jih bo moral pokriti drugi kvadrat.
       Zapomnimo si najmanjšo in največjo x- in y-koordinato. */
    if (xs[i] < x1) x1 = xs[i]; if (xs[i] > x2) x2 = xs[i];
    if (ys[i] < y1) y1 = ys[i]; if (ys[i] > y2) y2 = ys[i]; }
  /* Preverimo, ali je mogoče vse preostale točke pokriti z enim kvadratom. */
  return (y2 - y1 <= a && x2 - x1 <= a);
}
int main()
{
  int c, i, L, D, S, Z;
  FILE *f = fopen("musji.in", "rt"), *g = fopen("musji.out", "wt");
  fscanf(f, "%d", &c);
  while (c-- > 0)
  {
    fscanf(f, "%d", &n); fscanf(f, "%d", &a);
    /* Preberimo koordinate točk in si zapomnimo indeks najbolj
       leve, najbolj desne, najbolj spodnje in najbolj zgornje. */
    for (i = 0, L = 0, D = 0, S = 0, Z = 0; i < n; i++) {
      fscanf(f, "%d %d", &xs[i], &ys[i]);
      if (xs[i] < xs[L]) L = i; if (xs[i] > xs[D]) D = i;
      if (ys[i] < ys[S]) S = i; if (ys[i] > ys[Z]) Z = i; }
    /* Preverimo, če se da vse točke pokriti z dvema kvadratoma. */
    fprintf(g, "%s\n", Test(xs[L], ys[S]) || Test(xs[L], ys[Z] - a) ||
      Test(xs[D] - a, ys[S]) ? "DA" : "NE");
  }
  fclose(f); fclose(g); return 0;
}
```

5. Podajanje žoge

Uredimo otroke po naraščajoči x-koordinati, tiste z enako x-koordinato pa po naraščajoči y-koordinati. Tako pridejo skupaj tisti, ki bi utegnili podajati drug drugemu žogo v vodoravni smeri; za vsak par dveh zaporednih otrok v tem vrstnem redu moramo le preveriti, če imata res enako x-koordinato in če pripadata isti ekipi: če to dvoje drži, potem vemo, da lahko tadva otroka podajata žogo drug drugemu.

Za podaje v navpični smeri uporabimo enak razmislek, le vloga x- in y-koordinat je zamenjana.

Tako smo za vsakega otroka ugotovili, katerim drugim otrokom (največ štirim) lahko podaja; te podatke si zapomnimo v neki tabeli (v spodnjem programu je to sosedje). Dobili smo graf vseh možnih podaj, v njem pa nas zanima najkrajša pot od s do t; poiščemo jo lahko na primer z iskanjem v širino.

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

#define MaxN 100000

int xs[MaxN], ys[MaxN], ekipa[MaxN], is[MaxN], n, s, t, *k1, *k2;
int sosedje[MaxN][4], d[MaxN];

int Primerjaj(const void* a, const void* b)
{
  int u = *(int *) a, v = *(int *) b;
  int d = k1[u] - k1[v]; return d ? d : k2[u] - k2[v];
}

int main()
{
  int u, v, i, j, k, head, tail;
  /* Preberimo vhodne podatke. */
  FILE *f = fopen(argc > 1 ? argv[1] : "podaje.in", "rt");
  fscanf(f, "%d", &n);
  for (u = 0; u < n; u++) {
    fscanf(f, "%d %d %d", &xs[u], &ys[u], &ekipa[u]);
    for (i = 0; i < 4; i++) sosedje[u][i] = -1;
    d[u] = -1; }
  fscanf(f, "%d %d", &s, &t); s--; t--; fclose(f);

  for (k = 0; k < 2; k++) {
    /* V tabeli is pripravimo indekse otrok (od 0 do n - 1), kazalca
       k1 in k2 pa naj kažeta na koordinate, po katerih bomo urejali
       (pri k = 0 urejamo najprej po x in nato po y, pri k = 1 pa ravno obratno). */
    for (u = 0; u < n; u++) is[u] = u;
    if (k == 0) k1 = xs, k2 = ys; else k1 = ys, k2 = xs;
    qsort(is, n, sizeof(is[0]), &Primerjaj);
    /* V dobljenem vrstnem redu za vsak par sosednjih otrok poglejmo,
       če se ujemata v prvi koordinati in pripadata isti ekipi. Če je
       to dvoje res, si lahko podajata žogo, kar si bomo zapisali v tabelo sosedje. */
    for (i = 0; i < n; i++) for (j = 0, u = is[i]; j < 2; j++) {
      v = i + j * 2 - 1; if (v < 0 || v >= n) continue; else v = is[v];
      if (ekipa[v] == ekipa[u] && (xs[v] == xs[u] || ys[v] == ys[u]))
        sosedje[u][2 * k + j] = v; }}
```

```
  /* Z iskanjem v širino poiščimo najkrajšo pot od s do t.
     Tabelo is uporabljamo kot vrsto. */
  head = 0; tail = 0; is[tail++] = s; d[s] = 0;
  while (head < tail && d[t] < 0)
    /* Vzemimo naslednjo točko iz vrste; recimo ji u. Preglejmo njene sosede. */
    for (u = is[head++], i = 0; i < 4; i++) {
      /* Vemo, da je mogoče do u-ja priti v d[u] korakih; torej je mogoče do
         u-jevega soseda, v, priti v d[u] + 1 korakih. Če do v-ja še ne poznamo
         kakšne boljše poti, si to zdaj zapomnimo in ga dodajmo v vrsto. */
      v = sosedje[u][i];
      if (v < 0 || d[v] >= 0) continue;
      d[v] = d[u] + 1; is[tail++] = v; }

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

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Primerjanje oklepajev

Spodnja rešitev v zanki pregleduje oba podana niza. V vsaki iteraciji poišče naslednji oklepaj (ali zaklepaj) v nizu `a` in naslednji oklepaj (ali zaklepaj) v nizu `b` ter ju primerja med sabo. Čim opazimo kakšno neujemanje (sem šteje tudi možnost, da pridemo pri enem nizu do konca prej kot pri drugem), se ustavimo in vrnemo **false**. Niza sta si sumljivo podobna le, če smo pri obeh prišli hkrati do konca, ne da bi opazili kakšno neujemanje.

```
bool Prepisovanje(const char *a, const char *b)
{
  while (true)
  {
    /* Poiščimo naslednji oklepaj/zaklepaj v vsakem nizu. */
    while (*a && ! strchr("()[]", *a)) a++;
    while (*b && ! strchr("()[]", *b)) b++;
    /* Če smo opazili neujemanje ali prišli do konca nizov, končajmo. */
    if (*a != *b) return false;
    if (! *a) return true;
    /* Premaknimo se naprej od trenutnih oklepajev/zaklepajev. */
    a++; b++;
  }
}
```

2. Globalno segrevanje

Elegantna rešitev je, da se hkrati sprehajamo po letih in po krajih. Za vsako leto se premikamo naprej po seznamu krajev in izpisujemo kraje, ki se tisto leto potopijo; čim pa pridemo do kraja, ki leži tako visoko, da se to leto še ne potopi, lahko s trenutnim letom končamo, pri naslednjem letu pa bomo s pregledovanjem seznama krajev nadaljevali prav tu, kjer smo pri trenutnem letu končali. Postopek lahko opišemo tudi v psevdokodi:

```
postavimo se na začetek seznama krajev;
za vsako leto y od 1 do n:
  naj bo g_y gladina morja v letu y;
  izpiši 'leta y se potopijo:';
  dokler še nismo na koncu seznama krajev:
    naj bo k trenutni kraj v seznamu in naj bo h_k njegova nadmorska višina;
    if h_k ≥ g_y then break;
    izpiši kraj k in se premakni na naslednji kraj v seznamu;
```

Lepo pri tej rešitvi je, da pregleda vsak seznam (let in krajev) le po enkrat.

3. Riziko

Spodnja rešitev s petimi gnezdenimi zankami pregleda vse možne mete petih kock. Pri vsaki kombinaciji uredimo kocke vsakega igralca v padajočem vrstnem redu in nato po pravilih igre pogledamo, kakšen je izid te igre (2 : 0, 1 : 1 ali 0 : 2). Število izidov vsakega tipa hranimo v spremenljivkah `n02`, `n11` in `n20`, ki jih na koncu izpišemo.

```
#include <stdio.h>

int main()
{
  int A1, A2, A3, B1, B2, a1, a2, a3, b1, b2, t, n20 = 0, n11 = 0, n02 = 0;
  for (A1 = 1; A1 <= 6; A1++) for (A2 = 1; A2 <= 6; A2++)
  for (A3 = 1; A3 <= 6; A3++)
  for (B1 = 1; B1 <= 6; B1++) for (B2 = 1; B2 <= 6; B2++)
  {
    /* Kocke prvega igralca prepišimo iz A1, A2, A3 v a1, a2, a3 in jih uredimo padajoče.
       Prepišimo tudi kocki drugega igralca iz B1, B2 v b1, b2 in ju uredimo padajoče. */
    a1 = A1; a2 = A2; a3 = A3; b1 = B1; b2 = B2;
    if (a2 > a1) t = a1, a1 = a2, a2 = t;
    if (a3 > a2) t = a2, a2 = a3, a3 = t;
    if (a2 > a1) t = a1, a1 = a2, a2 = t;
    if (b2 > b1) t = b1, b1 = b2, b2 = t;
    /* Primerjajmo a1 in b1 ter a2 in b2
       in povečajmo ustreznega od števcev n20, n11 in n02. */
    if (a1 > b1 && a2 > b2) n20++;
    else if (a1 > b1 || a2 > b2) n11++;
    else n02++;
  }
  printf("%d %d %d\n", n20, n11, n02);
}
```

Izkaže se, da do izida $2:0$ pride v 2890 primerih, do izida $1:1$ v 2611 primerih, do izida $0:2$ pa le v 2275 primerih. V povprečju tako prvi igralec dobi v vsaki igri približno 1,08 točk, drugi pa 0,92 točk, tako da so pravila igre rahlo pristranska v korist prvega igralca.

Naloga postane še bolj zanimiva, če jo malo posplošimo: prvi igralec naj ima n kock, drugi k kock, primerjata naj največjih m kock, vsaka kocka pa naj ima s ploskev. (Prvotna naloga je imela torej $n = 3$, $k = m = 2$ in $s = 6$.) Možnih izidov igre je zdaj $m + 1$, namreč prvi igralec dobi t točk za nek $0 \le t \le m$, drugi igralec pa dobi preostalih $m - t$ točk.

Načeloma bi lahko za tako posplošeno nalogo uporabili podobno rešitev kot zgoraj, le da zdaj potrebujemo $n + k$ gnezdenih zank, vsaka od njih mora iti od 1 do s, urejanje kock posameznega igralca v padajočem vrstnem redu pa lahko preprosto in učinkovito izvedemo tako, da za vsako možno število pik (od 1 do s) preštejemo, koliko kock tega igralca ima toliko pik (*counting sort*). Če hočemo, da bo isti program uporaben za različne n in k, si ne moremo privoščiti fiksnega števila gnezdenih zank in je bolje uporabiti rekurzivni podprogram, pri katerem bo lahko globina rekurzije odvisna od vhodnih podatkov (v našem primeru n in k). Vseh možnih metov kock je zdaj s^{n+k}, pri vsakem pa imamo $O(s+n+k)$ dela, da uredimo kocke padajoče in izračunamo rezultat; časovna zahtevnost takšnega postopka je torej $O(s^{n+k}(s+n+k))$.

Ko kocke nekega igralca uredimo, lahko isto urejeno zaporedje nastane iz več različnih prvotnih neurejenih zaporedij. Na primer, pri $n = 3$ kockah s po $s = 3$ ploskvami je možnih $s^n = 27$ različnih neurejenih zaporedij, po urejanju pa nastane iz njih le 10 različnih urejenih zaporedij: 111, 112, 113, 122, 123, 133, 222, 223, 233, 333. V splošnem je različnih urejenih zaporedij pri n kockah s po s ploskvami $\binom{n+s-1}{s-1}$.[6] To je naprej enako $(n+1)\ldots(n+s-1)/(s-1)!$; če je $n \gg s$, je to približno

[6] O tem se lahko prepričamo takole. Vzemimo poljubno urejeno zaporedje kock in vanj vrinimo

$n^s/(s-1)!$ — vsekakor veliko manj od s^n. Urejenih zaporedij ni težko generirati, le meje zank moramo prilagoditi:

```
for (a1 = 1; a1 <= s; a1++)
  for (a2 = 1; a1 <= a1; a2++)
    for (a3 = 1; a1 <= a2; a3++)
```

(Podobno lahko naredimo tudi, če zaporedja pregledujemo z rekurzijo namesto z gnezdenimi zankami.) Vendar pa moramo zdaj paziti na naslednje: posamezno urejeno zaporedje lahko nastane iz enega ali več neurejenih zaporedij in to moramo pravilno upoštevati, ko štejemo izide igre. Če imamo urejeno zaporedje n kock, pri čemer ima n_i kock po i pik (za $i = 1, \ldots, s$), lahko neurejeno zaporedje iz njega sestavimo takole: najprej si moramo izmed vseh n možnih mest v zaporedju izbrati n_1 takih, kjer bodo kocke s po eno piko; nato si moramo izmed ostalih $n-n_1$ mest izbrati n_2 takih, kjer bodo kocke s po dvema pikama; itd. Tako lahko neurejeno zaporedje dobimo na $\binom{n}{n_1}\binom{n-n_1}{n_2}\ldots\binom{n-n_1-\ldots-n_{s-1}}{n_s} = n!/(n_1!n_2!\ldots n_s!)$ načinov. Postopek je torej zdaj nekako tak:

for $t := 0$ **to** m **do** *ŠteviloIzidov*$[t] := 0$;
za vsako $(a_1, \ldots, a_n)$, pri čemer je $s \geq a_1 \geq a_2 \geq \ldots \geq a_n \geq 1$:
  za vsako $(b_1, \ldots, b_k)$, pri čemer je $k \geq b_1 \geq b_2 \geq \ldots \geq b_k \geq 1$:
    $t := 0$;
    for $i := 1$ **to** m **do if** $a_i > b_i$ **then** $t := t + 1$; $\qquad(\star)$
    for $i := 1$ **to** s **do** $n_i := 0$; $k_i := 0$;
    for $i := 1$ **to** n **do** $n_{a_i} := n_{a_i} + 1$;
    for $i := 1$ **to** k **do** $k_{b_i} := k_{b_i} + 1$;
    ŠteviloIzidov[t] povečaj za $n!k!/(n_1!n_2!\ldots n_s!k_1!k_2!\ldots k_s!)$

Za vsak par urejenih zaporedij a in b torej izračunamo, koliko točk (t) bi pri tem paru dobil prvi igralec (drugi jih potem dobi $m-t$), nato pa tudi preštejemo, koliko kock s posameznim številom pik ima vsak igralec (kock z i pikami je n_i pri prvem in k_i pri drugem igralcu), tako da bomo znali število izidov primerno povečati. Pregledati moramo torej $\binom{n+s-1}{s-1}\binom{k+s-1}{s-1}$ parov (a, b), pri vsakem pa imamo načeloma $O(n + k + s)$ dela. Računanje števil n_i in k_i bi se dalo sicer skriti v zanke (oz. rekurzivne klice), ki nam generirajo zaporedja a in b, tako da bi imeli potem pri vsakem paru le še $O(m + s)$ dela. Izkaže pa se, da lahko iz n_i in k_i izračunamo število točk, t, tudi na bolj eleganten način kot z zanko po zgornjih m kockah vsakega igralca: če imamo več zaporednih indeksov i, pri katerih je vrednost a_i ves čas enaka, pa tudi vrednost b_i je ves čas enaka, potem pri vseh teh indeksih dobi točko isti igralec (prvi, če je $a_i > b_i$, sicer pa drugi), zato lahko tako skupino obdelamo v enem zamahu. Namesto zanke ($\star$) imamo zdaj takšen postopek:

$i := 0$; $p_1 :=$ s; $p_2 := s$; $u_1 := n_{p_1}$; $u_2 := k_{p_2}$;

$s-1$ „ograjic“, pri čemer i-ta ograjica ločuje kocke z i pikami od tistih z $i+1$ pikami; zdaj nam ni več treba pisati števila pik na posamezni kocki, saj je to enolično določeno že s tem, med katerima dvema ograjicama stoji ta kocka. Na ta način na primer iz zaporedja 123 nastane □|□|□, iz 133 nastane □||□□, iz 122 nastane □|□□| in podobno. Tako smo dobili bijektivno preslikavo med urejenimi zaporedji kock in nizi n simbolov □ in $s-1$ simbolov |; slednji so vsi dolgi $n+s-1$ simbolov in pri takem nizu si lahko na $\binom{n+s-1}{s-1}$ načinov izberemo, kje bodo ograjice; takih nizov je torej $\binom{n+s-1}{s-1}$, zato pa je toliko tudi urejenih zaporedij n kock s po s ploskvami.

while $i < m$:
  (* *Če zaporedji uredimo padajoče, so v prvem zaporedju na vseh indeksih od* $i+1$ *do* u_1 *kocke s po* p_1 *pikami, v drugem pa so na vseh indeksih od* $i+1$ *do* u_2 *kocke s po* p_2 *pikami.* *)
  $j := \min\{u_1, u_2\}$;
  if $p_1 > p_2$ **then** $t := t + \min\{j, m\} - i$;
  if $j = u_1$ **then** $p_1 := p_1 - 1$; $u_1 := u_1 + n_{p_1}$;
  if $j = u_2$ **then** $p_2 := p_2 - 1$; $u_2 := u_2 + k_{p_2}$;
  $i := j$;

Lepo pri tem postopku je, da porabi zdaj za obdelavo vsakega para (a, b) le še $O(s)$ časa, kar je koristno, če je $m \gg s$. Časovna zahtevnost celotnega postopka je torej $O(\binom{n+s-1}{s-1}\binom{k+s-1}{s-1}s)$, kar je približno $O(n^s k^s s)$, če je s majhen v primerjavi s številoma n in k.[7]

Kaj pa, če nas zanima le povprečno število točk prvega igralca, ne pa verjetnost vsakega od možnih izidov posebej? Izkaže se, da lahko do povprečnega števila točk pridemo precej hitreje, saj nam ne bo treba pregledovati vseh možnih zaporedij a in b. Posamezne kocke si lahko predstavljamo kot verjetnostne spremenljivke $A_1, \ldots, A_n, B_1, \ldots, B_k$, ki so med seboj neodvisne, vse pa so porazdeljene enakomerno na množici $\{1, \ldots, s\}$. Naj bo zdaj $A_{(i)}$ slučajna spremenljivka, ki pove i-to najmanjšo vrednost med števili $A_1, \ldots, A_n$; podobno definirajmo tudi $B_{(i)}$. Ko igralca primerjata i-ti največji kocki, to v bistvu pomeni, da primerjata $A_{(n+1-i)}$ in $B_{(k+1-i)}$; če je prva večja od druge, dobi točko prvi igralec, sicer pa drugi. Prvi igralec torej tu dobi točko z verjetnostjo $P(A_{(n+1-i)} > B_{(k+1-i)})$. Njegovo pričakovano število točk bomo dobili, če bomo to sešteli za $i = 1, \ldots, m$; to je zdaj rezultat, ki ga iščemo: $t(s, n, k, m) = \sum_{i=1}^{m} P(A_{(n+1-i)} > B_{(k+1-i)})$.

Pri izračunu posamezne od teh verjetnosti se lahko opremo na dejstvo, da imajo naše spremenljivke A_i, B_i (in zato tudi $A_{(i)}$, $B_{(i)}$) le s možnih vrednosti; zato je

$$\begin{aligned} P(A_{(i)} > B_{(j)}) &= \textstyle\sum_{a=1}^{s} P(A_{(i)} = a, A_{(i)} > B_{(j)}) \\ &= \textstyle\sum_{a=1}^{s} P(A_{(i)} = a, B_{(j)} < a) \\ &= \textstyle\sum_{a=1}^{s} P(A_{(i)} = a) P(B_{(j)} < a), \end{aligned}$$

pri čemer je bil zadnji korak upravičen, ker so kocke enega igralca neodvisne od kock drugega igralca. Zdaj torej vidimo, da bomo lahko $P(A_{(i)} > B_{(j)})$ izračunali v $O(s)$ korakih, če bomo poznali verjetnostno porazdelitev spremenljivk $A_{(i)}$ in $B_{(j)}$. Do tega pa ni težko priti:

$$\begin{aligned} P(A_{(i)} < a) &= P(\text{vsaj } i \text{ izmed } A_1, \ldots, A_n \text{ jih je manjših od } a) \\ &= \textstyle\sum_{j=i}^{n} P(\text{natanko } j \text{ izmed } A_1, \ldots, A_n \text{ jih je manjših od } a) \\ &= \textstyle\sum_{j=i}^{n} \binom{n}{j} P(A_i < a)^j P(A_i \geq a)^{n-j} \\ &= \textstyle\sum_{j=i}^{n} \binom{n}{j} (a-1)^j (n+1-a)^{n-j} / s^n \end{aligned}$$

[7] Pri vseh teh razmišljanjih o časovni zahtevnosti se je sicer koristno zavedati tudi tega, da cela števila, s katerimi imamo pri tej nalogi opravka, hitro postanejo zelo velika in zato posameznih aritmetičnih operacij nad njimi ne bi smeli šteti kot nečesa, kar vzame le konstantno mnogo časa.

in nato $P(A_{(i)} = a) = P(A_{(i)} < a+1) - P(A_{(i)} < a)$. Podobno lahko naredimo tudi za B-je. S tem postopkom lahko do povprečnega števila točk posameznega igralca pridemo že z $O(s(n+k))$ aritmetičnimi operacijami.

Zdaj, ko znamo precej učinkovito izračunati povprečni rezultat za poljubno kombinacijo parametrov s, n, k in m, se lahko vprašamo, kako moramo izbrati parametre, da bo igra čim bolj poštena. Videli smo na primer, da pri „standardnih“ parametrih $s = 6$, $n = 3$, $k = m = 2$, dobi prvi igralec povprečno 1,08 točk, drugi pa le 0,92. V splošnem se v vsaki igri razdeli m točk in idealno bi bilo, če bi v povprečju vsak igralec dobil po $m/2$ točk. Če dobi prvi igralec povprečno $t(s,n,k,m)$ točk, lahko „nepoštenost“ te kombinacije parametrov definiramo kot relativno odstopanje t-ja od „poštene“ vrednosti, $m/2$: $u(s,n,k,m) := (t(s,n,k,m) - m/2)/(m/2)$. V naših spodaj opisanih poskusih smo se omejili na $s = 6$, torej na običajne kocke s šestimi ploskvami.

Kombinacije parametrov lahko preiskujemo sistematično. Če fiksiramo k in m, je jasno, da postaja igra vse bolj naklonjena prvemu tekmovalcu, čim več kock ima; torej je $u(s,n,k,m)$ naraščajoča funkcija n-ja (pri fiksnih s, k in m).[8] Ker nas zanima tisti u, ki je najbližje 0 (takrat so parametri igre najbolj pošteni), moramo pravzaprav poiskati tisti n, pri katerem funkcija u preide iz negativnih vrednosti v pozitivne. Dobra hevristika je, da začnemo kar pri istem n, ki je dal najboljši rezultat pri $k-1$ in m; če je nato $u(s,n,k,m) > 0$, pregledujemo vse manjše vrednosti n, če pa je bila $u(s,n,k,m) < 0$, pregledujemo vse večje vrednosti n, dokler u ne spremeni predznaka.[9]

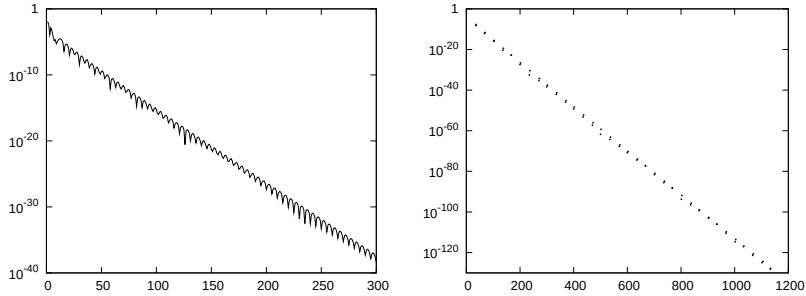
Recimo zdaj, da pregledujemo pare (k,m) po naraščajočem k, pri vsakem paru poiščemo tisti n, ki nam dá najmanjšo vrednost $|u|$, in si zapišemo tiste kombinacije, pri katerih smo dobili najmanjši $|u|$ doslej. Prvih nekaj tako dobljenih kombinacij kaže naslednja tabela:

n	k	m	$u(s,n,k,m)$	n	k	m	$u(s,n,k,m)$
2	1	1	0,157	64	18	6	$7{,}39 \cdot 10^{-5}$
3	2	2	0,079	75	21	7	$-1{,}61 \cdot 10^{-5}$
4	2	1	0,051	97	27	9	$-5{,}05 \cdot 10^{-6}$
4	3	3	0,016	44	35	32	$-2{,}54 \cdot 10^{-6}$
8	3	1	0,012	63	44	28	$-1{,}62 \cdot 10^{-6}$
7	4	2	$8{,}97 \cdot 10^{-3}$	173	48	16	$-3{,}86 \cdot 10^{-7}$
8	5	3	$3{,}62 \cdot 10^{-3}$	227	63	21	$9{,}80 \cdot 10^{-8}$
11	5	2	$-9{,}49 \cdot 10^{-4}$	324	90	30	$-3{,}38 \cdot 10^{-9}$
30	9	3	$-1{,}19 \cdot 10^{-4}$	421	117	39	$-1{,}22 \cdot 10^{-9}$

Vidimo lahko, da bi lahko originalno različico igre ($n = 3$, $k = m = 2$) naredili občutno bolj pošteno že, če bi vsakemu igralcu dodali po eno kocko ($n = 4$, $k =$

[8] To bi se dalo tudi formalno dokazati, vendar ta dokaz ni tako preprost in ga bomo tu raje izpustili. Podobno velja tudi, da je $u(s,n,k,m)$ padajoča funkcija k-ja pri fiksnih s, n in m.

[9] Pri tem moramo paziti na primere, ko u sploh nikoli ne postane večji od 0. Do tega pride, ker pravila igre določajo, da dobi prvi igralec točko le, če ima njegova kocka strogo več pik kot kocka drugega igralca; če ima torej drugi igralec maksimalno število pik, potem prvi ne more zmagati, pa če je vrgel še tako dobre kocke. Če pri fiksnih k in m pošljemo $n \to \infty$, ima prvi igralec vse več kock in največjih m med njimi ima vse bolj verjetno kar maksimalno možno vrednost, s; točke bo torej dobil povsod tam, kjer ima drugi tekmovalec kocko z manj kot s pikami. Tako vidimo, da je $\lim_{n\to\infty} t(s,n,k,m) = \sum_{i=1}^{m} P(B_{(k+1-i)} < s)$. Iz tega lahko izračunamo tudi $\lim_{n\to\infty} u(s,n,k,m)$. Pri $s = 6$ se izkaže, da je ta limita negativna za $m < k/3$; pri večjih s je ta limita večinoma negativna pri $m < 2k/s$, vendar z nekaj izjemami pri majhnih k.



Oba grafa prikazujeta $\min_n |u(s,n,k,m)|$ kot funkcijo m-ja pri $s = 6$ in $k = 3m$. Izkaže se, da je takrat minimum vrednosti $|u|$ dosežen pri $n \approx 3{,}6k$. Levi graf prikazuje $\min_n |u|$ za vse m od 1 do 300, desni pa za izbrane m-je z intervala od 1 do 1200 (tiste, pri katerih ima funkcija lokalne ekstreme).

$m = 3$). Kombinacije z večjim številom kock postanejo nepraktične, kažejo pa zelo zanimivo lastnost: dobre rezultate praviloma dobimo, ko je $k = 3m$ in $n \approx 3{,}6k$. Ta pojav velja tudi pri večjih vrednostih (n,k,m); nekaj primerov kaže naslednja tabela:

n	k	m	$u(s,n,k,m)$
993	276	92	$7{,}28 \cdot 10^{-16}$
3595	999	333	$1{,}09 \cdot 10^{-41}$
10799	3000	1000	$6{,}90 \cdot 10^{-114}$

Če pogledamo $\min_n |u(s,n,k,m)|$ za $s = 6$ in $k = 3m$ kot funkcijo m-ja, vidimo, da je $\log_{10} \min_n |u| \approx -0{,}11m - 4{,}5$.

4. Preglednice

Koristno je imeti globalno spremenljivko, v kateri si shranjujemo vsebino trenutne celice (ki jo naš podprogram NaslednjiZnak dobiva znak po znak); ko pridemo do konca celice (torej ko je naslednji znak vejica ali podpičje), pa jo izpišemo. Naša spodnja rešitev ima v ta namen tabelo vsebina, kazalec trenutniZnak pa označuje naš položaj v tej tabeli. Tudi podatek o tem, v kateri vrstici in stolpcu se trenutno nahajamo, hranimo v globalnih spremenljivkah; ko preberemo vejico, povečamo koordinato stolpca, ko pa preberemo podpičje, povečamo koordinato vrstice in postavimo stolpec spet na 1.

```
int vrstica = 1, stolpec = 1;
char vsebina[100];
char *trenutniZnak = vsebina;

void NaslednjiZnak(char c)
{
  if (c == ';') /* konec vrstice */
  {
    *trenutniZnak = 0;
    Celica(vrstica, stolpec, vsebina);
    ++vrstica; stolpec = 1;
    trenutniZnak = vsebina;
```

```
    }
    else if (c == ',') /* konec celice */
    {
      *trenutniZnak = 0;
      Celica(vrstica, stolpec, vsebina);
      ++stolpec;
      trenutniZnak = vsebina;
    }
    else
      *trenutniZnak++ = c;
}
```

5. Smučarji

Da bo manj pisanja, bomo pri tej rešitvi namesto o minimalni dopustni teži smučarja, ki jo zahteva nek par smuči, govorili kar o teži smuči.

Recimo zdaj, da gledamo otroka A in B, pri čemer je A težji od B. Če ima pri nekem veljavnem razporedu smuči otrok A lažje smuči kot otrok B, si jih lahko izmenjata, pa razpored ostane veljaven in enako dober. Podobno, če je A brez smuči, B pa jih ima, lahko B da svoje smuči A-ju, pa razpored ostane veljaven (in enako dober). Iz tega vidimo, da je dovolj, če se omejimo na takšne razporede, pri katerih dobi smuči le najtežjih nekaj otrok in pri katerih dobijo težji otroci vedno težje smuči kot lažji otroci.

Pregledujmo torej smuči padajoče po teži. Za najtežje smuči nam razmislek iz prejšnjega odstavka pove, da če jih bomo dali sploh kakšnemu otroku, jih moramo dati najtežjemu. Če je celo ta otrok zanje prelahek, vemo, da teh smuči ne moremo uporabiti; če pa je ta otrok zanje dovolj težak, ni nobenega dobrega razloga, da mu jih ne bi dali. Če mu bomo namreč dali neke lažje smuči, bodo tiste najtežje ostale neuporabljene, in tak razpored ostane veljaven (in enako dober), če temu otroku namesto lažjih smuči damo najtežje (saj smo rekli, da je on tudi za te najtežje dovolj težak), tiste lažje pa bodo mogoče prišle prav še za kakšnega lažjega otroka. Z enakim razmislekom nadaljujemo še z ostalimi pari smuči (v padajočem vrstnem redu po minimalni dopustni teži) — vsake damo najtežjemu takemu otroku, ki še ni dobil smuči, če pa je ta otrok prelahek zanje, ostanejo neuporabljene.

Naloge so sestavili: smučarji — Nino Bašić; primerjanje oklepajev — Andrej Bauer; lego kocke, kozarci, assembler — Boris Gašperin; reklama, kompresija slike, rekonstrukcija grafa — Tomaž Hočevar; kemijske formule, podajanje žoge — Jurij Kodre; globalno segrevanje — Mitja Lasič; majevska števila — Mark Martinec; okoljevarstveni ukrepi — Mojca Miklavec; (opomba) — Klemen Simonič in Mitja Trampuš; riziko — Andraž Tori; ve,jic,e, mušji drekci — Mitja Trampuš; preglednice — Klemen Žagar; vsota, telefonske številke — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2009

1. Avtobusi

Celoten postopek rešitve je pravzaprav že precej podrobno opisan v besedilu naloge, moramo ga le pozorno prebrati in rešitev zapisati v obliki podprograma.

Naloga pravi, da je izhodiščna cena vožnje od začetne postaje do neke druge postaje premosorazmerna z razdaljo med tema dvema postajama. Izračunajmo si torej najprej ceno na enoto dolžine (v spodnji rešitvi je to spremenljivka cenaNaMeter), ki jo dobimo tako, da polno ceno (od začetne do končne postaje; dobili smo jo v parametru polnaCena) delimo z razdaljo od začetne do končne postaje (to pa najdemo na koncu tabele razdalje, torej v razdalje[stPostaj − 1]). Ker sta tako cena kot razdalja vrednosti tipa **int**, je koristno enega od teh dveh operandov pretvoriti v **double**, preden ju delimo, saj bo v nasprotnem primeru naš program izvedel celoštevilsko deljenje in količnik zaokrožil na celo število, to pa bi povzročilo, da bi v nadaljevanju podprograma izračunali napačne (premajhne) cene voženj.

Zdaj torej poznamo ceno na meter in se lahko v zanki sprehodimo po postajah in izhodiščno ceno pri vsaki postaji dobimo tako, da pomnožimo ceno na meter z oddaljenostjo trenutne postaje od začetne. Nato jo moramo zaokrožiti navzgor, za kar spodnji program uporablja funkcijo ceil iz standardne knjižnice jezika C (podobne funkcije najdemo tudi pri drugih programskih jezikih). Ostane le še to, da dobljeno ceno primerjamo s ceno vožnje do prejšnje postaje (prejCena) in jo, če je treba, ustrezno podražimo, da bo vsaj za en evro dražja od cene vožnje do prejšnje postaje.

```
#include <math.h>

void IzracunajCene(int polnaCena, const int razdalje[], int cene[], int stPostaj)
{
  int prejCena, i;
  double cenaNaMeter = polnaCena / (double) razdalje[stPostaj − 1];
  for (i = 0; i < stPostaj; i++)
  {
    /* Izračunajmo izhodiščno ceno in jo zaokrožimo navzgor. */
    cene[i] = (int) ceil(razdalje[i] * cenaNaMeter);
    /* Poskrbimo še, da bo večja od cene do prejšnje postaje. */
    prejCena = (i == 0) ? 0 : cene[i − 1];
    if (cene[i] <= prejCena) cene[i] = prejCena + 1;
  }
}
```

2. Podnizi

Naloga je zelo podobna običajnemu problemu iskanja pojavitev podniza v besedilu; posebnost pri tej nalogi je le v tem, da če se podniz pojavlja večkrat znotraj iste besede, štejemo od teh pojavitev le eno. Spodnja rešitev ima zato spremenljivko zeViden, ki pove, ali smo podniz v trenutni besedi že videli ali ne. Ko najdemo pojavitev podniza, postavimo zeViden na **true**, ko pa v besedilu pridemo do presledka, postavimo zeViden na **false**. Števec pojavitev (spremenljivka kolikokrat) povečamo le, če je bil zeViden pred odkritjem te pojavitve **false**, tako da, če se pojavi podniz večkrat v isti besedi, štejemo le prvo pojavitev.

```
#include <stdbool.h>

int Kolikokrat(char *besedilo, char *podniz)
{
  int i, kolikokrat = 0;
  bool zeViden = false;
  for (; *besedilo; besedilo++)
  {
    i = 0; while (podniz[i] && podniz[i] == besedilo[i]) i++;
    if (! podniz[i]) {
      if (! zeViden) kolikokrat++;
      zeViden = true; }
    if (*besedilo == ' ') zeViden = false;
  }
  return kolikokrat;
}
```

Za odkrivanje pojavitev podniza v besedilu smo uporabili zelo preprost postopek — za vsak možni položaj podniza v besedilu primerjamo znake podniza z istoležnimi znaki besedila, dokler ne opazimo neujemanja ali pa pridemo do konca podniza. (Če nastopi konec besedila prej kot konec podniza, bomo to zaznali kot neujemanje, ker bomo v besedilu videli znak `"\0"`, v podnizu pa ne.) Našo rešitev bi lahko še izboljšali, če bi za odkrivanje pojavitev podniza v besedilu uporabili kakšen učinkovitejši postopek (Knuth-Morris-Pratt, Boyer-Moore, Rabin-Karp ipd.).

3. Latovščina

Spodnja rešitev se najprej z zanko zapelje po besedi in poišče v njej prvi samoglasnik; nanj bo kazala spremenljivka `s`. Zdaj lahko izpišemo *l* in preostanek besede od samoglasnika `s` naprej. Nato se še enkrat sprehodimo od začetka besede do `s` in izpišemo še ta začetni del besede (vključno s samoglasnikom `s`), na koncu pa izpišemo še `te`. Paziti moramo še na primer, ko samoglasnika v besedi sploh ne najdemo; takrat jo moramo, kot pravi naloga, izpisati nespremenjeno.

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

void Latovscina(char *beseda)
{
  /* Poiščimo prvi samoglasnik v besedi. */
  char *s = beseda;
  while (*s && ! strchr("aeiou", *s)) s++;
  /* Če samoglasnika ni, izpišimo besedo nespremenjeno. */
  if (! *s) { printf("%s", beseda); return; }
  /* Izpišimo „l“ in del besede od prvega samoglasnika naprej. */
  printf("l%s", s);
  /* Izpišimo začetek besede do vključno prvega samoglasnika. */
  do { fputc(*beseda, stdout); } while (beseda++ != s);
  printf("te"); /* Dodajmo še „te“ na koncu. */
}
```

4. Disk

Preprosta, vendar neučinkovita rešitev je, da pred vsakim premikom glave pregledamo celoten seznam še neprebranih cilindrov in poiščemo med njimi tistega, ki je

najbližji trenutnemu položaju glave. To bo naslednji cilinder, ki ga bomo prebrali; glava se bo torej premaknila tja, njegovo številko pa bomo pobrisali iz seznama še neprebranih cilindrov. Zapišimo ta postopek še v C-ju (seznam cilindrov, ki jih je treba prebrati, je v tabeli a, trenutni položaj glave pa v spremenljivki g).

```
void Branje(int a[], int n, int g)
{
  int i, naj;
  while (n > 0) {
    /* Trenutni položaj glave je g.
       Poglejmo, kateri od še neprebranih cilindrov je najbližji g. */
    for (naj = 0, i = 1; i < n; i++)
      if (abs(g − a[i]) < abs(g − a[naj])) naj = i;
    /* Najbližji cilinder je a[naj]; glava se zdaj premakne tja in ga prebere. */
    g = a[naj]; printf("%d\n", a[naj]);
    /* Pobrišimo a[naj] iz seznama cilindrov, ki jih je treba še prebrati. */
    a[naj] = a[−−n]; }
}
```

Ta rešitev je precej neučinkovita; ko imamo v seznamu še k neprebranih cilindrov, izvede notranja zanka $k-1$ iteracij, da poišče tistega, ki je najbližji trenutnemu položaju glave. Ker se število neprebranih cilindrov počasi zmanjšuje od n proti 0, se izvede vsega skupaj $(n-1)+(n-2)+\ldots+2+1 = n(n-1)/2$ iteracij notranje zanke, tako da je časovna zahtevnost našega postopka $O(n^2)$.

Naj bodo $a_1, \ldots, a_n$ številke cilindrov, ki jih moramo prebrati, urejene v naraščajočem vrstnem redu. Do učinkovitejše rešitve pridemo z naslednjim opažanjem: v vsakem trenutku tvorijo že prebrani cilindri neko strnjeno podzaporedje prvotnega zaporedja $a_1, \ldots, a_n$. Z drugimi besedami, v vsakem trenutku obstajata neka i in j, tako da so že prebrani cilindri $a_i, a_{i+1}, \ldots, a_{j-1}, a_j$, vsi ostali pa so še neprebrani. O tem se lahko prepričamo z indukcijo; ko preberemo prvi cilinder, recimo a_k, trditev očitno drži (če vzamemo $i = j = k$). Recimo zdaj, da so že prebrani cilindri od a_i do a_j. Glava se torej nahaja na tistem od njih, ki je bil nazadnje prebran, torej nekje na intervalu $[a_i, \ldots, a_j]$. Če bo naslednji prebrani cilinder a_{i-1} ali a_{j+1}, bo naša trditev še vedno držala. Če bo naslednji prebrani cilinder eden od $a_1, \ldots, a_{i-2}$, pa vrstni red branja ne ustreza zahtevam naloge, saj je trenutnemu položaju glave (ki je $g \geq a_i$) gotovo bližji cilinder a_{i-1} kot pa eden od $a_1, \ldots, a_{i-2}$. Iz podobnih razlogov tudi vidimo, da naslednji prebrani cilinder ne more biti eden od $a_{j+2}, \ldots, a_n$, ker takšen vrstni red spet ne bi ustrezal zahtevam naloge.

Učinkovitejša rešitev lahko torej vzdržuje indeksa i in j ter na vsakem koraku pogleda, kateri od cilindrov a_{i-1} in a_{j+1} je bližje trenutnemu položaju glave. Tako imamo le še konstantno mnogo dela, da si izberemo naslednji register, ki ga bomo prebrali; časovna zahtevnost celotnega postopka se s tem zmanjša na $O(n)$ namesto $O(n^2)$.

```
void Branje2(int a[], int n, int g)
{
  int i, j;
  /* Poglejmo, kateri cilinder je najbližji začetnemu položaju glave... */
  for (i = 0, j = 1; j < n; j++)
    if (abs(a[j] − g) < abs(a[i] − g)) i = j;
  /* ... in ga preberimo. */
  j = i; g = a[i]; printf("%d\n", g);
```

```
  /* V zanki preberimo še vse ostale cilindre. */
  while (i > 0 || j < n − 1)
  {
    /* Trenutni položaj glave je g. Poglejmo, ali je zdaj bližji
       cilinder a[i − 1] ali a[j + 1]. Pri tem pride a[i − 1] v poštev
       le, če je i > 0; cilinder a[j + 1] pa pride v poštev le, če je j < n − 1. */
    if (i > 0 && (j == n − 1 || abs(a[i − 1] − g) < abs(a[j + 1]))) g = a[−−i];
    else g = a[++j];
    printf("%d\n", g);
  }
}
```

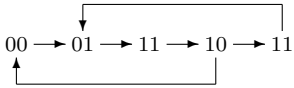
5. Železnica

Nalogo lahko razdelimo na dva precej ločena problema: (1) ko se čez nek par senzorjev pripelje vagon, moramo biti sposobni to zaznati in tudi ugotoviti, v katero smer se pelje; (2) na podlagi teh podatkov pa moramo znati pravilno upravljati z zapornico in sireno.

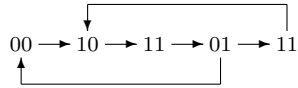
Podproblem (2) je pravzaprav preprost: v globalnih spremenljivkah štejmo, koliko je trenutno na progi vagonov, ki se peljejo v levo (recimo L), in koliko takih, ki se peljejo v desno (recimo D). Ko se vagon, ki se premika v levo, zapelje mimo desnega para senzorjev, moramo števec L povečati za 1, ko pa se zapelje mimo levega para senzorjev, moramo L zmanjšati za 1. Podobno lahko razmišljamo tudi pri vagonih, ki se peljejo v desno. Pravila za upravljanje z zapornico in sireno so zdaj takšna: če sta L in D kdaj oba hkrati večja od 0, moramo sprožiti sireno; ko se $L + D$ poveča z 0 na 1, moramo dvigniti zapornico; ko pa se $L + D$ zmanjša z 1 na 0, jo moramo spustiti.

Lotimo se zdaj še podproblema (1). Stanje para senzorjev lahko predstavimo z dvema števkama, pri čemer 0 pomeni, da žarek senzorja ni prekinjen, 1 pa, da je. Ko se do para senzorjev z leve pripelje prvi vagon, se stanje senzorjev spremeni iz 00 v 01 (levi senzor še ne zaznava vagona, desni pa ga že) in od tam v 11 (ko se vagon premakne še malo dlje proti levi) in nato 10 (ko se vagon zapelje že popolnoma mimo desnega senzorja, levi pa ga še zaznava). Če za tem vagonom ne prihaja naslednji, bomo iz 10 kmalu prišli v 00 in smo na istem kot pred prihodom tega vagona. Bolj zanimivo vprašanje je, kaj se zgodi, če za prvim vagonom takoj pride naslednji. Naloga pravi, da je razdalja med vagonoma 1 m, enaka kot razdalja med senzorjema. Zato ne vemo zagotovo, ali bomo iz stanja 10 (ko levi senzor še zaznava prvi vagon, desni pa trenutno ne zaznava ničesar) prišli za hip v stanje 11 (ko levi senzor še vedno zaznava prvi vagon, desni pa je že zaznal drugi vagon)[10] ali 00 (ko levi senzor ne zaznava več prvega vagona, desni pa še ne zaznava drugega); v vsakem primeru pa bomo iz enega od teh dveh stanj nato kmalu prišli v 01 (ko levi senzor ne zaznava ničesar, ker je v presledku med vagonoma, desni senzor pa zaznava drugi vagon), iz tega pa v 11 in tako naprej enako kot pri prvem vagonu. Podoben razmislek lahko opravimo tudi za vagone, ki se premikajo v desno namesto v levo. Možni prehodi v stanju števcev so torej naslednji:

[10] Pri nalogi, ki smo jo uporabili na tekmovanju (2009.1.5), te možnosti ni bilo, ker je bila razdalja med vagonoma večja kot razdalja med senzorjema.



vagoni, ki se peljejo v levo



vagoni, ki se peljejo v desno

Iz teh slik lahko vidimo, da smeri gibanja posameznega vagona ne moremo vedno zlahka prepoznati iz sprememb stanja senzorjev pri tem vagonu. Ko se mimo senzorjev pelje vagon v levo, gre lahko stanje skozi stanja $00 \to 01 \to 11 \to 10$ ali pa skozi stanja $11 \to 01 \to 11 \to 10$ (slednje je sicer mogoče le, če to ni prvi vagon v vlaku). Edina prehoda, ki sta skupna vsem vagonom v levo, sta torej $01 \to 11 \to 10$; kot pa vidimo iz desne slike, lahko oba tadva prehoda nastopita tudi pri vagonu, ki se pelje v desno.

Smer gibanja vagona lahko zanesljivo prepoznamo le, če sta bila senzorja pred prihodom vagona v stanju 00: prehod $00 \to 01$ nam pove, da prihaja vlak, ki se pelje v levo, prehod $00 \to 10$ pa, da prihaja vlak, ki se pelje v desno. To smer si je koristno zapomniti v neki spremenljivki. Odtlej vsakič, ko prideta senzorja v stanje 01 (če imamo vlak, ki se pelje v levo) oz. 10 (če imamo vlak, ki se pelje v desno), vemo, da se začenja nov vagon. Ko prideta senzorja v stanje 00, pa vemo, da je vlaka konec (oz. tudi če ga ni, je razmik med vagonoma tolikšen, da bomo pri naslednjem vagonu lahko spet prepoznali njegovo smer).

```
bool stanje[4] = { false, false, false, false };
enum { Levo = −1, Desno = 1 };
int L = 0, D = 0, smer[2] = { 0, 0 };

void SenzorSprememba(int stevilkaSenzorja, bool prekinjen)
{
  int LL = L, DD = D, s = stevilkaSenzorja − 1;
  /* Poglejmo, kateremu paru pripada senzor s (levemu ali desnemu)
     in kateri senzor je v paru z njim (recimo mu t). */
  int par = s / 2, t = s ^ 1;
  /* Pri prehodu 00 → 01 vemo, da se začenja vlak, ki pelje v levo;
     pri prehodu 00 → 10 vemo, da se začenja vlak, ki pelje v desno;
     pri prehodu v 00 pa vemo, da je vlaka konec. */
  if (! stanje[t])
    if (prekinjen) smer[par] = (s & 1) ? Levo : Desno; else smer[par] = 0;
  stanje[s] = prekinjen;
  /* Če je smer == Levo in je novo stanje 01
     ali pa je smer == Desno in je novo stanje 10,
     pomeni, da smo zaznali nov vagon, ki pelje v to smer. */
  if (smer[par] == Levo && ! stanje[2 * par] && stanje[2 * par + 1])
    LL += (par == 1) ? 1 : −1;
  else if (smer[par] == Desno && ! stanje[2 * par] && stanje[2 * par + 1])
    DD += (par == 0) ? 1 : −1;
  /* Poglejmo zdaj, če je treba kaj narediti z zapornico ali sireno.
     Staro število vagonov je v L in D, novo pa v LL in DD. */
  if (L + D == 0 && LL + DD > 0) Zapornica(Gor);
  else if (L + D > 0 && LL + DD == 0) Zapornica(Dol);
  if (LL > 0 && DD > 0) Sirena();
  L = LL; D = DD;
}
```

6. Peščene piramide

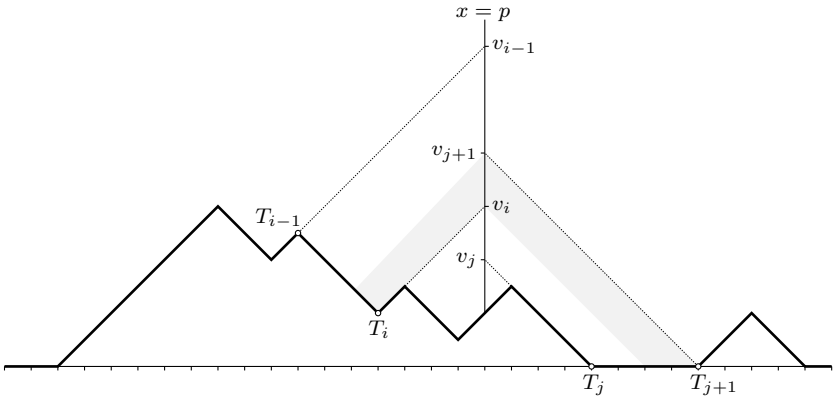
Recimo, da že imamo neko zaporedje točk (x_i, y_i), $i = 1, \ldots, n$ (urejeno po naraščajočih x-koordinatah), ki opisuje stanje naših piramid po prvih nekaj nasipanjih peska.[11] Predpostavimo lahko, da je vsaka daljica, ki jo opisujeta dve zaporedni točki, bodisi vodoravna (in to z y-koordinato 0) bodisi pod kotom 45° (saj začnemo z vodoravno podlago in odtlej pri nasipanju peska nastajajo vedno piramide s stranicami pod kotom 45°). Recimo, da v naslednjem koraku nasipamo pesek pri koordinati $x = p$. Če v našem zaporedju točk še ni take z x-koordinato točno p, jo dodajmo (y-koordinato določimo z interpolacijo med najbližjima točkama pred in za $x = p$).

Za vsako točko našega zaporedja, recimo $T_k = (x_k, y_k)$, se lahko vprašamo, kako visoka bi morala biti piramida z vrhom pri $x = p$, da bi dosegla oz. pogoltnila točko T_k; vidimo lahko, da bi bil vrh piramide takrat na y-koordinati $v_k := y_k + |x_k - p|$. Če se v našem zaporedju točk postavimo na tisto z $x = p$ in se od tam premikamo v levo, bodo vrednosti v_k ves čas naraščale (ali pa ostajale enake); to je posledica dejstva, da je daljica med dvema zaporednima točkama v našem zaporedju vedno bodisi vodoravna bodisi pod kotom 45°. Enako velja tudi, če začnemo pri $x = p$ in se premikamo v našem zaporedju v desno — tudi takrat vrednosti v_k ves čas naraščajo (ali ostajajo enake). Če je torej na primer $x_i \le p$, vemo, da ko bi višina naše nove piramide rasla od v_i proti v_{i-1}, bi levi rob nastajajoče piramide počasi zasipal daljico od T_i proti T_{i-1}. Podobno, če je $x_j \ge p$, bi v času, ko bi višina naše nove piramide rasla od v_j proti v_{j+1}, desni rob te piramide počasi zasipal daljico od T_j proti T_{j+1}.

Zdaj se torej lahko hkrati premikamo po našem zaporedju točk z dvema kazalcema: oba začneta pri $x = p$, indeks i se premika v levo, indeks j pa v desno. V vsakem koraku torej gledamo levo daljico T_iT_{i-1}, ki jo zasipamo pri višinah z intervala $[v_i, v_{i-1}]$, in desno daljico T_jT_{j+1}, ki jo zasipamo pri višinah z intervala $[v_j, v_{j+1}]$. (Dobljena intervala sta v bistvu projekciji daljic T_iT_{i-1} in T_jT_{j-1} na premico $x = p$; glej sliko na str. 83.) Izračunajmo presek intervalov $[v_i, v_{i-1}]$ in $[v_j, v_{j+1}]$; to je torej interval $[v', v'']$ za $v' = \max\{v_i, v_j\}$ in $v'' = \min\{v_{i-1}, v_{j+1}\}$. Območje, ki ga piramida na novo pokrije, ko se njen vrh pomika od v' proti v'', je sestavljeno iz dveh trapezov (na sliki na str. 83 je pobarvano s svetlo sivo barvo).

Na levi strani leži sivi trapez znotraj malo večjega trapeza, ki ga omejujejo oglišča T_i, T_{i-1}, (p, v_i) in (p, v_{i-1}) (zadnji dve sta projekciji prvih dveh na premico $x = p$). Definirajmo $u_k := |x_k - p|\sqrt{2}$ za vsak k od 1 do n. Pri tem, ko v narašča od v_i do v_{i-1}, narašča premosorazmerno z v tudi širina našega trapeza, in sicer od u_i do u_{i-1}. Ni torej težko izračunati širine pri nekem konkretnem v: to je $w_L(v) := u_i + \frac{v - v_i}{v_{i-1} - v_i}(u_{i-1} - u_i)$, kar je naprej enako $c_L + d_L v$, če pišemo $d_L := (u_{i-1} - u_i)/(v_{i-1} - v_i)$ in $c_L := u_i - d_L v_i$. Kot smo videli zgoraj, nas ta trapez ne bo zanimal v celoti, ampak le za v-je na intervalu od v' do v'' (to je sivi del trapeza na sliki). Če ga torej gledamo od v' do nekega konkretnega v z intervala $[v', v'']$, imamo malo manjši trapez z osnovnicama $w_L(v')$ in $w_L(v)$ ter višino $(v - v')/\sqrt{2}$.

[11] Na začetku lahko vzamemo zaporedje dveh točk, $(x_L, 0)$, $(x_D, 0)$, ki opisuje ravno podlago pred prvim nasipanjem. Krajišči x_L in x_D je pametno postaviti dovolj daleč narazen, da nam pesek nikoli ne bo padel čez rob te daljice; če je na primer x'_L najbolj leva koordinata, pri kateri kdaj nasipamo pesek, x'_D pa najbolj desna, lahko zdaj vzamemo $x_L = x'_L - \sqrt{A}$ in $x_D = x'_D + \sqrt{A}$, če je A skupna količina vsega peska, ki ga bomo nasuli.



Njegova ploščina je torej $A_L(v) := (w_L(v) + w_L(v'))/2 \cdot (v - v')/\sqrt{2}$, kar je naprej enako $(c_L + d_L(v + v')/2) \cdot (v - v')/\sqrt{2}$.

Analogen razmislek lahko izvedemo tudi na desni strani, kjer imamo trapez z oglišči T_j, T_{j+1}, (p, v_j) in (p, v_{j+1}); podobno kot prej c_L, d_L in $A_L(v)$ zdaj dobimo c_D, d_D in $A_D(v)$. Ploščina obeh trapezov skupaj je zdaj $A(v) = A_L(v) + A_D(v) = (c + d(v + v')/2) \cdot (v - v')/\sqrt{2}$ za $c = c_L + c_D$, $d = d_L + d_D$. Ta ploščina je kvadratna funkcija v-ja in jo lahko zapišemo kot $A(v) = a_2 v^2 + a_1 v + a_0$ za $a_2 = d/2\sqrt{2}$, $a_1 = c/\sqrt{2}$ in $a_0 = -(cv' + dv'^2/2)/\sqrt{2}$.

Tako torej vidimo, da če smo imeli doslej piramido z vrhom (p, v') in nasujemo pri $x = p$ še vsaj $A(v'')$ dodatnega peska, bo vrh piramide narastel do višine v''. Takrat je vsaj ena od obeh opazovanih daljic (torej T_iT_{i-1} in T_jT_{j+1}) v celoti zasuta in se moramo po zaporedju pomakniti še dlje stran od $x = p$; torej, če je $v'' = v_{i-1}$, zmanjšamo i za 1; in če je $v'' = v_{j+1}$, povečamo j za 1. Zdaj imamo pred sabo nov par daljic, celotni opisani postopek ponovimo še na njem in tako naprej.

Postopek se ustavi, ko ugotovimo, da je $A(v'')$ večji od količine peska, ki nam je še preostala. Če imamo še K enot peska in je $K < A(v'')$, moramo zdaj poiskati višino, ki jo piramida doseže, ko peska zmanjka; poiščimo torej tisti nenegativni v, ki reši kvadratno enačbo $A(v) = K$ oz. $a_2 v^2 + a_1 v + (a_0 - K) = 0$. Zdaj vemo, da je vrh nove piramide v točki (p, v). Katera točka na T_iT_{i-1} se projicira v (p, v)? To je $T_i' := T_i + \lambda(T_{i-1} - T_i)$ za $\lambda = (v - v_i)/(v_{i-1} - v_i)$. Podobno tudi na desni ugotovimo, katera točka na T_jT_{j+1} se projicira v (p, v); recimo ji T_j'. Iz našega zaporedja točk lahko zdaj pobrišemo vse točke od vključno T_i do vključno T_j, saj jih je zasula naša nova piramida; namesto njih pa vstavimo med T_{i-1} in T_{j+1} zdaj tri nove točke: T_i', (p, v) in T_j'.

7. Enkratno prirejanje

Zanke po i, kakršno ima prvotni podprogram Kolikokrat, si ne moremo privoščiti, saj bi se v njej spremenljivki i priredila nova vrednost v vsaki iteraciji, tako da bi prekršili omejitev, da smemo vsaki spremenljivki prirediti vrednost največ enkrat. Pomagamo si lahko tako, da namesto zanke uporabimo rekurzijo. Napisati moramo rekurzivni podprogram, ki naredi tisto, kar je v prvotni različici naredila ena iteracija zanke, nato pa rekurzivno pokliče sam sebe, s čimer poskrbi za izvajanje preostanka

zanke. Vrednosti, ki so jih spremenljivke imele ob začetku iteracije, dobi rekurzivni podprogram kot parametre, njihove vrednosti ob koncu iteracije pa izračuna v novih spremenljivkah (mm in kk) in jih pošlje kot parametre rekurzivnemu klicu, ki bo izvedel preostanek zanke.

```
int KolikokratRekurzivno(int a[], int n, int i, int m, int k)
{
  int mm, kk;
  if (i == n) return k; /* konec zanke oz. rekurzije */
  if (i == 0 || a[i] < m) mm = a[i], kk = 1;
  /* Ker si ne moremo privoščiti izraza k = k + 1, moramo ločiti
     primer, ko je treba k povečati za 1 (ker je a[i] == m),
     in primer, ko ostane k nespremenjen (ker je a[i] > m). */
  else if (a[i] == m) mm = m, kk = k + 1;
  else mm = m, kk = k;
  return KolikokratRekurzivno(a, n, i + 1, mm, kk);
}

int Kolikokrat(int a[], int n) { return KolikokratRekurzivno(a, n, 0, 0, 0); }
```

Kako pa bi z enkratnim prirejanjem izvedli urejanje tabele števil? Začnimo na primer s postopkom urejanja z izbiranjem (*selection sort*): poiščimo najmanjše število v tabeli a; v urejeni različici tabele bo to število vsekakor stalo na začetku, torej ga vpišimo v b[0]; nato poiščimo najmanjše izmed preostalih števil v tabeli a, ga vpišimo v b[1] in tako naprej. Toda kako vemo, katera števila v a so „preostala“, torej taka, ki jih še nismo zapisali v tabelo b? Elementov ne moremo premikati po tabeli a, da bi imeli neuporabljene elemente vedno v začetnem delu tabele (kot je to sicer običajno pri urejanju z izbiranjem), saj bi tako prekršili omejitev o enkratnem prirejanju. Ker elemente tabele a prepisujemo v b v naraščajočem vrstnem redu, so načeloma v vsakem trenutku „preostali“ elementi tabele a natanko tisti, ki so večji od števila, ki smo ga doslej nazadnje vpisali v b. V b[0] vpišemo najmanjši element tabele a; nato v b[1] vpišemo najmanjši tak element tabele a, ki je večji od b[0]; nato v b[2] vpišemo najmanjši tak element tabele a, ki je večji od b[1]; in tako naprej.

Posebej pa moramo paziti še na možnost, da ima več elementov v tabeli a enako vrednost. Ko poiščemo naslednjo najmanjšo vrednost v tabeli a, lahko spotoma še preštejemo, kolikokrat se pojavlja (ta problem je pravzaprav skoraj čisto enak tistemu, ki smo ga reševali zgoraj s podprogramom Kolikokrat), in nato v tabelo b vpišemo toliko kopij te vrednosti.

```
/* Vrne najmanjši element izmed a[0], ..., a[n − 1]. */
int Najmanjsi(const int a[], int n)
{
  int m;
  if (n == 1) return a[0];
  m = Najmanjsi(a, n − 1);
  return m < a[n − 1] ? m : a[n − 1];
}

/* Poišče v a[0..n − 1] najmanjši tak element, ki je večji od vecjiOd.
   Če ni nobenega takega, vrne vecjiOd. */
int Naslednji(const int a[], int n, int vecjiOd)
{
  int m;
```

```
  if (n == 0) return vecjiOd;
  m = Naslednji(a, n − 1, vecjiOd);
  return vecjiOd < a[n − 1] && (m <= vecjiOd || a[n − 1] < m) ? a[n − 1] : m;
}

/* Skopira vse pojavitve vrednosti x iz območja a[0..n − 1] v tabelo b od celice b[k]
   naprej. Vrne indeks prve naslednje še neuporabljene celice iz b po tem kopiranju. */
int Skopiraj(const int a[], int n, int b[], int k, int x)
{
  int kk;
  if (n == 0) return k;
  if (a[n − 1] == x) b[k] = x, kk = k + 1;
  else kk = k;
  return Skopiraj(a, n − 1, b, kk, x);
}

/* Skopira iz tabele a v tabelo b od celice b[k] naprej vse tiste vrednosti, ki so večje od
   b[k − 1] (oz. vse vrednosti sploh, če je k = 0), in to v naraščajočem vrstnem redu. */
void UrediRekurzivno(const int a[], int n, int b[], int k)
{
  int x, kk;
  if (k >= n) return;
  if (k == 0) x = Najmanjsi(a, n);
  else x = Naslednji(a, n, b[k − 1]);
  kk = Skopiraj(a, n, b, k, x);
  UrediRekurzivno(a, n, b, kk);
}

void Uredi(const int a[], int n, int b[]) { UrediRekurzivno(a, n, b, 0); }
```

8. Poravnavanje desnega roba

Vhodno besedilo pregledujmo besedo po besedo, dokler se nam ne nabere toliko besed, da naslednja že ne bi šla več v isto vrstico kot vse dosedanje (ker bi bile, skupaj s presledki med besedami, dolge več kot n znakov). Pri tem tudi štejmo, koliko besed se nam je nabralo (spodnji podprogram ima v ta namen spremenljivko stBesed). Zdaj lahko izračunamo, koliko dodatnih presledkov bomo morali vriniti, da bo nastala vrstica dolžine natanko n (spremenljivka dodatni). Nato se še enkrat sprehodimo čez te besede in jih izpišimo, pri tem pa med njimi vrinimo še dodatne presledke. Naloga pravi, da jih moramo razporediti čim bolj enakomerno. Recimo, da bo v trenutni vrstici s besed in da bo treba vanjo vriniti skupno d dodatnih presledkov; potem imamo $s-1$ obstoječih presledkov med besedami in mednje je treba enakomerno razporediti še tistih d dodatnih, torej je smiselno reči, naj pri vsakem od obstoječih presledkov pride še $d/(s-1)$ novih. Z drugimi besedami, skupno število dodatnih presledkov, vrinjenih ob prvih k obstoječih presledkov, naj bo približno $k \cdot d/(s-1)$. Spodnja rešitev hrani v spremenljivki izpisani število že vrinjenih presledkov in jih na koncu vsake besede dodaja tako dolgo, da to število doseže $k \cdot d/(s-1)$. Vprašanje je še, kako naj $k \cdot d/(s-1)$ zaokrožimo (saj to število v splošnem ni celo); če bi vedno zaokrožali navzdol (oz. navzgor), bi bili dodatni presledki v povprečju premaknjeni bolj proti desnemu (oz. levemu) robu vsake vrstice; zato raje zaokrožimo na najbližje celo število (izračunamo torej $\lfloor k \cdot d/(s-1) + 1/2 \rfloor$).

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

```
void Izpisi(char *besedilo, int n)
{
  char *vrstica, *p, *konec;
  int stBesed, dodatni, izpisani, k;
  for (vrstica = besedilo; *vrstica; )
  {
    /* Poglejmo, koliko naslednjih besed lahko izpišemo v trenutno vrstico. */
    for (p = vrstica, konec = p, stBesed = 0; *p; ) {
      while (*p == ' ') p++; /* Preskočimo morebitne presledke. */
      while (*p && *p != ' ') p++; /* Preskočimo besedo. */
      if (p - vrstica > n) break; /* Ali gre nova beseda še lahko v to vrstico? */
      konec = p; stBesed++; }
    /* Koliko presledkov moramo dodati v to vrstico? */
    dodatni = n - (konec - vrstica);
    if (! *konec) dodatni = 0; /* Zadnji vrstici ne poravnavamo desnega roba. */
    /* Če je v vrstici ena sama beseda, presledkov ne bomo mogli dodajati;
       postavimo stBesed na 2, da spodaj ne bo prišlo do deljenja z 0. */
    if (stBesed < 2) stBesed = 2;
    /* Izpišimo novo vrstico. */
    izpisani = 0; k = 0;
    while (vrstica < konec) {
      /* Na meji med presledkom in besedo bomo vrivali morebitne dodatne
         presledke. Spremenljivka k šteje, koliko od teh mej smo doslej že
         videli (vsega skupaj jih je stBesed - 1). */
      if (vrstica[0] == ' ' && vrstica[1] != ' ') k++;
      /* Vrinimo dodatne presledke, če jih je treba. */
      while (izpisani < (dodatni * k + (stBesed - 1) / 2) / (stBesed - 1))
        fputc(' ', stdout), izpisani++;
      fputc(*vrstica++, stdout); }
    fputc('\n', stdout);
    /* Preskočimo presledke, da se naslednja vrstica izpisa ne bo začela s presledkom. */
    while (*vrstica == ' ') vrstica++;
  }
}
```

9. Lačni mravljinčar

Nalogo lahko elegantno rešimo tako, da se rekurzivno sprehajamo po drevesu rovov in sproti vzdržujemo podatek o tem, koliko centimetrov jezika mravljinčarju še ostane, če je že prišel do trenutne točke. Če se iz trenutne točke rovi nadaljujejo v eno ali več globljih točk, izvedemo za vsako od njih po en rekurzivni klic. Posebej moramo paziti še na primere, ko kakšne od teh naslednjih točk ne moremo doseči, ker bi presegli dolžino mravljinčarjevega jezika; v tem primeru rekurzivnega klica ne smemo izvesti, moramo pa pogledati, koliko mravelj lahko v tem rovu vendarle še dosežemo. V vsakem primeru na koncu pogledamo, katero nadaljevanje poti nam prinese največ mravelj in to vrnemo kot rezultat podprograma.

podprogram PregljPoddrevo(u, d):
(* *Vhodna podatka: u je neka točka v drevesu; d je dolžina jezika, ki še ostane, če mravljinčar z njim že doseže točko u. Podprogram vrne največje število mravelj, ki ga lahko mravljinčar doseže pri tej dolžini jezika od točke u naprej.* *)
$m := 0$;
za vsakega u-jevega otroka v v drevesu:

```
    if d_uv > d then c := g_uv d
    else c := g_uv d_uv + PregljePoddrevo(v, d − d_uv);
    if c > m then m := c;
  return m;
```

Rezultat, po katerem sprašuje naloga, lahko zdaj dobimo tako, da rekurzijo poženemo v korenu drevesa ($u = 1$) z jezikom dolžine l, torej kličemo PregljePoddrevo$(1, l)$.

10. Prepogibanje traku

Naj bo $g[i, d]$ logična vrednost (**true** ali **false**), ki pove, ali je mogoče z rezanjem po pravilih iz besedila naloge predelati začetni niz $s_1 \dots s_n$ v niz $s_i \dots s_{i+d-1}$. To lahko učinkovito računamo po padajoči dolžini: ko ugotovimo, da je mogoče dobiti niz $s_i \dots s_{i+d-1}$, lahko zanj pogledamo, kakšne podnize mu je mogoče odrezati na začetku ali na koncu, in tako odkrijemo vse krajše nize, ki jih lahko dobimo iz $s_i \dots s_{i+d-1}$ z enim dodatnim rezom. Tako se počasi premikamo od daljših nizov proti krajšim in sčasoma odkrijemo vse nize, ki jih lahko dosežemo z rezanjem po pravilih naloge.

```
for d := 1 to n do for i := 1 to n + 1 − d do g[i, d] := false;
g[1, n] := true;
for d := n downto 1 do for i := 1 to n + 1 − d:
  if not g[i, d] then continue;
  for k := 1 to ⌈d/2⌉ − 1:
    (* Ali lahko odrežemo prvih k znakov niza s_i ... s_{i+d−1}? *)
    if s_i s_{i+1} ... s_{i+k−1} = s_{i+2k} s_{i+2k−1} ... s_{i+k+1} then g[i + k, d − k] := true;
    (* Ali lahko odrežemo zadnjih k znakov niza s_i ... s_{i+d−1}? *)
    if s_{i+d−2k−1} s_{i+d−2k} ... s_{i+d−k−2} = s_{i+d−1} s_{i+d−2} ... s_{i+d−k} then
                                                          g[i, d − k] := true;
```

V notranji zanki imamo dva stavka **if**, ki preverita, ali lahko v skladu s pravili naloge nizu $s_i s_{i+1} \dots s_{i+d-1}$ odrežemo prvih oz. zadnjih k znakov. Tadva pogoja pravzaprav preverita, ali je $s_i \dots s_{i+2k}$ palindrom (tedaj smemo odrezati prvih k znakov) in ali je $s_{i+d-2k-1} \dots s_{i+d-1}$ palindrom (tedaj smemo odrezati zadnjih k znakov). Zato je koristno, če si podatke o tem, kateri podnizi prvotnega niza s so palindromi, naračunamo vnaprej in jih hranimo v neki tabeli. Naj bo torej $p[i, d]$ logična vrednost, ki pove, ali je $s_i \dots s_{i+d-1}$ palindrom (potrebujemo jo le za lihe d):

```
for i := 1 to n do p[i, d] := true;
for d := 3 to n do if d mod 2 = 1:
  for i := 1 to n + 1 − d do
    (* Podniz s_i ... s_{i+d−1} je palindrom natanko tedaj, ko sta prvi in
       zadnji znak enaka in je podniz med njima tudi sam palindrom. *)
    p[i, d] := p[i + 1, d − 2] and s_i = s_{i+d−1};
```

Pogoja iz notranje zanke algoritma za računanje tabele g se tako poenostavita v

```
    if p[i, 2k + 1] then g[i + k, d − k] := true;
    if p[i + d − 2k − 1, 2k + 1] then g[i, d − k] := true;
```

Zdaj torej vidimo, da nam izračun tabele p vzame $O(n^2)$ časa, izračun tabele g pa $O(n^3)$ časa. Na koncu moramo le še pogledati, kateri je najmanjši tak d, pri katerem je $g[i,d] = \textbf{true}$ za vsaj en indeks i; to je rezultat, po katerem sprašuje naloga.

V praksi bi bila lahko koristna še naslednja izboljšava: ko imamo pripravljeno tabelo p, si lahko za vsak indeks i pripravimo seznam dolžin d, za katere je $s_i \ldots s_{i+d-1}$ palindrom (recimo temu seznamu $P[i]$), in seznam dolžin d, za katere je $s_{i-d+1} \ldots s_i$ palindrom (recimo temu seznamu $P'[i]$). (V obeh seznamih naj bodo le lihi d-ji.) Prej smo imeli v glavnem delu algoritma zanko, ki je šla po vseh možnih k in pri vsakem preverila, ali bi lahko odrezali začetnih/končnih k znakov; s pomočjo novih seznamov P in P' pa si lahko privoščimo zanki, ki bosta šli le po tistih k, pri katerih je rezanje možno:

for each $k \in P[i]$ **do if** $2k+1 \le d$ **then** $g[i + \frac{k-1}{2}, d - \frac{k-1}{2}] := \textbf{true}$;
for each $k \in P'[i+d-1]$ **do if** $2k+1 \le d$ **then** $g[i, d - \frac{k-1}{2}] := \textbf{true}$;

V najslabšem primeru nismo s tem sicer ničesar pridobili, saj sta seznama lahko dolga $O(n)$ elementov; če pa je palindromnih podnizov malo, lahko s tem prihranimo precej časa. (Vsi seznami $P[i]$ in $P'[i]$ naj bodo tudi urejeni naraščajoče, tako da se lahko zanki po k takoj prekineta, ko postane $2k+1$ večji od d.)

11. Tajna pošta

Postopek rešitve je pravzaprav precej podrobno opisan že v besedilu naloge. Sprehodimo se po sprejemnem zapisniku in si za vsakega pošiljatelja x pripravimo seznam datumov, na katere je kaj pošiljal; recimo temu seznamu $r(x)$ (isti datum se lahko v seznamu pojavi tudi večkrat, če je agent na tisti dan poslal več sporočil). Podobno naredimo še v oddajnim zapisnikom, iz katerega za vsako šifro agenta z dobimo seznam $r'(z)$. Zdaj moramo te sezname primerjati; če je na primer seznam $r(x)$, dobljen iz sprejemnega zapisnika za pošiljatelja x, enak seznamu $r'(z)$, ki smo ga dobili iz oddajnega zapisnika za pošiljatelja z, potem je mogoče, da šifra z v oddajnem zapisniku predstavlja istega agenta kot številka x v sprejemnem zapisniku: $h(x) = z$. (Vrstni red datumov v seznamu pri teh primerjavah ni pomemben, zato moramo datume v vsakem seznamu najprej urediti, če nista bila že vhodna zapisnika urejena po datumih.) Če se za nek x zgodi, da ni nobenega z-ja, ki bi ustrezal pogoju $r(x) = r'(z)$, sledi, da sta zapisnika nekonsistentna in nista mogla nastati na način, ki ga opisuje besedilo naloge. Če pa se za nek x zgodi, da obstaja več z-jev, za katere je $r(x) = r'(z)$, potem vidimo, da po postopku, ki ga predpisuje naloga, ne moremo zanesljivo rekonstruirati funkcije h, saj ne vemo, kateri od teh z-jev je pravi.

Vprašanje je še, kako lahko učinkovito preverimo, kateri seznam, dobljen iz oddajnega zapisnika, se ujema z nekim seznamom, dobljenim iz vhodnega zapisnika. Ena možnost je, da z dvema gnezdenima zankama po x in z pregledamo vse pare agentov in pri vsakem preverimo, če se $r(x)$ in $r'(z)$ ujemata.

Bolj učinkovita možnost je, da pripravimo tabelo parov $(r(x), x)$ za $x = 1, \ldots, n$ in jo uredimo; podobno pripravimo tudi tabelo parov $(r'(z), z)$ za $z = 1, \ldots, n$ in jo uredimo. Zdaj bi moralo za vsak i veljati, da sta $r(x)$ pri i-tem elementu prve tabele in $r'(z)$ pri i-tem elementu druge tabele enaka (če nista, vemo, da sta zapisnika nekonsistentna) in da je pripadajoči z možna vrednost funkcije $h(x)$.

Še ena elegantna rešitev je, da vrednosti $r(x)$ in $r'(z)$ uporabimo kot ključe v razpršeni tabeli (*hash table*); kot pripadajočo vrednost pri takem ključu pa imejmo dva seznama, ki povesta, pri katerih x-ih in pri katerih z-jih se pojavi ta ključ. Če na koncu tega postopka pri kakšnem ključu seznama nista enako dolga, lahko zaključimo, da sta zapisnika nekonsistentna; sicer pa vemo, da mora (za vsak tak par seznamov) g preslikati x-e iz prvega seznama v z-je iz drugega seznama.

Vse, kar smo doslej opisovali za sezname datumov pošiljanja, lahko seveda uporabimo tudi pri datumih oddajanja, iz katerih po enakem postopku rekonstruiramo funkcijo $g(x)$.

Ker je implementacija opisanega postopka v C/C++ precej dolgovezna in ne pretirano zanimiva, zapišimo raje implementacijo v pythonu, ki je precej krajša. Spodnji program za ugotavljanje, kateri $r(x)$ se ujemajo s katerimi $r'(z)$, uporablja razpršeno tabelo:

```
import sys

# Preberimo vhodne podatke.
n, m = map(int, sys.stdin.readline().split())
sprejemniZapisnik = [map(lambda s: int(s) - 1, sys.stdin.readline().split()) for i in range(m)]
oddajniZapisnik = [map(lambda s: int(s) - 1, sys.stdin.readline().split()) for i in range(m)]

# Pripravimo sezname datumov pošiljanja in prejemanja.
# vzorci[1][0][x] vsebuje r(x); vzorci[1][1][z] vsebuje r'(z).
# vzorci[0] vsebuje na enak način dobljene sezname datumov prejemanja.
vzorci = [[[[ ] for x in range(n)] for i in range(2)] for j in range(2)]
for (d, x, y) in sprejemniZapisnik: vzorci[0][1][y].append(d); vzorci[1][0][x].append(d)
for (d, z, x) in oddajniZapisnik: vzorci[0][0][x].append(d); vzorci[1][1][z].append(d)

# Primerjajmo vzorce iz obeh zapisnikov.
# V GH[0] bomo pripravili tabelo funkcije g, v GH[1] pa funkcije h.
GH = [[-1] * n for j in range(2)]
for j in range(2): # j = 0 obdela vzorce prejemanja, j = 1 pa vzorce pošiljanja
    h = {}
    for i in range(2): # imamo dva nabora vzorcev, za vsak zapisnik po enega
        for x in range(n):
            kljuc = tuple(sorted(vzorci[j][i][x]))
            if kljuc not in h: h[kljuc] = ([], [])
            h[kljuc][i].append(x)
    # Za vsak dobljeni ključ poglejmo, kateri agentje iz vsakega
    # zapisnika so imeli tak vzorec pošiljanja oz. prejemanja.
    for (L1, L2) in h.itervalues():
        # Če se dolžini seznamov L1 in L2 ne ujemata, sta zapisnika nekonsistentna.
        # Če imata seznama več kot en element, pa to pomeni, da po postopku iz besedila
        # naloge ne moremo enolično rekonstruirati funkcij g in h.
        assert len(L1) == len(L2)
        for i in range(len(L1)): GH[j][L1[i]] = L2[i]

# Izpišimo rezultate: x, g(x) in h(x) za vsak x.
for x in range(n): print "%d %d %d" % (x + 1, GH[0][x] + 1, GH[1][x] + 1)
```

Razmislimo o tem, kako lahko ta postopek še izboljšamo. Oglejmo si naslednji primer (zaradi preglednosti smo ločili agente od njihovih šifer; agentje so označeni z $x_1, \ldots, x_n$, njihove šifre v oddajnem zapisniku so $y_1, \ldots, y_n$, v sprejemnem pa $z_1, \ldots, z_n$):

Zapisnik sprejemnega kurirja			Zapisnik oddajnega kurirja		
Dan	Pošiljatelj	Prejemnik	Dan	Pošiljatelj	Prejemnik
1	x_1	y_2	1	z_1	x_2
1	x_2	y_1	1	z_2	x_1
2	x_1	y_3	2	z_1	x_3
2	x_2	y_1	2	z_2	x_1

Naš dosedanji postopek bi za agenta x_1 ugotovil, da je poslal dve sporočili, eno na prvi dan in eno na drugi dan; enako bi se izkazalo tudi pri x_2, z_1 in z_2. Ta postopek torej ne bi mogel ugotoviti, ali je $h(x_1) = z_1$ in $h(x_2) = z_2$ ali $h(x_1) = z_2$ in $h(x_2) = z_1$. Oba agenta imata enak seznam datumov oddaj, zato ju ne moremo ločiti med sabo. Toda človek, ki pogleda gornja zapisnika, ju zlahka loči: x_1 je poslal sporočili dvema različnima prejemnikoma, enako tudi z_1; agent x_2 pa je obe svoji sporočili poslal istemu prejemniku, enako pa tudi z_2. Torej velja $h(x_1) = z_1$ in $h(x_2) = z_2$. Koristno je torej, če za vsakega agenta nimamo le enega seznama datumov pošiljanja, pač pa ta seznam razdelimo na podsezname po prejemnikih. Podobno lahko tudi vsak seznam datumov prejemanja razdelimo na podsezname po pošiljateljih.

Še ena izboljšava pa je naslednja. Oglejmo si spet primer iz besedila naloge:

Zapisnik sprejemnega kurirja			Zapisnik oddajnega kurirja		
Dan	Pošiljatelj	Prejemnik	Dan	Pošiljatelj	Prejemnik
1	x_1	y_3	1	z_3	x_2
1	x_2	y_4	1	z_2	x_1
2	x_1	y_2	2	z_3	x_3
2	x_1	y_1	2	z_3	x_4
3	x_3	y_1	3	z_4	x_4
3	x_4	y_4	3	z_1	x_1

Postopek iz besedila naloge ni mogel zanesljivo ugotoviti, ali je $h(x_3) = z_1$ in $h(x_4) = z_4$ ali pa $h(x_3) = z_4$ in $h(x_3) = z_1$, saj imata oba agenta enak vzorec pošiljanja sporočil: oba sta poslala po eno sporočilo in to na dan številka 3. Tudi zamisel iz prejšnjega odstavka, da seznam datumov pošiljanja razdelimo na podsezname po prejemnikih, nam tu ne bo pomagala, saj je bilo poslano eno samo sporočilo in je zato tudi prejemnik en sam. Toda če podrobneje pogledamo zapisnik sprejemnega kurirja, vidimo, da je agent x_3 poslal svoje sporočilo takemu prejemniku (namreč y_1), ki je prejel vsega skupaj dve sporočili, eno na dan 2 in eno na dan 3; agent x_4 pa je poslal svoje sporočilo takemu prejemniku (namreč y_4), ki je prejel vsega skupaj dve sporočili, eno na dan 1 in eno na dan 3. Podobno lahko naredimo tudi za agenta s šiframa z_4 in z_1 v zapisniku oddajnega kurirja; agent z_4 je tam na primer poslal sporočilo nekomu (namreč x_4), ki je prejel dve sporočili, eno na dan 2 in eno na dan 3; agent z_1 pa je tam poslal sporočilo nekomu (namreč x_1), ki je prejel dve sporočili, eno na dan 1 in eno na dan 3. Iz tega lahko zaključimo, da mora šifra z_4 pripadati agentu x_3, šifra z_1 pa agentu x_4: torej $h(x_3) = z_4$, $h(x_4) = z_1$. S tem dodatnim razmislekom smo torej uspeli v tem konkretnem primeru funkcijo h popolnoma zanesljivo rekonstruirati.

Vendar pa se dá tudi ta dodatni razmislek še izboljšati. Oglejmo si naslednji primer:

Zapisnik sprejemnega kurirja			Zapisnik oddajnega kurirja		
Dan	Pošiljatelj	Prejemnik	Dan	Pošiljatelj	Prejemnik
1	x_1	y_1	1	z_3	x_1
1	x_2	y_4	1	z_2	x_4
2	x_3	y_1	2	z_1	x_1
2	x_3	y_2	2	z_1	x_2
2	x_4	y_4	2	z_4	x_4

Postopek iz besedila naloge bi za agente x_1, x_2, z_2 in z_3 opazil, da so vsi poslali le eno sporočilo in to vsi na prvi dan, torej ne bi mogel ugotoviti, ali je $h(x_1) = z_2$ in $h(x_2) = z_3$ ali $h(x_1) = z_3$ in $h(x_2) = z_2$. Delitev seznamov datumov pošiljanja na podsezname po prejemnikih nam tudi ne pomaga, saj je sporočilo vsakič eno samo in je zato tudi prejemnik en sam. Poskusimo uporabiti razmislek iz prejšnjega odstavka: x_1 je poslal svoje sporočilo na dan 1 takemu prejemniku (namreč y_1), ki je prejel vsega skupaj dve sporočili, eno prvi dan in eno drugi dan, od različnih pošiljateljev. Toda enako lahko rečemo tudi za x_2: ta je poslal svoje sporočilo na dan 1 takemu prejemniku (namreč y_4), ki je tudi prejel vsega skupaj dve sporočili, eno prvi in eno drugi dan, od različnih pošiljateljev. Enako velja tudi za z_3 in z_2. Na podlagi tega razmisleka torej še vedno ne moremo ugotoviti, ali je $h(x_1) = z_2$ in $h(x_2) = z_3$ ali $h(x_1) = z_3$ in $h(x_2) = z_2$.

Lahko pa gremo še korak dlje: agent x_1 je poslal svoje sporočilo na dan 1 takemu prejemniku (namreč y_1), ki je prejel skupaj dve sporočili, eno prvi dan od takega pošiljatelja (namreč x_1), ki je poslal le to sporočilo, eno pa drugi dan od takega pošiljatelja (namreč x_3), ki je poslal skupno dve sporočili. Po drugi strani pa je agent x_2 poslal svoje sporočilo na dan 1 takemu prejemniku (namreč y_4), ki je prejel skupaj dve sporočili, eno prvi dan od takega pošiljatelja (namreč x_2), ki je poslal le to sporočilo, eno pa drugi dan od takega pošiljatelja (namreč x_4), ki je tudi poslal le to sporočilo. Zdaj imamo torej o tem, kako sta pošiljala sporočila agenta x_1 in x_2, končno dovolj kompleksen opis, da ju lahko ločimo med seboj. Enak razmislek bi lahko uporabili še na oddajnem zapisniku in se prepričali, da se vzorec pošiljanja agenta z_3 ujema s tistim pri agentu x_1, vzorec pošiljanja agenta z_2 pa s tistim pri agentu x_2, torej je $h(x_1) = z_3$ in $h(x_2) = z_2$.

Opisani postopek si bomo lažje predstavljali, če ga zapišemo malo bolj matematično. Imejmo zapisnik L, ki je vreča (multimnožica)[12] trojic iz $D \times U \times V$, pri čemer je D množica datumov, U množica pošiljateljev in V množica prejemnikov. Definirali bomo zaporedje vse bolj kompleksnih opisov tega, kdaj in s kakšnimi ljudmi si nek agent izmenjuje sporočila. Za vsak $t \geq 0$ bomo definirali za vsakega pošiljatelja $u \in U$ njegov *vzorec pošiljanja* $r_t(u)$ in za vsakega prejemnika $v \in V$ njegov *vzorec prejemanja* $s_t(v)$. Začnemo z $r_0(u) = \{\}$ za vsak $u \in U$ ter $s_0(v) = \{\}$ za vsak $v \in V$; nadaljujemo pa z

$$r_{t+1}(u) = \{\!\{(s_t(v), \{\!\{d : (d, u, v) \in L\}\!\}) : v \in V\}\!\}$$
$$s_{t+1}(v) = \{\!\{(r_t(u), \{\!\{d : (d, u, v) \in L\}\!\}) : u \in U\}\!\}.$$

[12] Vreča (angl. *bag*) oz. multimnožica je matematična struktura, ki se od običajne množice razlikuje po tem, da je v njej lahko posamezen element prisoten tudi po večkrat. V nadaljevanju jih bomo označevali s simboli $\{\!\{\ldots\}\!\}$ namesto $\{\ldots\}$. Primer: $\{1, 2\}$ in $\{1, 1, 2\}$ sta le dva različna zapisa ene in iste množice; $\{\!\{1, 2\}\!\}$ in $\{\!\{1, 1, 2\}\!\}$ pa sta dve različni vreči. Vrstni red elementov pa pri vrečah ni pomemben (enako kot pri navadnih množicah); tako je npr. $\{\!\{1, 1, 2\}\!\} = \{\!\{1, 2, 1\}\!\}$.

Vzorec oddajanja $r_{t+1}(u)$ torej sestavimo tako, da za vsakega možnega prejemnika v zapišemo njegov vzorec prejemanja $s_t(v)$ in dneve, na katere je ta prejemnik dobival sporočila od u. Podobno sestavimo $s_{t+1}(v)$ s pomočjo vzorcev $r_t(u)$. Začnemo s trivialnimi vzorci r_0 in s_0, ki nam o agentih ne povedo nič; na naslednjem koraku imamo vzorce r_1 in s_1, ki so že malo bolj zmogljivi od tistih, ki jih uporablja postopek iz besedila naloge, saj na primer za vsakega pošiljatelja u ne dobimo enega samega seznama datumov poslanih sporočil, pač pa je tak seznam razdeljen na podsezname po prejemnikih. Vzorce r_2 in s_2 smo uporabili v zadnjem od zgoraj naštetih primerov (tistem s štirimi agenti in petimi sporočili). Ni pa nujno, da se ustavimo pri $t = 2$; načeloma bi se dalo sestaviti iz njih še bolj kompleksne vzorce r_3, s_3 in tako naprej.

Kakšen je zdaj postopek za reševanje naloge s takšnimi vzorci? Naj bo X množica agentov, Y množica šifer zanje iz sprejemnega zapisnika in Z množica šifer iz oddajnega zapisnika. Sprejemni zapisnik L_s je torej vreča trojic iz $D \times X \times Y$, oddajni zapisnik L_o pa vreča trojic iz $D \times Z \times X$. Izberimo si nek t in iz sprejemnega zapisnika sestavimo vse vzorce $r_t(x)$ in $s_t(y)$, iz oddajnega zapisnika pa vse vzorce $r'_t(z)$ in $s'_t(x)$ (označili smo jih s črtico $'$, da jih bomo lažje ločili od tistih iz sprejemnega zapisnika). Ne glede na to, kako velik t si izberemo in kako kompleksne vzorce dobimo, so ti vzorci na koncu še vedno sestavljeni le iz datumov (torej elementov množice D), ki so zloženi skupaj v vreče, te so zložene v pare, ti pari spet v vreče in tako naprej. Zato lahko vzorce, dobljene iz različnih zapisnikov, primerjamo med sabo. Agentu x lahko pripada šifra $h(x) = z$ le, če je $r_t(x) = r'_t(z)$. Postopek reševanja je pravzaprav enak tistemu iz besedila naloge, le da so uporabljeni vzorci bolj kompleksni, zato imamo več možnosti, da bomo pravilno in enolično rekonstruirali šifrirni funkciji g in h. Če za nek x ne obstaja noben z, ki bi imel $r_t(x) = r'_t(z)$, potem vemo, da sta zapisnika nekonsistntna. Če ima po več x-ov (in več z-jev) enak vzorec $r_t(x)$ oz. $r'_t(z)$, pa za te agente tudi zdaj ne bomo mogli zanesljivo ugotoviti, kateremu pripada katera šifra.

Za primer iz besedila naloge smo zgoraj videli, da imata agenta 3 in 4 enak vzorec pošiljanja r_1, vendar različen vzorec pošiljanja r_2, tako da bi lahko šifrirni funkciji s $t = 2$ že popolnoma identificirali, s $t = 1$ pa še ne. Izkaže se, da za vsak t obstajajo primeri, kjer ta t še ne zadošča, $t + 1$ pa že; poleg tega obstajajo tudi primeri, pri katerih sploh noben t ne zadošča. Vprašamo se torej lahko, kako velik t je pametno vzeti pri posameznem testnem primeru.

Pokazati je mogoče (dokaz prepustimo bralcu za vajo), da za vsak t velja naslednje: če imata dva agenta u in u' enak vzorec r_{t+1}, imata tudi enak vzorec r_t; z drugimi besedami, $r_{t+1}(u) = r_{t+1}(u') \Rightarrow r_t(u) = r_t(u')$; in enako velja tudi za vzorce s_t in s_{t+1}.

Predstavljajmo si zdaj relacijo R_t, definirano takole: $u\ R_t\ u'$ natanko tedaj, ko je $r_t(u) = r_t(u')$; podobno definirajmo še S_t. Hitro se vidi, da so te relacije ekvivalenčne; posamezni ekvivalenčni razred tvorijo vsi agentje, ki imajo enak vzorec r_t oz. s_t. Ker smo zgoraj videli, da iz $r_{t+1}(u) = r_{t+1}(u')$ sledi $r_t(u) = r_t(u')$, to pomeni, da če sta u in u' v istem ekvivalenčnem razredu relacije R_{t+1}, sta tudi v istem ekvivalenčnem razredu relacije R_t. Ekvivalenčni razredi relacije R_{t+1} nastanejo torej z drobitvijo ekvivalenčnih razredov relacije R_t; zato ima R_{t+1} vsaj toliko ekvivalenčnih razredov kot R_t in če imata obe enako število ekvivalenčnih razredov, sta si popolnoma enaki. Pokazati je mogoče tudi, da če imata R_t in R_{t+2} enako število

ekvivalenčnih razredov, potem so jima enake tudi vse nadaljnje relacije: R_t, R_{t+1}, R_{t+2}, R_{t+3} in tako naprej. Ker pa imamo le n agentov, tudi število ekvivalenčnih razredov ne more preseči n, torej se nam gotovo ne bo treba ukvarjati z relacijami R_t za $t > 2n$. V praksi je smiselno povečevati t le tako dolgo, dokler pri kakšni od relacij R_t, S_t, R'_t in S'_t še prihaja do drobitev na manjše ekvivalenčne razrede; dobro pa je vedeti, da nam dlje kot do $t = 2n$ gotovo ne bo treba iti.

Želimo torej si, da bi pri nekem t naša množica agentov razpadla na n ekvivalenčnih razredov, torej da ima vsak agent svoj vzorec pošiljanja (oz. oddajanja), po katerem se razlikuje od vseh ostalih. Takrat lahko s primerjavo vzorcev $r_t(x)$ in $r'_t(z)$ enolično rekonstruiramo funkcijo h, s primerjavo vzorcev $s'_t(x)$ in $s_t(y)$ pa funkcijo g. Žal pa se lahko zgodi, da do tega ne pride, četudi vzamemo še tako velik t. Oglejmo si naslednji primer:

Zapisnik sprejemnega kurirja			Zapisnik oddajnega kurirja		
Dan	Pošiljatelj	Prejemnik	Dan	Pošiljatelj	Prejemnik
1	x_1	y_1	1	z_1	x_2
1	x_2	y_2	1	z_2	x_1

Hitro se lahko prepričamo, da imata v sprejemnem zapisniku x_1 in x_2 enak vzorec pošiljanja, y_1 in y_2 pa enak vzorec prejemanja, ne glede na to, kako velik t vzamemo. Podobno velja tudi za oddajni zapisnik.

Če imamo primer, ko ima (tudi pri še tako velikem t) več agentov enak vzorec pošiljanja (ali prejemanja), to še ne pomeni, da je bil naš dosedanji razmislek brez haska. Vendarle smo pridobili nekaj koristnih omejitev glede tega, kakšni morata biti funkciji g in h. Za vsak vzorec pošiljanja, recimo p, imamo en ekvivalenčni razred iz X/R_t, v katerem so tisti $x \in X$, ki imajo $r_t(x) = p$, in en ekvivalenčni razred iz Z/S_t, v katerem so tisti $z \in Z$, ki imajo $r'_t(z) = p$. Če tadva ekvivalenčna razreda nista enako velika, vemo, da sta vhodna zapisnika nekonsistentna, sicer pa vemo vsaj to, da mora funkcija h vse x-e iz prvega ekvivalenčnega razreda preslikati v z-je iz drugega ekvivalenčnega razreda. Tako smo torej vsaj malo omejili nabor možnih vrednosti $h(x)$ za posamezne konkretne x. Podobno nam primerjava vzorcev $s'_t(x)$ in $s_t(y)$ dá nekaj omejitev glede tega, kakšne so možne vrednosti $g(x)$ za posamezne konkretne x.

Če imamo na primer pri relacijah R_t oz. R'_t dva ekvivalenčna razreda, enega s tremi in enega s štirimi agenti, imamo pri prvem razredu 3! možnosti glede tega, kako preslikamo agente v njihove šifre, pri drugem razredu pa 4! možnosti; tako imamo torej skupaj $3! \cdot 4!$ kandidatk za funkcijo h, ki ustrezajo tem omejitvam. Načeloma jih tudi ne bi bilo težko našteti, na primer z rekurzijo (čeprav njihovo število hitro postane neobvladljivo veliko). Podobno lahko storimo tudi za funkcije g. Zavedati pa se moramo, da če imamo neko funkcijo g in neko funkcijo h, ki vsaka zase ustrezata tem omejitvam (torej da za vsak x velja $r_t(x) = r'_t(h(x))$ in $s'_t(x) = s_t(g(x))$), to še ne pomeni, da ta par funkcij skupaj tvori veljavno rešitev naše naloge. Pri zgornjem primeru z dvema agentoma smo videli, da imajo relacije R_t, R'_t, S_t in S'_t vse po en ekvivalenčni razred. Funkcija $g_1 := \{(x_1, y_1), (x_2, y_2)\}$ ustreza tem omejitvam, enako tudi $g_2 := \{(x_1, y_2), (x_2, y_3)\}$, $h_1 := \{(x_1, z_1), (x_2, z_2)\}$ in $h_2 := \{(x_1, z_2), (x_2, z_1)\}$. Toda če vzamemo zdaj skupaj funkciji g_1 in h_2 ter s prvo dešifriramo oddajni zapisnik, z drugo pa sprejemni zapisnik, vidimo, da se dešifrirana

zapisnika ne ujemata. Podobno se zgodi tudi, če uporabimo g_2 in h_1. Veljavna para funkcij pri tem primeru sta le (g_1, h_1) in (g_2, h_2).

Vidimo torej, da funkcij g in h ne smemo izbirati neodvisno drugo od druge. Če fiksiramo vrednost $g(x)$ za nekaj konkretnih agentov x, lahko ta odločitev vpliva na to, katere h so še dopustne, in obratno. Naloge bi se torej lahko lotili s postopkom, ki se zgleduje po logičnem programiranju z omejitvami (*constraint logic programming*, CLP). Za vsak x vzdržujemo množico možnih vrednosti $g(x)$ in množico možnih vrednosti $h(x)$. Na začetku napolnimo te množice s tistimi agenti, ki se z x-om ujemajo v vzorcu pošiljanja oz. prejemanja (r_t oz. s_t). Nato si izberimo nek tak x, pri katerem je na primer za $h(x)$ še več kot ena možna vrednost. Izberimo eno od njih, na primer z, in poglejmo, kaj se zgodi, če postavimo $h(x) := z$. Takoj lahko vse trojice, ki imajo v L_o pošiljatelja z, dekodiramo: če je v L_o trojica (d, z, x'), mora biti v nešifriranem zapisniku (recimo mu L) trojica (d, x, x'). Zdaj si lahko za vsak x' pripravimo seznam (recimo mu $Q_x(x')$), v katerem za vsak dan d piše, koliko sporočil je x poslal agentu x' na dan d. Podobno si lahko tudi iz L_s za vsak $y \in Y$ pripravimo seznam (recimo mu $Q'_x(y)$), v katerem za vsak d piše, koliko sporočil je x poslal agentu s šifro y na dan d. Možne vrednosti za $g(x')$ so le tisti y, za katere je $Q_x(x') = Q'_x(y)$. Na ta način lahko torej zalogo možnih vrednosti za $g(x')$ pri nekaterih x' še dodatno omejimo. Lahko se celo izkaže, da se zaradi te dodatne omejitve zaloga vrednosti $g(x')$ pri kakšnem konkretnem x' popolnoma izprazni; to je znak, da je bila naša predpostavka $h(x) = z$ napačna in moramo poskusiti prirediti $h(x)$ kakšno drugo vrednost. Drugače pa lahko zdaj rekurzivno nadaljujemo: spet si izberemo nekega agenta, poskusimo zanj izbrati neko konkretno vrednost funkcije g ali h in pogledamo, kako to vpliva možne vrednosti teh dveh funkcij pri ostalih agentih. Na ta način bomo sčasoma našli vse veljavne pare funkcij (g, h), pri tem pa, upajmo, sproti čim bolj omejevali prostor preiskovanja.

12. Mehurčki v slamici

Nalogo lahko rešimo s simulacijo. Mehurčki se premikajo s konstantno hitrostjo in do sprememb pride le, ko en mehurček ujame drugega in se združi z njim. Določiti moramo torej, kateri par mehurčkov se združi najprej. V poštev pridejo le primeri, ko se mehurček združi s tistim tik nad (ali pod) njim; če imamo na primer po vrsti enega nad drugim mehurčke a, b in c, se a ne more združiti s c, ne da bi se pred tem najprej združil z b. Torej nam ni treba gledati vseh možnih parov mehurčkov, pač pa le take pare, kjer sta mehurčka eden tik nad drugim v slamici (in med njima ni nobenega tretjega). Zato je koristno vzdrževati seznam, v katerem so mehurčki urejeni naraščajoče po višini; za začetek torej uredimo mehurčke iz vhodnih podatkov po višini, pri čemer lahko uporabimo kateregakoli od mnogih znanih algoritmov za urejanje (na primer quicksort).

Oglejmo si zdaj dva zaporedna mehurčka, recimo i in $i + 1$. Prvi se nahaja na višini h_i in se giblje s hitrostjo v_i, drugi pa je na višini h_{i+1} (ki je $> h_i$) in se giblje s hitrostjo v_{i+1}. Čez t časovnih enot bosta torej na višinah $h_i + v_i t$ in $h_{i+1} + v_{i+1} t$; ujameta se torej po času $t_i := (h_{i+1} - h_i)/(v_i - v_{i+1})$. Če je $v_i \leq v_{i+1}$, pa spodnji mehurček zgornjega sploh ne bo mogel ujeti.

Načeloma gremo torej lahko v zanki po i, pri vsakem izračunamo t_i in si zapomnimo najmanjšega med njimi. Nato se še enkrat zapeljemo po seznamu in popra-

vimo položaj mehurčkov tako, da bo odražal stanje t_i časovnih enot za dosedanjim: mehurček j se premakne s h_j na $h_j + v_j t_i$. Tiste, ki so pri tem presegli višino 1, pobrišemo (in ustrezno povečamo števec mehurčkov, ki so dosegli vrh). Nato, če vidimo dva ali več zaporednih mehurčkov na isti višini, jih združimo v enega (odvečne elemente pobrišemo iz seznama in izračunamo premer in hitrost novega mehurčka); če je ta na višini točno 1, pobrišemo tudi njega (in povečamo števec mehurčkov, ki so dosegli vrh). Postopek ponavljamo, dokler se seznam ne izprazni.

Posebej moramo paziti na primer, ko pri vsakem i velja $v_i \le v_{i+1}$; takrat ne bo prišlo do nobene združitve več in postopek lahko takoj končamo (števec mehurčkov, ki bodo dosegli vrh, pa povečamo za število vseh še preostalih mehurčkov našega seznama).

Vidimo, da se po vsaki iteraciji glavne zanke število mehurčkov zmanjša vsaj za 1 (ker se vsaj dva mehurčka združita v enega), torej bomo izvedli kvečjemu $n - 1$ iteracij; in ker nam vsaka iteracija vzame $O(n)$ časa (dvakrat se moramo sprehoditi čez cel seznam: prvič računamo vse t_i, drugič pa popravljamo položaje mehurčkov in jih združujemo), je časovna zahtevnost celotnega postopka $O(n^2)$.

Hitrejšo rešitev dobimo, če čase t_i hranimo v kopici (*heap*); to je drevesasta podatkovna struktura, pri kateri so manjše vrednosti pri vrhu (najmanjša vrednost sploh je v korenu drevesa), dodajanje in brisanje posameznega elementa pa traja $O(\log n)$ časa. Tako bomo lahko hitro in učinkovito ugotovili, katera je prva naslednja združitev dveh mehurčkov; razmisliti pa moramo še o tem, kako se izogniti ostalim operacijam, zaradi katerih je naša prvotna rešitev porabila $O(n)$ časa za vsako združitev: to je računanje časov t_i za vse mehurčke pred združitvijo in računanje novih položajev h_i za vse mehurčke po združitvi. Koristno je, če čase t_i definiramo relativno glede na začetek cele simulacije, ne pa relativno glede na zadnjo prejšnjo združitev; tako bo treba ob združitvi na novo izračunati le t_i združenega mehurčka in tistega pod njim (t_{i-1}). Glede položajev h_i pa je koristno, če ob položaju zapišemo še čas (recimo u_i), ob katerem je bil mehurček v tem položaju; iz tega lahko izračunamo tudi položaj mehurčka v kasnejših trenutkih, saj je njegova hitrost konstantna (vsaj do naslednje združitve). Mehurčke imejmo povezane v dvojno povezano verigo, da bomo po združitvah lažje vzdrževali podatke o tem, kateri mehurček je tik nad ali pod nekim drugim; v spodnji psevdokodi to verigo predstavlja seznam L, ki naj bo sestavljen iz dveh tabel, $nad[i]$ oz. $pod[i]$, ki povesta, kateri mehurček je neposredno nad oz. pod mehurčkom i; najnižji mehurček naj ima $pod[i] = -1$, najvišji naj ima $nad[i] = -1$, če pa sta oba enaka -1, to pomeni, da i sploh ni več v seznamu.

$H :=$ prazna kopica; $T := 0$; $L :=$ prazen dvojno povezan seznam; *rezultat* $:= 0$;
for $i := 1$ **to** n: $v_i := f(d_i)$; $u_i := 0$; dodaj i na konec seznama L;
for $i := 1$ **to** $n - 1$: DodajTrk(i)

while H ni prazna:
  vzemi iz H tisti i, ki ima najmanjšo vrednost t_i;
  if i ni v seznamu L: **continue**
  $h := h_i + (t_i - u_i)v_i$; (* *višina, pri kateri se i združi z naslednjim mehurčkom nad sabo* *)
  if $h > 1$: (* *i doseže vrh slamice, preden dohiti tistega nad sabo* *)
    povečaj *rezultat* za 1;
    $j := pod[i]$; pobriši i iz seznama L; DodajTrk(j)

else: (* *mehurčka se res združita na višini h, še preden prideta do vrha* *)
$j := nad[i]$; pobriši j iz seznama L; (* *morali bi ga tudi iz kopice, vendar je lažje, če ga pustimo v njej in ga kasneje ignoriramo* *)
$h_i := h$; $u_i := t_i$; $d_i := (d_i^2 + d_j^2)^{1/2}$; $v_i := f(d_i)$;
DodajTrk(i); DodajTrk($pod[i]$);

(* *Zdaj so ostali v L le mehurčki, ki ne bodo ujeli tistega nad sabo.* *)
povečaj *rezultat* za število elementov seznama L;

Na koncu tega postopka imamo v spremenljivki *rezultat* število mehurčkov, ki dosežejo vrh slamice (ob upoštevanju združitev). Za dodajanje časov trkov v kopico uporabljamo naslednji podprogram:

podprogram DodajTrk(i):
if $i < 0$: **return**
$j := nad[i]$; **if** $j < 0$: **return**
if $v_i \le v_j$: **return** (* i ne bo nikoli ujel j-ja *)
(* *Rešimo enačbo $h_i + (t - u_i)v_i = h_j + (t - u_j)v_j$.* *)
$t_i := (h_j - h_i - u_j v_j + u_i v_i)/(v_i - v_j)$;
če je i že v kopici H, popravi tam njegov ključ na t_i,
sicer dodaj i v kopico H z vrednostjo t_i kot ključem;

13. Kitka

Stanje trakov lahko predstavimo s seznamom. Na začetku dodamo v seznam po vrsti števila od 1 do n, kar predstavlja začetni razpored trakov; nato pa moramo za vsak korak (a_i, b_i) pobrisati iz seznama število a_i in ga vriniti takoj za številom b_i. Na koncu moramo izpisati m-ti element seznama.

Glavno vprašanje je zdaj v tem, kako zagotoviti, da bodo vse te operacije čim bolj učinkovite. Ker potrebujemo brisanje in vrivanje elementov na bolj ali manj poljubnih mestih v seznamu, je koristno seznam predstaviti z dvojno povezano verigo (*doubly linked list*), v kateri sta ob vsakem elementu kazalca na prejšnji in naslednji element seznama; tako bomo lahko brisali in vrivali elemente v času $O(1)$.

Da bomo lahko element a_i učinkovito vrinili za b_i, moramo biti sposobni hitro ugotoviti, kje v seznamu element b_i sploh je. Lahko bi si omislili nekakšno kazalo — tabelo, v kateri je vsako število od 1 do n kazalec na tisti člen verige, v katerem se nahaja to število. Še lažje pa je, če seznam (verigo) predstavimo kar z dvema tabelama, recimo jima *pred* in *nasl*: za vsak element x (od 1 do n) naj nam $pred[x]$ pove, kateri je neposredni predhodnik števila x v našem seznamu, $nasl[x]$ pa, kateri je x-ov neposredni naslednik. Prvi člen seznama naj ima $pred[x] = 0$, zadnji pa $nasl[x] = 0$.

(* *Inicializacija seznama.* *)
for $x := 1$ **to** n **do** $pred[x] := x - 1$; $nasl[x] := x + 1$;
$prvi := 1$; $nasl[n] := 0$;

for $i := 1$ **to** k:

(* *Pobrišimo a_i iz seznama.* *)
$p := pred[a_i]$; $s := nasl[a_i]$;

if $p = 0$ **then** $prvi := s$ **else** $nasl[p] := s$;
if $s > 0$ **then** $pred[s] := p$;

(* *Vrinimo* a_i *za* b_i. *)
$pred[a_i] := b_i$; $nasl[a_i] := nasl[b_i]$; $nasl[b_i] := a_i$;

(* *Izpišimo m-ti element seznama.* *)
$x := prvi$; **for** $i := 2$ **to** m **do** $x := nasl[x]$;
izpiši x;

Časovna zahtevnost te rešitve je $O(n + k)$ — najprej potrebujemo $O(n)$ časa za inicializacijo seznama, nato $O(1)$ časa za vsako od k prepletanj in na koncu še $O(m)$ (kar je tudi $O(n)$) časa za iskanje m-tega elementa.

Za primere, ko je n veliko večji od k, pa lahko sestavimo še učinkovitejšo rešitev, ki bo porabila manj časa in tudi manj pomnilnika (gornja rešitev porabi $O(n)$ prostora za vzdrževanje seznama elementov). Pomagali si bomo z dejstvom, da če je število prepletanj (torej k) majhno, se seznam še ni mogel prav dosti spremeniti od svojega začetnega stanja; večino seznama še vedno sestavljajo območja (recimo jim *čete*), kjer si elementi sledijo lepo po vrsti: x, $x + 1$, $x + 2$ in tako naprej. Vsako tako četo lahko opišemo preprosto s tem, da si zapomnimo njen prvi člen in njeno dolžino; na primer, par $(6, 5)$ pomeni četo z elementi 6, 7, 8, 9, 10. Tako lahko na primer seznam

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

krajše zapišemo kot

$$(1, 4), (6, 5), (5, 1), (11, 3).$$

Začetni seznam $1, 2, \ldots, n$ je že sam po sebi ena sama dolga četa, namreč $(1, n)$.

Kaj se zgodi, ko element a_i premaknemo za element b_i? Naj bo A četa, v kateri je pred tem prepletanjem ležal element a_i; znotraj nje naj bo A' četa, ki jo tvorijo elementi levo od a_i, in A'' četa, ki jo tvorijo elementi desno od a_i. Podobno naj bo B četa, v kateri je pred tem prepletanjem ležal element b_i; znotraj nje naj bo B' četa, ki jo tvori element b_i in vsi elementi levo od njega, in B'' naj bo četa, ki jo tvorijo elementi desno od b_i. Če je imel torej prej celoten seznam obliko

$$\alpha, A, \beta, B, \gamma$$

oz. (kar je isto)

$$\alpha, A', a_i, A'', \beta, B', B'', \gamma$$

(pri čemer α, β in γ predstavljajo poljubna zaporedja 0 ali več čet), bo imel po naslednjem prepletanju obliko

$$\alpha, A', A'', \beta, B', a_i, B'', \gamma,$$

pri čemer zdaj a_i tvori sam svojo četo. Vidimo torej, da se z enim prepletanjem število čet poveča za 3. Ker je imel seznam na začetku eno samo četo, bo imel po k prepletanjih skupno $3k + 1$ čet (ali pa še manj, če upoštevamo, da so nekatere od teh čet lahko dolge 0 elementov in jih smemo pobrisati — na primer, če je a_i prvi element čete A, bo četa A' prazna); in ker je vsaka četa predstavljena le z dvema številoma, bomo za takšno predstavitev seznama porabili le $O(k)$ prostora namesto $O(n)$.

Razmisliti moramo še o tem, kako učinkovito ugotoviti, v kateri četi leži a_i in v kateri b_i. Če pogledamo začetne elemente vseh čet in poiščemo med njimi največjega takega, ki je manjši ali enak x, je to ravno začetni element čete, v kateri leži x.[13] Ena možnost je torej, da se preprosto sprehodimo po seznamu vseh čet in poiščemo pravo, vendar nam bo to vzelo $O(k)$ časa; in ker moramo to narediti pri vsakem prepletanju, teh pa je k, bo časovna zahtevnost celotnega postopka $O(k^2)$. Učinkovitejšo rešitev dobimo, če začetne elemente čet hranimo v primerno uravnoteženem binarnem iskalnem drevesu,[14] v katerem bomo lahko najmanjši tak element, ki je manjši ali enak b_i, poiskali že v $O(\log k)$ časa, pa tudi dodajanje novega elementa v tako drevo zahteva le $O(\log k)$ časa. Ob vsakem začetnem elementu čete mora drevo seveda vsebovati tudi kazalec na četo samo (oz. številko čete ali nekaj podobnega, s pomočjo nečesa bomo lahko takoj prišli do te čete v seznamu). Časovna zahtevnost celotnega postopka je tako $O(k \log k)$, kar je bolje od $O(n+k)$, če je k dovolj manjši od n.

Zapišimo novi postopek še s psevdokodo. Čete si mislimo oštevilčene od 1 naprej in povezane v dvojno verigo s pomočjo tabel *pred* in *nasl*; prvi element čete i naj bo $z[i]$, njena dolžina pa $d[i]$. Spremenljivka L hrani skupno število čet.

(* *Inicializacija.* *)
$L := 1$; *prva* $:= 1$; $z[1] := 1$; $d[1] := d$; *pred*$[1] := 0$; *nasl*$[1] := 0$;
$T :=$ prazno drevo; dodaj v T ključ $z[1]$ s spremljevalno vrednostjo 1;

for $i := 1$ **to** k:

(* *Poiščimo četi A in B, v katerih ležita elementa a_i in b_i.* *)
poišči v T največji ključ, ki je $\le a_i$, in naj bo A njegova spremljevalna vrednost;
poišči v T največji ključ, ki je $\le b_i$, in naj bo B njegova spremljevalna vrednost;
$d_A := d[A]$; $d_B := d[B]$;

(* *Alocirajmo nove čete. Četi A' oz. B' bomo zapisali kar čez četi A oz. B.* *)
$A' := A$; $d[A'] := a_i - z[A]$;
$A'' := L + 1$; $z[A''] := a_i + 1$; $d[A''] := d_A - d[A'] - 1$;
$A''' := L + 2$; $z[A'''] := a_i$; $d[A'''] := 1$;
$B' := B$; $d[B'] := b_i + 1 - z[B]$;
$B'' := L + 3$; $z[B''] := b_i + 1$; $d[B'] := d_B - d[B']$;
$L := L + 3$;

(* *Dodajmo nove čete v drevo.* *)
for each $c \in \{A'', A''', B''\}$:
dodaj v T ključ $z[c]$ s spremljevalno vrednostjo c;

[13] O tem se lahko prepričamo takole. Naj bo r največji začetni element, ki je $\le x$; in naj bo s začetni element čete, ki vsebuje število x. Recimo, da je $s > r$. Če bi zdaj veljalo $s \le x$, bi r ne bil največji tak začetni element, ki je $\le x$; to bi bilo protislovje, torej je $s > x$. Toda ker so elementi v četi naraščajoči, je nemogoče, da bi x ležal v četi, ki ima začetni element (v našem primeru s) večji od x. Tako torej možnost $s > r$ odpade. Recimo zdaj, da je $s < r$; imamo torej $s < r \le x$; če x leži v četi, ki se začne z elementom s, mora ta četa vsebovati tudi vse elemente med njima, torej tudi r, kar je v nasprotju s predpostavko, da je r začetni element neke čete. Ker sta nas tako predpostavki $s > r$ kot $s < r$ pripeljali v protislovje, lahko zaključimo, da sta s in r enaka.

[14] Pri uravnoteževanju drevesa nam lahko pomaga tudi dejstvo, da že vnaprej poznamo vse možne elemente, ki bi se utegnili kdaj znajti v njem: to so poleg vrednosti 1 še števila a_i, $a_i + 1$ in $b_i + 1$ za vsak i.

```
(* Za A' (bivšo A) vrinimo A''. *)
s := nasl[A]; nasl[A''] := s; if s > 0 then pred[s] := A'';
nasl[A'] := A''; pred[A''] := A';
(* Za B' (bivšo B) vrinimo A''' in B''. *)
s := nasl[B]; nasl[B''] := s; if s > 0 then pred[s] := B'';
nasl[B'] := A'''; pred[A'''] := B';
nasl[A'''] := B''; pred[B''] := A''';
(* Popravimo drevo in pobrišimo morebitne prazne čete. *)
for each c ∈ {A', A'', B', B''}:
  if d[c] > 0:
    v drevo T dodaj ključ z[c] (če ga še ni v njem);
    v T postavi spremljevalno vrednost pri ključu z[c] na c;
  else: (* pobrišimo prazno četo *)
    p := pred[c]; s := nasl[c];
    if p = 0 then prva := s else nasl[p] := s;
    if s > 0 then pred[s] := p;

(* Izpišimo m-ti element. *)
c := prva; while m > d[c]: m := m − d[c]; c := nasl[c]
izpiši z[c] + m − 1;
```

14. Tovor

Neenačbe lahko predstavimo z usmerjenim grafom $G = (V, E)$, ki ima točke $V = \{1, 2, \ldots, n\}$ (po eno za vsako spremenljivko $a_1, a_2, \ldots, a_n$) in m povezav: za vsako neenačbo $a_i < a_j$ dodajmo v graf povezavo $i \to j$. Če v tem grafu obstaja pot $i_1 \to i_2 \to \ldots \to i_k$, to pomeni, da imamo v podatkih neenačbe $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$, torej je v vsaki dopustni rešitvi a_{i_1} manjši od a_{i_k}. Če bi poleg tega obstajala še povezava $i_k \to i_1$, bi to pomenilo, da mora biti hkrati tudi a_{i_k} manjši od a_{i_1}, kar je nemogoče; tak nabor neenačb torej sploh ne bi imel rešitev, besedilo naloge pa zagotavlja, da se to ne bo zgodilo. Naš graf G je torej acikličen.

Recimo zdaj, da nas zanima vrednost $s(k)$ oz. $|S(k)|$; z drugimi besedami, zanima nas, koliko je takih spremenljivk, ki imajo pri vseh dopustnih rešitvah drugačno vrednost od a_k. Če za neko i obstaja v G pot od i do k, smo v prejšnjem odstavku videli, da pri vsaki dopustni rešitvi velja $a_i < a_k$; in podobno, če obstaja pot od k do i, to pomeni, da pri vsaki dopustni rešitvi velja $a_k < a_i$. Množica $S(k)$ torej vsebuje vse posredne in neposredne prednice in potomke točke k v grafu.

Kaj pa ostale točke, torej tiste, ki niso niti k-jeve prednice niti potomke? Naj bo t poljubna taka točka in naj bo $\mathbf{a} \in A$ poljubna rešitev našega sistema neenačb. Ločimo tri možnosti: $a_t = a_k$, $a_t > a_k$ in $a_t < a_k$.

(1) Če pri tej rešitvi velja $a_t = a_k$, vidimo, da a_t ni vedno (v vsaki dopustni rešitvi) različna od a_k, torej $t \notin S(k)$.

(2) Če pri tej rešitvi velja $a_t > a_k$, jo predelajmo tako, da vrednost a_t in vseh njenih prednic zmanjšamo za $a_t - a_k$; dobljeni novi n-terici recimo $\mathbf{a}'$. Izkaže se, da je $\mathbf{a}'$ še vedno dopustna rešitev. O tem se lahko prepričamo takole: vzemimo poljubno neenačbo, recimo $a_i < a_j$.

(2.1) Če nobena od i in j ni prednica t-ja (med slednje štejmo tudi t samo), je $a'_i = a_i$ in $a'_j = a_j$, zato ta neenačba velja tudi pri $\mathbf{a}'$.

(2.2) Če je j prednica t-ja, mora biti tudi i prednica t-ja (saj obstoj neenačbe $a_i < a_j$ pomeni, da imamo v grafu povezavo $i \to j$), torej sta se pri predelavi $\mathbf{a}$ v $\mathbf{a}'$ obe zmanjšali za enako vrednost (namreč $a_t - a_k$), tako da je $a'_i = a_i - (a_t - a_k) < a_j - (a_t - a_k) = a'_j$ in neenačba drži tudi pri $\mathbf{a}'$.

(2.3) Ostane še možnost, da je i prednica t-ja, j pa ne. Torej je $a'_i = a_i - (a_t - a_k)$ in $a'_j = a_j$. Ker je $a_t > a_k$, je $a_t - a_k > 0$ in $a'_i = a_i - (a_t - a_k) < a_i < a_j = a'_j$, torej je naša neenačba izpolnjena tudi pri $\mathbf{a}'$.

Ta razmislek lahko opravimo za vse neenačbe in tako vidimo, da je $\mathbf{a}'$ res dopustna rešitev. Ker t ni niti prednica niti potomka k-ja, sledi, da tudi k ni prednica t-ja, torej se a_k pri predelavi $\mathbf{a}$ v $\mathbf{a}'$ ni spremenila: $a'_k = a_k$. Po drugi strani se a_t je spremenila, in sicer smo jo zmanjšali za $a_t - a_k$, torej ima zdaj vrednost $a'_t = a_t - (a_t - a_k) = a_k = a'_k$. Prišli smo torej do dopustne rešitve, v kateri sta spremenljivki k in t enaki, tako da lahko zaključimo, da $t \neq S(k)$.

(3) Če pa bi pri $\mathbf{a}$ veljalo $a_t < a_k$, bi lahko razmišljali analogno kot pri točki (2), le da bi morali vse t-jeve potomke (vključno s t sámo) povečati za $a_k - a_t$.

Tako torej vidimo, da če t ni niti potomka niti prednica točke k, potem gotovo obstaja taka dopustna rešitev, v kateri je $a_t = a_k$, tako da takšna t ne leži v množici $S(k)$. Slednja torej vsebuje natanko vse prednice in potomke točke k in ničesar drugega. Te točke pa lahko naštejemo s kakšnim od znanih postopkov za pregledovanje grafa, na primer z iskanjem v širino ali pa v globino. To nam vzame $O(n+m)$ časa in ker moramo ta postopek izvesti za vse možne k (teh pa je n), je časovna zahtevnost celotne rešitve $O(n(n+m))$.

Oglejmo si zdaj še drugo različico naloge, pri kateri iščemo maksimum funkcije s', definirane kot $s'(k) := \min_{\mathbf{a}\in A} |r(k, \mathbf{a})|$. Z drugimi besedami, s' nam pove, da se pri vsaki dopustni rešitvi $\mathbf{a} \in A$ od a_k razlikuje vsaj $s'(k)$ spremenljivk, vendar pa so to pri različnih $\mathbf{a}$ lahko različne spremenljivke. Za nadaljevanje našega razmišljanja se izkaže za bolj prikladno, če vprašanje obrnemo: namesto da iščemo takšno a_k, od katere je čim več drugih spremenljivk različnih, iščimo takšno a_k, ki ji je čim manj drugih spremeljivk enakih. Definirajmo torej komplement množice r, to je $\hat{r}(k, \mathbf{a}) = \{i \in 1..n : a_i = a_k\}$, z njeno pomočjo še $\hat{s}'(k) := \max_{\mathbf{a}\in A} |\hat{r}(k, \mathbf{a})|$. Očitno velja $|\hat{r}(k, \mathbf{a})| = n - |r(k, \mathbf{a})|$, iz tega pa se hitro vidi tudi $\hat{s}'(k) = n - s'(k)$; torej, če bomo znali računati funkcijo $\hat{s}'$, bomo znali računati tudi s' in bomo tako rešili našo nalogo.

Izberimo si nek $k \in 1..n$ in $\mathbf{a} \in A$ ter v mislih označimo na grafu G točke iz množice $\hat{r}(k, \mathbf{a})$. Kaj je značilno zanje? Zgoraj smo že videli, da če obstaja v G pot od i do j, sta spremenljivki a_i in a_j različni pri vsaki dopustni rešitvi. V množici $\hat{r}(k, \mathbf{a})$ pa imamo spremenljivke, ki imajo v $\mathbf{a}$ vse enako vrednost (vse so enake a_k, zato so tudi enake druga drugi). Torej za vsaki dve točki $i, j \in \hat{r}(k, \mathbf{a})$ sledi, da v G ne obstaja niti pot od i do j niti pot od j do i.

Na misel nam lahko torej pride, da bi poskušali največjo $\hat{r}(k, \mathbf{a})$ (torej največjo pri danem k, po vseh možnih $\mathbf{a} \in A$) poiskati tako, da bi poskusili na grafu G označiti čim več točk (med njimi tudi točko k) ob omejitvi, da nobeni dve označeni točki ne smeta biti povezani s potjo (ob upoštevanju smeri povezav). Tu pa se pojavi pomislek: ali se lahko zgodi, da mi sicer res označimo neko tako množico točk (recimo

ji R), mogoče celo poiščemo največjo táko sploh, potem pa se izkaže, da ne obstaja nobena taka $\mathbf{a} \in A$, za katero je $\hat{r}(k, \mathbf{a}) = R$? To bi bilo neugodno, saj bi pomenilo, da če poiščemo največjo takšno R, to ne bo rešilo našega problema, saj utegne biti večja od največje $\hat{r}(k, \mathbf{a})$ (po vseh $\mathbf{a} \in A$).

Prepričajmo se o tem, da do te težave ne more priti. Izberimo si poljubno $k \in V$ in poljubno tako množico $R \subseteq V$, ki vsebuje k in pri kateri za vsaki $i, j \in R$ velja, da v G ne obstaja niti pot od i do j niti od j do i. Radi bi se torej prepričali, da obstaja takšna dopustna rešitev $\mathbf{a} \in A$, za katero je $R = \hat{r}(k, \mathbf{a})$. Če je G nepovezan, lahko naslednji razmislek ponovimo za vsako od njegovih šibko povezanih komponent, zato v nadaljevanju predpostavimo, da je G šibko povezan.

Naj bo P množica vseh tistih $u \in V$, za katere obstaja pot od u do vsaj ene točke iz R. Vemo, da nobena točka iz P ne leži tudi v R (saj bi v tem primeru obstajala pot med dvema točkama iz R). Množico vseh preostalih točk označimo s S, torej $S := V - P - R$. Naj bosta G_P oz. G_S podgrafa G-ja, ki ju inducirata množici P oz. S. Ker je G acikličen, sta G_P in G_S tudi in ju lahko topološko uredimo. Vzemimo poljuben topološki vrstni red točk grafa G_P in za vsako $u \in P$ naj bo t_u položaj te točke v tem vrstnem redu (to je torej število od 1 do $|P|$); podobno vzemimo poljuben topološki vrstni red točk grafa G_S in za vsako $u \in S$ naj bo t_u položaj te točke v tem vrstnem redu (to je torej število od 1 do $|S|$). Definirajmo zdaj n-terico $\mathbf{a}$ takole: $a_u := t_u - |P| - 1$ za $u \in P$; $a_u = 0$ za $u \in R$; in $a_u := t_u$ za $u \in S$.

Tako smo dobili nek $\mathbf{a}$, nabor vrednosti vseh spremenljivk $a_1, \ldots, a_n$; zanj očitno drži, da je $\hat{r}(k, \mathbf{a}) = R$, saj smo množico točk $V = 1..n$ razbili na tri disjunktne dele: P, R in S; spremenljivke iz P so dobile negativne vrednosti, tiste iz S pozitivne vrednosti, tiste iz R (med njimi je tudi a_k) pa vrednost 0. Prepričati se moramo le še o tem, da je $\mathbf{a} \in A$, torej da dobljeni nabor vrednosti res upošteva vse omejitve. No, pa vzemimo poljubno omejitev $a_u < a_v$ (obstoj te omejitve med drugim pomeni, da G vsebuje povezavo $u \to v$) iz naše naloge in ločimo nekaj možnosti:

(1) Če je $v \in P$, to pomeni, da je v prednica neke $w \in R$. Zaradi povezave $u \to v$ je torej tudi u prednica tiste $w \in R$, torej je $u \in P$. Ker sta u in v obe v P, sta obe prisotni tudi v grafu G_P in je zato v njem tudi povezava $u \to v$. To pa pomeni, da je u v vsakem topološkem vrstnem redu grafa G_P pred v; torej imamo $t_u < t_v$, torej $a_u = t_u - |P| - 1 < t_v - |P| - 1 = a_v$, torej je omejitev $a_u < a_v$ izpolnjena.

(2) Če je $v \in R$, to pomeni, da je $u \in P$ (zaradi obstoja povezave $u \to v$ je u prednica točke v, ta pa je v R). Ker je $v \in R$, je $a_v = 0$, in ker je $u \in P$, je $a_u < 0$, torej je omejitev $a_u < a_v$ izpolnjena.

(3) Če je $v \in S$, ločimo nekaj možnosti glede na to, kje je u. Ker je $v \in S$, je $a_v > 0$; če je $u \in P$, bo $a_u < 0$, in če je $u \in R$, bo $a_u = 0$, tako da bo omejitev $a_u < a_v$ vsekakor izpolnjena. Ostane še možnost, da je $u \in S$. Ker sta tako u kot v prisotni v S, sta obe prisotni tudi v grafu G_S, skupaj s povezavo $u \to v$ med njima. Zaradi te povezave vemo, da v vsakem topološkem vrstnem redu grafa G_S točka u nastopi pred točko v, torej bo dobila a_u manjšo vrednost kot a_v, tako da je omejitev $a_u < a_v$ tudi v tem primeru izpolnjena.

Tako torej vidimo, da za vsako $R \subseteq V$, ki ima zgoraj opisane lastnosti (vsebuje k in za noben par $i, j \in R$ v G ni poti od i do j ali obratno), res obstaja neka dopustna rešitev $\mathbf{a} \in A$, tako da je $R = \hat{r}(k, \mathbf{a})$. Če torej poiščemo največjo tako R, bo njena velikost enaka $\hat{s}'(k)$.

Definirajmo nov usmerjen graf G', ki je tranzitivna ovojnica prvotnega grafa G. Povezava $u \to v$ je torej v G' prisotna natanko tedaj, ko obstaja v G pot od u do v (lahko je dolga 1 ali več korakov, spoštovati pa mora seveda smeri povezav). Množici $R \subseteq V$ pravimo, da je *neodvisna* (v grafu G'), če nobena povezava grafa G' nima obeh krajišč v R. Zgoraj smo torej videli, da je $\hat{s}'(k)$ ravno enaka velikosti največje take neodvisne množice v grafu G', ki vsebuje tudi točko k; lahko pa jo dobimo tudi tako, da iz G' pobrišemo k in vse njene sosede, v tako dobljenem podgrafu poiščemo velikost največje neodvisne množice sploh in ji prištejemo 1.

Tako smo naš problem prevedli na dobro znani problem največje neodvisne množice v grafu. Ta je v splošnem sicer NP-težak, izkaže pa se, da je za nekatere družine grafov čisto obvladljiv; med drugim to drži tudi za grafe, ki so nastali kot tranzitivna ovojnica nekega acikličnega grafa (tako kot v našem primeru, ko smo G' dobili kot tranzitivno ovojnico grafa G).[15] Znano je (Dilworthov izrek), da je velikost največje neodvisne množice pri takem grafu enaka najmanjšemu številu disjunktnih poti, ki jih potrebujemo, da pokrijemo celoten graf G' (torej da leži vsaka točka grafa na natanko eni poti); to pa je naprej enako najmanjšemu številu poti, ne nujno disjunktnih, ki jih potrebujemo, da pokrijemo celoten G (zdaj torej zahtevamo, da leži vsaka točka na vsaj eni poti, ne nujno na natanko eni poti).

Pokazati je mogoče tudi, da lahko to število poti poiščemo tako, da rešimo problem pretoka po grafu G: vpeljimo dve novi točki, izvor s in ponor t, za vsako $u \in V$ dodajmo še dve povezavi $s \to u \to t$, vse povezave (stare in nove) naj imajo neomejeno kapaciteto, omejitev pa je ta, da mora biti pretok skozi vsako točko $u \in V$ vsaj 1 (če se hočemo točki k in njenim sosedam izogniti, pa zanje predpišimo, da mora biti pretok skoznje natanko 0). V okviru teh omejitev poiščimo minimalni pretok od s to t in njegova vrednost je ravno enaka številu poti, ki nas zanima.

Naloge so sestavili: prepogibanje traku, tovor — Nino Bašić; disk — Andrej Brodnik; latovščina — Luka Bradeško; avtobusi — Primož Gabrijelčič; podnizi — Mitja Lasič; lačni mravljinčar, mehurčki v slamici, kitka — Mitja Trampuš; železnica, peščene piramide — Miha Vuk; poravnavanje desnega roba, tajna pošta — Klemen Žagar; enkratno prirejanje — Janez Brank.

[15] V preostanku te rešitve bomo le na kratko skicirali, kako se lahko lotimo iskanja največje neodvisne množice v tranzitivnem grafu, za več podrobnosti pa si je koristno pomagati z literaturo: D. Kagaris, S. Tragoudas: *Maximum independent sets on transitive graphs and their applications in testing and CAD*, ICCAD 1997, pp. 736–40; M. C. Golumbic: *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980; in članek "Dilworth's theorem" v Wikipediji. Mimogrede, Dilworthov izrek bi prišel prav tudi pri lanski nalogi s poštarjem (2010.3.5): v lanskem biltenu smo na str. 74–5 pokazali povezavo med problemom najmanjšega števila poštarjevih in problemom nadaljšega padajočega zaporedja; to povezavo bi se dalo krajše in lažje utemeljiti kot posledico omenjenega izreka.

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. Primerjanje oklepajev

- V naših rešitvah je naloga rešena tako, da se sprehaja po nizih `a` in `b`, preskakuje ne-oklepajske znake in primerja oklepaje. Za enako dobro naj šteje tudi rešitev, ki bi iz nizov najprej eksplicitno pobrisala ne-oklepajske nize in ju šele potem primerjala; tudi je vseeno, če si v ta name ustvari pomožno kopijo obeh nizov ali pa dela kar s tistima, ki sta podana kot parametra.
- Če bi rešitev zaradi kakšne hude neučinkovitnosti za primerjanje dveh nizov dolžine $O(n)$ porabila več kot $O(n)$ časa, na primer $O(n^2)$ ali kaj podobnega, naj dobi največ 10 točk (če drugače daje pravilne rezultate).

2. Globalno segrevanje

- Vseeno je, ali rešitev predpostavi, da so podatki v tabeli (*array*), v povezanem seznamu (*linked list*) ali pa celo v datoteki ali čem podobnem.
- Vseeno je, ali rešitev šteje mesto za potopljeno šele, ko gladina morja postane večja od njegove višine, ali pa že takrat, ko ji postane šele enaka.
- Če gre rešitev pri vsakem letu po celotnem seznamu vseh krajev, naj dobi kvečjemu 10 točk. Z drugimi besedami, radi bi rešitev, ki za m krajev in n let porabi le $O(n + m)$ časa, ne pa $O(n \cdot m)$ časa.
- Če rešitev naredi kakšne predpostavke o največji možni dolžini vhodnih zaporedij, naj se ji odbije največ 5 točk.

3. Riziko

- Če v primerih, ko je $a_1 = b_1$ (ali pa $a_2 = b_2$) rešitev pripiše točko prvemu igralcu (namesto drugemu, kot izhaja iz besedila naloge), naj se ji odbijejo 3 točke.
- Vseeno je, ali rešitev možne kombinacije petih kock našteva z gnezdenimi zankami, z rekurzijo ali še kako drugače.
- Za urejanje kock posameznega tekmovalca lahko uporabi rešitev morebitne podprograme za urejanje iz standardne knjižnice svojega programskega jezika (četudi je to za samo dve ali tri števila neučinkovito).
- Možna napaka pri tej nalogi je, da rešitev pregleduje le takšne kombinacije kock, ki so že urejene padajoče (**for** a1 := 1 **to** 6 **do for** a2 := 1 **to** a1 **do for** a3 := 1 to a2 ...). Če rešitev deluje na ta način, mora pri štetju pravilno upoštevati, da lahko vsaka taka kombinacija nastane z urejanjem več različnih prvotnih kombinacij (največ šestih). Če tega ne dela pravilno, naj dobi največ 10 točk.

4. Preglednice

- Naloga posebej pravi, naj kličemo funkcijo Celica, čim preberemo celotno vsebino celice. Če rešitev kliče funkcijo Celica šele na koncu vrstice ali celo na koncu preglednice, med tem pa si podatke za vse te celice kopiči v pomnilniku, naj se ji odbije 5 točk. Če ob tem naredi še kakšne dodatne predpostavke o največjem številu celic v vrstici/preglednici ali pa o skupni dolžini njihove vsebine, naj se ji odbije še 5 točk.

- Če rešitev pomotoma šteje vrstice in stolpce od 0 namesto od 1 naprej, naj se ji zaradi tega odbije največ 2 točki.

5. Smučarji

- Pri tej nalogi je koristno pregledovati smuči in otroke padajoče po teži. Vseeno je, ali se rešitev ukvarja s tem, kako vhodne podatke urediti, ali pa predpostavi, da so že urejeni; tudi je vseeno, če uporabi za urejanje kakšnega od algoritmov z zahtevnostjo $O(n^2)$ namesto $O(n \log n)$.

- Rešitve z eksponentno časovno zahtevnostjo (npr. take, ki z rekurzijo pregledajo vse možne razporede otrok na smuči) naj dobijo največ 10 točk (če drugače dajejo pravilne rezultate).

- Če rešitev temelji na požrešnem algoritmu (tako kot naša rešitev zgoraj), je vseeno, če je ta implementiran v času $O(n \cdot m)$ namesto $O(n + m)$ (če ne štejemo časa urejanja); tudi ni nujno, da je postopek opisan tako podrobno, da se to sploh vidi iz rešitve (tako kot se tudi ne vidi iz naše rešitve zgoraj).

- Pomembno je tudi, ali se iz odgovora vidi, da ima tekmovalec nekakšen argument za pravilnost svoje rešitve (in je ni zapisal le zato, ker se mu po občutku zdi, da je najbrž pravilna). (To seveda ne pomeni, da pričakujemo formalen dokaz z indukcijo (in mogoče še s protislovjem), kakršne se ponavadi uporablja pri požrešnih algoritmih.)

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
1. Primerjanje oklepajev	srednje težka naloga v prvi skupini
2. Globalno segrevanje	težja v prvi ali lažja naloga v drugi skupini
3. Riziko	lažja ali srednje težka naloga v drugi skupini
4. Preglednice	srednja naloga v drugi skupini
5. Smučarji	težja naloga v drugi ali lažja v tretji 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 215), tako da smo jih letos podelili osem. 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. Letos smo v drugi skupini podelili nekaj srebrnih tekmovanj preveč, ker smo v seznam rezultatov sprva uvrstili tudi osem tekmovalcev z doseženimi 0 točkami, za katere se je kasneje izkazalo, da se tekmovanja niso udeležili. 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	Primož Godec	4	Škof. klas. gimn. Lj.	18	20	20	12	16	86
1Z	2	Žan Kusterle	2	Vegova Ljubljana	14	20	20	12	18	84
2S	3	Filip Peter Lebar	1	Gimnazija Vič	8	15	17	18	15	73
2S		Tadej Petreski	3	II. gimnazija Maribor	13	17	19	8	16	73
3S	5	Žiga Gradišar	1	Škof. klas. gimn. Lj.	8	19	18	7	18	70
3S	6	Sašo Marković	3	II. gimnazija Maribor	12	10	15	18	14	69
S	7	Alec Smrekar	2	Gimnazija Piran	15	18	19	5	10	67
S	8	Jakob Merljak	1	Gimnazija Bežigrad	0	20	20	15	11	66
S		Ines Meršak	2	Gimnazija Vič	12	15	20	14	5	66
S	10	Gregor Miklošič	3	II. gimnazija Maribor	0	19	19	13	13	64
S	11	Aljaž Jeromelj	1	II. gimnazija Maribor	6	15	15	12	12	60
S	12	Aljaž Frančič	3	II. gimnazija Maribor	15	20	15	7	1	58
S		Rok Lekše	1	Škof. kl. g. Lj. + ZRI	8	20	15	12	3	58
S	14	Klemen Plazar	3	ERŠ Velenje	7	15	20	7	8	57
S	15	Juš Debelak	3	TŠC Kranj	10	14	20	12	0	56
S		Gregor Mohorko	3	ŠCC Gimnazija Lava	0	13	18	18	7	56
S	17	Alenka Bahovec	3	Škof. klas. gimn. Lj.	15	14	20	5	0	54
S		Žiga Šmelcer	1	Škof. klas. gimn. Lj.	10	14	20	10	0	54
S	19	Denis Jazbec	4	SŠ za KER	13	16	20	4	0	53
S	20	Jan Perme	1	Gimnazija Vič	8	18	12	10	4	52
S	21	Rok Bevc	3	Vegova Ljubljana	3	20	12	6	8	49
S	22	Toni Kocjan Turk	2	ŠC Novo mesto	4	15	15	5	7	46
S		Filip Koprivec	1	Gimnazija Vič	1	20	20	0	5	46
S	24	Igor Đukanovič	2	ERŠ Velenje	18	15	1	3	6	43
S	25	Žiga Kokelj	1	Gimnazija Škofja Loka	15	3	4	6	14	42
S		Gregor Menih	2	ERŠ Velenje	0	17	17	8	0	42
S		Tadej Štadler	3	ERŠ Velenje	0	18	19	4	1	42
S	28	Marko Ljubotina	4	SŠJJ Ivančna Gorica	14	15	0	4	7	40
S	29	Matija Andrejčič	4	ŠC Novo mesto	0	5	12	15	7	39
S		Gorazd Kunej	4	SERŠ Maribor	1	19	12	4	3	39
S		Aleksander Rajhard	1	Gimnazija Škofja Loka	1	3	19	8	8	39
S		Gaj Žižek	2	Gimnazija Šentvid	8	11	0	12	8	39
S	33	Beno Šircelj	3	STŠ Koper	12	14	8	0	4	38
S	34	Žan Knafelc	9	ZRI	0	20	5	5	6	36
S		Luka Prebil	1	Gimnazija Vič	8	13	5	9	1	36
S	36	Renato Korez	4	SERŠ Maribor	0	20	0	12	2	34
S	37	Aljaž Borštnik	2	Gimnazija Vič	4	9	8	5	7	33

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
	38	Jan Golob	1	Gimnazija Vič	1	15	3	7	6	32
		Miha Oražem	3	Vegova Ljubljana	0	15	10	7	0	32
	40	Urban Lavbič	3	SŠ za KER	0	15	8	3	5	31
		Luka Toni	3	TŠC Kranj	6	19	2	4	0	31
		Leon Stanko	3	ŠCC Gimnazija Lava	15	1	13	0	2	31
	43	Marko Grešak	2	ŠC Novo mesto	0	18	0	8	4	30
	44	Aleš Jud	2	SPITŠ Murska Sobota	1	13	5	4	5	28
	45	Marion Antonia van Midden	4	SŠJJ Ivančna Gorica	0	10	10	3	4	27
	46	Mojca Komavec	4	Gimnazija Šentvid	6	1	12	3	4	26
	47	Rok Janež	2	ŠC Novo mesto	0	12	2	7	3	24
		Dominik Sočič	2	SPITŠ Murska Sobota	1	15	3	3	2	24
		Medard Švigelj	1	Škof. klas. gimn. Lj.	0	10	8	0	6	24
	50	Vid Leskovar	2	II. gimnazija Maribor	3	0	15	0	3	21
		Andrej Orehek	3	SŠJJ Ivančna Gorica	5	5	8	3	0	21
	52	Mitja Zidar	4	SŠJJ Ivančna Gorica	0	0	10	0	8	18
	53	Juš Hladnik	8	OŠ Pirniče	0	10	4	3	0	17
		Žiga Jerše	4	Gimnazija Šentvid	0	10	0	5	2	17
		Jaka Šušteršič	4	Gimnazija Škofja Loka	4	5	2	6	0	17
		Blaž Velkavrh	1	Gimnazija Vič	8	2	3	4	0	17
	57	Gregor Mrak	2	TŠC NG, Teh. gimn.	0	15	1	0	0	16
		Matic Slemič	2	TŠC NG, Teh. gimn.	15	1	0	0	0	16
	59	Jana Blažič	2	ERŠ Velenje	5	5	0	4	1	15
	60	Alen Berk	2	SŠ za KER	0	2	9	3	0	14
		Janez Kuhar	3	Gimnazija Litija	3	5	0	5	1	14
		Matjaž Milinovič	4	Gimnazija Škofja Loka	0	0	0	3	11	14
	63	Tomaž Mikola	4	SŠ za KER	7	3	0	0	3	13
	64	Martin Opršnik	2	SŠ za KER	0	12	0	0	0	12
	65	Primož Prevc	1	Škof. klas. gimn. Lj.	0	0	3	0	6	9
	66	Jaka Gubanc	1	Škof. klas. gimn. Lj.	4	0	0	0	2	6
		Simon Žekš	2	SPITŠ Murska Sobota	0	1	0	4	1	6
	68	Uroš Hudomalj	1	Gimnazija Bežigrad	0	0	0	5	0	5
	69	Jan Korenč	4	TŠC Kranj	1	2	0	0	1	4
	70	Luka Kozina	8	OŠ Pirniče	0	0	0	3	0	3
	71	Simon Weiss	1	ZRI	0	1	0	0	1	2
	72	Luka Domitrovič	2	ERŠ Velenje	0	0	0	0	0	0
		Samo Kralj	1	Gimnazija Vič	0	0	0	0	0	0
		Kaja Nemanič	4	Gimnazija Šentvid	0	0	0	0	0	0

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Andraž Dobnikar	3	Vegova Ljubljana	19	18	18	1	20	76
1Z	2	Žiga Gosar	3	Gimnazija Vič	16	8	20	15	11	70
2Z	3	Patrik Zajec	1	ZRI Ljubljana	18	17	15	6	12	68
2S	4	Tadej Ciglarič	2	Gimnazija Bežigrad	19	12	7	15	13	66
2S	5	Vid Kocijan	1	ZRI Ljubljana	20	10	12	15	8	65
3S	6	Aleš Razpotnik	4	Vegova Ljubljana	8	14	10	18	13	63
3S	7	Živa Urbaničič	1	ZRI Ljubljana	18	14	0	14	14	60
S	8	Nejc Grenc	4	Škof. klas. gimn. Lj.	19	16	10	13	0	58
S		Žiga Zalokar	3	Vegova Ljubljana	16	14	2	14	12	58
S	10	Jan Aleksandrov	1	ZRI Ljubljana	18	10	0	15	11	54
S	11	Matej Biberovič	4	Srednja šola Ravne	13	9	4	15	12	53
S		Sašo Stanovnik	3	Gimnazija Bežigrad	14	14	0	15	10	53
S		Matevž Žugelj	3	Vegova Ljubljana	17	9	2	15	10	53
S	14	Gašper Primožič	4	Srednja šola Ravne	20	10	0	15	7	52
S	15	Erik Langerholc	2	Gimnazija Bežigrad	13	13	0	13	12	51
S	16	Jure Kolenko	4	Vegova Ljubljana	10	5	7	13	15	50
S		Aleksander Novaković	3	Gimnazija Bežigrad	13	10	2	12	13	50
S	18	Tomi Raguž	4	STŠ Koper	1	14	4	15	13	47
S	19	Matic Bernik	3	Vegova Ljubljana	1	14	10	13	8	46
S	20	Dejan Paradiž	3	Srednja šola Ravne	5	7	5	14	13	44
S	21	Marko Novak	2	ZRI Ljubljana	15	0	8	8	12	43
S	22	Denis Celcer	4	SERŠ Maribor	5	12	1	11	10	39
S	23	Mihael Polanec	4	SERŠ Maribor	4	14	4	4	12	38
	24	Tadej Vengust	4	SERŠ Maribor	8	9	5	0	12	34
	25	Etien Rožnik	4	STŠ Koper	15	14	0	0	0	29
		Denis Tepeš	2	SŠ za KER	8	6	2	13	0	29
	27	Marko Jelenko	4	SŠ za KER	14	3	2	0	8	27
		Tadej Pregl	4	SERŠ Maribor	1	1	0	14	11	27
	29	Miran Helbl	4	Srednja šola Ravne	1	3	0	10	12	26
	30	Luka Hrvatin	4	STŠ Koper	14	10	0	0	0	24
		Klemen Kozjek	3	Vegova Ljubljana	0	0	10	3	11	24
		Jan Markočič	4	TŠC NG, Teh. gimn.	4	9	0	0	11	24
	33	Jernej Izak	4	Srednja šola Ravne	5	0	4	2	11	22
	34	Grega Močnik	4	Srednja šola Ravne	4	0	0	4	9	17
	35	Marko Očko	4	SŠ za KER	4	12	0	0	0	16
	36	Jure Domajnko	3	SPITŠ Murska Sobota	0	0	0	0	15	15
	37	Michel Adamič	2	Gimnazija Bežigrad	4	8	0	0	0	12
	38	Žiga Pušnik	2	SŠ za KER	1	5	2	0	0	8
	39	Denis Smej	3	SPITŠ Murska Sobota	0	0	0	0	0	0

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Matjaž Leonardis	3	ZRI	100	100	100	97	100	497
2Z	2	Klemen Kloboves	4	ZRI	77		27	91	100	295
2Z	3	Andraž Bajt	4	TŠC NG, Teh. gimn.	62	20		100	20	202
3S	4	Gašper Medved	3	Vegova Ljubljana	90	17		35		142
S	5	Jure Slak	3	Gimnazija Vič	37	40		0	20	97
S	6	Maks Kolman	3	Gimnazija Vič	37			57		94
S	7	Tibor Djurica	3	Gimnazija Poljane	24	0	27	0	40	91
S	8	Jasna Urbančič	3	ZRI	85			0		85
	9	Sandi Majninger	4	II. gimnazija Maribor	62	10		0	0	72
	10	Urban Škvorc	3	Gimnazija Bežigrad	47	10				57
	11	Aljoša Mrak	4	TŠC NG, Teh. gimn.	52					52
	12	Sven Cerk	3	ZRI	45					45
	13	Ernest Beličič	4	Vegova Ljubljana	20	10				30
	14	Tadej Škvorc	3	Gimnazija Bežigrad	10	10				20
	15	Luka Zakrajšek	4	Vegova Ljubljana	17				0	17
	16	Marko Bavdek	3	TŠC NG, Teh. gimn.	2	1			10	13
	17	Matej Vehar	4	ŠC Novo mesto	4	0				4

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Primož Godec	16 GB iPod nano
1	1	Žan Kunsterle	64 GB SSD disk
1	2	Tadej Petreski	2 TB zunanji disk R. Jamnik: *Teorija iger*
1	2	Filip Peter Lebar	64 GB USB flash disk K. Devlin: *Nova zlata doba matematike*
1	3	Žiga Gradišar	1 TB zunanji disk S. Hawking: *Kratka zgodovina časa*
1	3	Sašo Marković	1 TB zunanji disk S. Hawking: *Kratka zgodovina časa*
2	1	Andraž Dobnikar	3 TB zunanji disk Dasgupta *et al.: Algorithms* K. Devlin: *Nova zlata doba matematike*
2	1	Žiga Gosar	3 TB zunanji disk Dasgupta *et al.: Algorithms* K. Devlin: *Nova zlata doba matematike*
2	2	Patrik Zajec	2 TB zunanji disk Dasgupta *et al.: Algorithms* J. Rosenthal: *Ko strela udari: skrivnostni svet verjetnosti*
2	2	Tadej Ciglarič	2 TB zunanji disk I. Vidav: *Števila in matematične teorije*
2	2	Vid Kocijan	64 GB USB flash disk K. Devlin: *Nova zlata doba matematike*
2	3	Aleš Razpotnik	1,5 TB zunanji disk S. Hawking: *Kratka zgodovina časa*
3	1	Matjaž Leonardis	APC SMART-UPS 750 VA Cormen *et al.: Introduction to algorithms* R. Jamnik: *Teorija iger* Wilson, Watkins: *Uvod v teorijo grafov* J. Rosenthal: *Ko strela udari: skrivnostni svet verjetnosti*
3	2	Klemen Kloboves	3 TB zunanji disk Cormen *et al.: Introduction to algorithms* R. Jamnik: *Teorija iger* Wilson, Watkins: *Uvod v teorijo grafov* J. Rosenthal: *Ko strela udari: skrivnostni svet verjetnosti*
3	2	Andraž Bajt	64 GB SSD disk Cormen *et al.: Introduction to algorithms* Wilson, Watkins: *Uvod v teorijo grafov*
3	3	Gašper Medved	3 TB zunanji disk L. Luchetti: *Strast do trilčkov*
Tekmovanje programov — Minolovec			
— Dijaki:			
	3	Luka Horvat	1.5 TB zunanji disk
	4	Maks Kolman	miška Logitech G700
— Študentje:			
	1	Gregor Čepin	1.5 TB zunanji disk
	2	Rok Kralj	miška Logitech G700

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	
Gimnazija Bežigrad	Andrej Šuštaršič
Gimnazija Litija	Jan Maver
Gimnazija Piran	Špela Rožman, Janez Urevc
Gimnazija Poljane	Janez Malovrh
Gimnazija Šentvid	Klemen Blokar, Nastja Lasič
Gimnazija Škofja Loka	
Gimnazija Vič	Klemen Bajec
OŠ Pirniče	Urška Wertl
Srednja elektro-računalniška šola Maribor (SERŠ)	Slavko Nekrep
Srednja poklicna in tehniška šola Murska Sobota (SPITŠ)	Simon Horvat, Karel Maček
Srednja šola Josipa Jurčiča, Ivančna Gorica	Darko Pandur
Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Dušan Fugina
Srednja tehniška šola (STŠ) Koper	Andrej Florjančič, Senka Sabotin
Škofijska klasična gimnazija Šentvid	Nace Hudobivnik, Helena Medvešek
Šolski center Celje, Gimnazija Lava	Karmen Kotnik, Tomislav Viher
Šolski center Novo mesto	Mile Božič, Tomaž Ferbežar, 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
Tehniški šolski center Kranj	Gašper Strniša
Tehniški šolski center Nova Gorica, Tehniška gimnazija	Barbara Pušnar
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Gregor Hrastnik, Miran Zevnik
Vegova Ljubljana	Matjaž Kodela, Nataša Makarovič, Dušan Sitar, Darjan Toth
Zavod za računalniško izobraževanje (ZRI), Ljubljana	Jelko Urbančič

TEKMOVANJE PROGRAMOV — MINOLOVEC

Podobno kot v prejšnjih letih smo tudi letos organizirali tekmovanje programov. Opis naloge smo objavili avgusta 2010 skupaj z razpisom za tekmovanje v znanju, tekmovalci pa so imeli čas do 12. marca 2011 (dva tedna pred tekmovanjem), da pošljejo svoje programe. Letošnja naloga je od tekmovalcev zahtevala, da napišejo logiko za igranje minolovca (minesweeper) z rahlo prilagojenimi pravili.

Opis naloge

Dana je karirasta mreža velikosti $w \times h$. V nekatere celice so postavljene mine (število miniranih celic, m, tudi izvemo na začetku igre), vendar so na začetku igre vse celice zakrite in ne vemo, v katerih so mine. Vsaka poteza je sestavljena iz tega, da si igralec izbere neko zakrito celico in jo odkrije. Od sistema dobi podatek o tem, ali je v tej celici mina ali ne, poleg tega pa izve tudi število min v sosednjih osmih celicah (ne glede na to, ali so te sosednje celice zakrite ali odkrite).

V originalni različici minolovca je cilj igre v tem, da odkrijemo vse neminirane celice, ne da bi pri tem odkrili kakšno minirano celico. Ker pa se je odkrivanju miniranih celic pogosto težko izogniti (ker se iz tega, kar trenutno vidimo na mreži, včasih ne da za nobeno celico povsem zanesljivo sklepati, da v njej ni mine, tako da nam ne ostane drugega, kot da začnemo ugibati), smo za potrebe našega tekmovanja to pravilo spremenili. V naši različici minolovca se igra konča šele, ko so odkrite vse celice, na katerih ni min, izziv za igralca pa je v tem, da odkrije vse neminirane celice in pri tem odkrije čim manj miniranih celic.

Da bo naloga lažja, smo tudi zagotovili, da bodo pri vseh testnih primerih celice v vogalih mreže neminirane.

Rezultati

Za nalogo je bilo precej zanimanja in na koncu smo prejeli rešitve 14 tekmovalcev (8 dijakov in 6 študentov). Prejete programe smo preizkusili na 11500 minskih poljih, jih pri vsakem polju razvrstili po številu pohojenih min in na koncu izračunali povprečno uvrstitev vsakega programa. To prikazuje spodnja tabela:

Mesto	Ime	Šola	Povp. uvrstitev
1	Gregor Čepin	FRI	1,9
2	Rok Kralj	IŠRM	2,0
3	Luka Horvat	ERŠ Ptuj	3,4
4	Maks Kolman	Gim. Vič	4,1
5	Simon Kozina	IŠRM	5,6
6	Gašper Medved	Vegova Ljubljana	6,1
7	Matija Andrejčič	ŠC Novo mesto	6,3
8	David Ogorevc	STŠ Krško	6,5
	Luka Pušić	Gim. Ledina	6,5
10	Martin Freser	FNT Maribor	7,4
11	Matjaž Leonardis	ZRI	10,8
	Matej Vehar	ŠC Novo mesto	10,8
13	Peter Žužek	IŠRM	13,0
14	Blaž Pavlica	IŠRM	⋆

⋆ Opomba: ta program se je pri večini testnih primerov sesul.

Podatki o minskih poljih, na katerih smo preizkušali dobljene programe:

Širina w	Višina h	Št. min m	Delež miniranih polj $m/(wh)$	Št. polj s temi parametri
17	13	45	20 %	1000
17	13	50	23 %	1000
17	13	55	25 %	1000
20	20	80	20 %	500
20	20	90	23 %	500
20	20	100	25 %	500
30	16	100	21 %	1000
30	16	110	23 %	1000
30	16	120	25 %	1000
30	16	130	27 %	1000
30	16	140	29 %	1000
30	30	225	25 %	1000
130	7	220	24 %	1000

Rešitev

Na začetku igre lahko odkrijemo celice v vogalih mreže, saj naloga zagotavlja, da v njih ni min. V nadaljevanju igre pa bomo morali s sklepanjem ugotavljati, katere celice so minirane in katere niso.

Včasih lahko za kakšno celico zelo preprosto ugotovimo, da jo smemo odkriti. Recimo, da imamo neko odkrito celico c in zanjo vemo, da je okoli nje m min. Preglejmo c-jeve sosede in preštejmo, koliko (recimo k) je takih, za katere že vemo, da so minirane (bodisi ker smo jih že odkrili ali pa smo s sklepanjem ugotovili, da je tam mina). Poleg tega tudi preštejmo, koliko c-jevih sosed (recimo z) je še zakritih in zanje še ne vemo, ali so minirane ali ne. Če se izkaže, da je $k = m$, potem vse mine okoli celice c že poznamo in vemo, da lahko tistih z zakritih sosed odkrijemo, saj na njih gotovo ni min. Podobno, če se izkaže, da je $k + z = m$, lahko zaključimo, da so na vseh tistih z zakritih sosedah mine, torej si nekje označimo, da so tam mine, odkrivajmo pa jih do nadaljnjega raje ne.

Če se z dosedanjim razmislekom ne da najti nobenega polja, ki bi ga lahko varno odkrili, lahko poskusimo z malo naprednejšim sklepanjem. Razdelimo v mislih celice naše mreže na tri disjunktne množice: v A naj bodo vse odkrite celice; v B naj bodo tiste zakrite celice, ki imajo kakšno odkrito sosedo; v C pa naj bodo vse preostale celice. Za vsako $b \in B$ imejmo spremenljivko $x_b \in \{0, 1\}$, ki pove število min na celici b. Poleg tega imejmo še spremenljivko y, ki pove skupno število min na celicah iz C; zanjo seveda velja $y \in \{0, \ldots, |C|\}$. Naj bo zdaj a neka taka celica iz A, ki ima kakšno zakrito sosedo; ker je $a \in A$, je a odkrita, torej poznamo število min okoli nje; recimo mu m_a. Preštejmo a-jeve odkrite sosede, na katerih so mine; recimo, da jih je k_a. Preostalih $m_a - k_a$ min mora torej ležati na a-jevih zakritih sosedah; te pa po definiciji vse pripadajo množici B. Dobili smo torej omejitev (enačbo)

$$\sum_{b \in B \,:\, b \text{ je soseda } a} x_b = m_a - k_a.$$

To ponovimo za vse take $a \in A$, ki imajo kakšno zakrito sosedo, in dobimo nek nabor omejitev. Še eno omejitev pa lahko dobimo takole: naj bo k število min na odkritih celicah (iz A) in m število min na celotni mreži (ta podatek smo dobili na začetku

igre). Na zakritih celicah je torej $m - k$ min; od tega je v območju B pač $\sum_{b \in B} x_b$ min, v območju C pa y min; tako imamo omejitev

$$y + \textstyle\sum_{b \in B} x_b = m - k.$$

Na tako dobljenem sistemu omejitev lahko uporabimo prijeme logičnega programiranja z omejitvami (CLP, *constraint logic programming*). Za vsako spremenljivko vzdržujmo zalogo možnih vrednosti (začnemo z $x_b \in \{0, 1\}$ in $y \in \{0, \ldots, |C|\}$), nato pa s pomočjo omejitev te zaloge vrednosti klestimo tako, da iz njih brišemo vrednosti, s katerimi omejitev zagotovo ni mogoče izpolniti:

za vsako omejitev O:
  za vsako spremenljivko X, ki nastopa v tej omejitvi:
    za vsako možno vrednost x iz zaloge vrednosti spremenljivke X:
      če se ostalim spremenljivkam, ki nastopajo v O, ne da izbrati takšnih vrednosti (iz njihovih zalog vrednosti), da bi bila omejitev O izpolnjena,
      potem pobriši x iz zaloge vrednosti spremenljivke X;

Tak postopek ponavljamo, dokler se zaloge vrednosti še kaj zmanjšujejo. Če se kakšni x_b zaloga vrednosti zmanjša na $\{0\}$, je to znak, da na celici b ni mine in jo lahko odkrijemo; če se ji zaloga vrednosti zmanjša na $\{1\}$, pa je to znak, da na tej celici gotovo je mina.

Če s tem razmislekom ne dobimo nobene nove poteze, gremo lahko še korak dlje. Izberimo si neko $b \in B$ in se vprašajmo: kaj bi bilo, če bi tu bila mina? V mislih torej omejimo zalogo vrednosti x_b na $\{1\}$ in poženimo zgoraj omenjeni postopek usklajevanja omejitev. Zdaj se lahko zgodi, da bo zaloga vrednosti kakšne spremenljivke sčasoma postala prazna množica; to je znak, da je naš nabor omejitev zdaj nekonsistenten (ni takega nabora vrednosti spremenljivk, pri katerem bi bile vse omejitve izpolnjene), torej je bila naša predpostavka, da je na b mina, neupravičena (saj je vodila v protislovje); torej vemo, da na b ni mine in lahko to celico odkrijemo.

Če tudi takšno razmišljanje ne pripelje do novih potez, bi lahko načeloma sicer nadaljevali rekurzivno in naredili takšne predpostavke za vse več b-jev hkrati; prej ali slej bi lahko kar našteli vse nabore vrednosti spremenljivk, ki so konsistentni z našimi omejitvami, vendar bi to prav lahko trajalo neobvladljivo veliko časa (eksponentno veliko v odvisnosti od velikosti minskega polja). Zato se je v tem primeru bolj koristno zateči k ugibanju, na primer takole: če za neko celico c vemo, da je okoli nje z zakritih sosed (takih, za katerih še ne vemo, ali so na njih mine) in m min, od katerih jih k že poznamo, si lahko mislimo, da je na vsaki od tistih zakritih sosed (recimo c') verjetnost mine približno $(m - k)/z$.[16] Če je neka zakrita c' soseda več različnim c-jem, bomo pri vsakem mogoče dobili drugačno verjetnost tega, da je na c' mina; v tem primeru bodimo previdni in med temi verjetnostmi upoštevajmo največjo. Ko tako pregledamo celotno minsko polje in za vsako c' dobimo neko verjetnost, odkrijmo tisto c', pri kateri je verjetnost mine najmanjša.

[16] To je v resnici seveda le naša poceni pridobljena ocena prave verjetnosti mine. Pravo verjetnost bi dobili takole: če bi našteli vse razporede min, ki so konsistentni s tem, kar trenutno vemo o minskem polju, bi morali pogledati, pri kolikšnem deležu teh razporedov je v celici c' mina. Žal ta pristop k računanju verjetnosti v praksi ni uporaben, saj je takšnih razporedov lahko eksponentno veliko.

To rešitev bi se najbrž dalo še kako izboljšati. Na primer, zanimivo vprašanje je, ali je v kakšnem primeru pametno odkriti celico, za katero že vemo, da je na njej mina; s tem bomo sicer pohodili mino (in si tako malo poslabšali rezultat), vendar bomo po drugi strani tudi izvedeli, koliko je min na sosednjih poljih, to pa je potencialno koristen nov podatek, ki bi mogoče prišel prav pri nadaljnjem sklepanju.

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 od 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 Pragi), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bo tokrat v Wrocławu na Poljskem).

Na letošnjem UPM je sodelovalo 33 ekip s skupno 89 tekmovalci, ki so prišli z vseh treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednji strani prikazuje ekipe, ki so rešile vsaj eno nalogo.

	Ekipa	Št. rešenih nalog*	Čas
1	Tomaž Hočevar (FRI), Matjaž Leonardis (Gim. Bežigrad), Jan Berčič (FMF)	28	35:25:06
2	Žiga Ham (FRI), Nace Hudobivnik (FMF), Matej Aleksandrov (FMF)	16	19:46:20
3	Matic Potočnik, Anže Pečar, Miha Zidar (FRI)	10	16:18:16
4	Jurij Volčič, Jakob Vidmar, Natan Žabkar (FMF)	10	16:46:04
5	Peter Koželj (FMF), Klemen Kloboves (Gim. Šk. Loka / IŠRM)	10	21:19:47
6	Jure Slak, Žiga Gosar, Maks Kolman (Gimnazija Vič)	8	13:07:09
7	Tim Kos, Matej Kren, Aleksander Kelenc (FNM Maribor)	7	10:55:08
8	Žiga Zalokar, Andraž Dobnikar, Marko Novak (Vegova Lj.)	7	14:00:25
9	Jernej Strasner, Aleš Horvat, Jani Zajc (FAMNIT)	6	10:04:42
10	Dragana Božović, Martin Duh, Martin Frešer (FNM Maribor)	6	10:15:54
11	Ernest Beličič, Aleš Razpotnik, Jure Kolenko (Vegova Lj. / FRI)	6	10:21:12
12	Žan Kafol, Tomaž Karič, Primož Karič (FRI)	6	11:39:15
13	Aleksandar Tošić, Duško Topić, Simon Mezgec (FAMNIT)	6	17:26:09
14	Aleksandar Todorović, Marko Tavčar, Tilen Božič (FAMNIT)	5	8:12:43
15	Matej Petkovič, Luka Černe, Filip Kozarski (FMF)	5	9:26:38
16	Matevž Černe, Anže Žitnik, Primož Črnigoj (FRI)	3	4:35:22
17	Matej Filipovič, Tomaž Tomažinčič, Andrej Godec (FAMNIT)	3	8:14:14

* Opomba: naloge z najslabšega od prvih treh krogov se ne štejejo, naloge z zadnjega kroga pa se štejejo dvojno. Enako je tudi pri času.

Na srednjeevropsko tekmovanje so se uvrstile ekipe 1 (s Klemnom Klobovesom namesto Matjaža Leonardisa, ker je bil slednji še srednješolec), 2 in 3 kot predstavnice Univerze v Ljubljani, 7 kot predstavnica Univerze v Mariboru in 13 kot predstavnica Univerze na Primorskem. V konkurenci 64 ekip s 30 univerz iz 6 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
13	Jan Berčič, Tomaž Hočevar, Klemen Kloboves	6	15:35:11
24	Matej Aleksandrov, Žiga Ham, Nace Hudobivnik	5	11:59:24
49	Aleksander Kelenc, Tim Kos, Matej Kren	2	2:46:38
60	Anže Pečar, Matic Potočnik, Miha Zidar	1	16:39
63	Simon Mezgec, Duško Topić, Aleksandar Tošić	1	2:33:06

Na srednjeevropskem tekmovanju je bilo 10 nalog, od tega jih je zmagovalna ekipa rešila devet.

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 125–131.

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++: **&**, **|**, **^**, **~**)	☐	☐	☐
Operatorja **shl** in **shr** (v C/C++: **<<**, **>>**)	☐	☐	☐
[če pišeš v C++] razred map iz STLa	☐	☐	☐
[če pišeš v C++] razred priority_queue iz STLa	☐	☐	☐

[Naslednja skupina vprašanj se je ponovila za vsako nalogo po enkrat.]
Zahtevnost naloge: ☐ prelahka ☐ lahka ☐ primerna ☐ težka ☐ pretežka ☐ ne vem
Naloga je (ali: bi) vzela preveč časa: ☐ da ☐ ne ☐ ne vem
Mnenje o besedilu naloge:
— dolžina besedila: ☐ prekratko ☐ primerno ☐ predolgo
— razumljivost besedila: ☐ razumljivo ☐ težko razumljivo ☐ nerazumljivo
Naloga je bila: ☐ zanimiva ☐ dolgočasna ☐ že znana ☐ povprečna
Si jo rešil(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 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 70 tekmovalcev prve skupine, 39 tekmovalcev druge skupine in 17 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. 126. 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: 3,56 v prvi skupini (v prejšnjih letih 3,34, 3,56, 3,57), 3,39 v drugi skupini (prejšnja leta 3,38, 3,46, 3,62) in 3,57 v tretji skupini (predlani 3,92; leto prej 3,74).

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, letos pa je skorajda povsem izginila ($R^2 = 0{,}11$, prejšnja leta okoli 0,4).

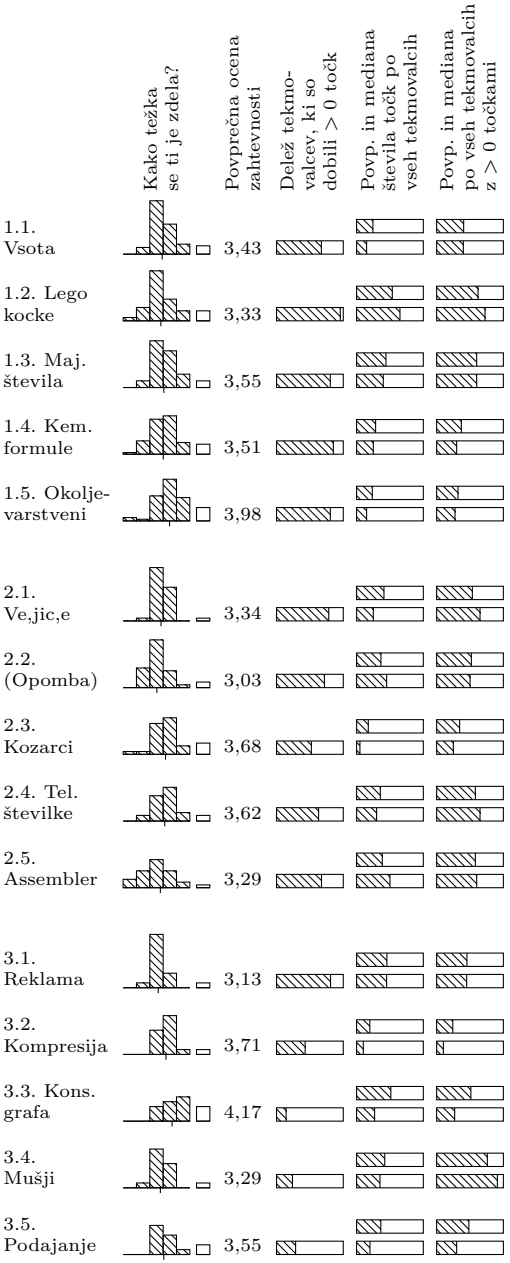
Največ pripomb o tem, kako da je naloga težka, je bilo pri nalogah 1.5 (okoljevarstveni ukrepi), 2.3 (kozarci), 3.2 (kompresija slike) in 3.3 (konstrukcija grafa). Naloga 3.2 je res malo težja, sploh če hočemo učinkovito rešitev; pri 1.5 pa je mogoče glavni problem v dolžini besedila. Pri 2.3 in 3.3 si mnenje, da sta težki, najbrž lahko razložimo s tem, da sta nalogi bolj nestandardnega tipa; je pa zanimivo, da to ne velja pri nalogi 2.5 (assembler), ki so jo tekmovalci ocenili kot eno od najlažjih.

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 127. Nad razumljivostjo besedil ni veliko pripomb (podobno kot lani in še malo manj kot prejšnja leta); najtežje razumljive so se jim zdele naloge 1.5 (okoljevarstveni ukrepi), 2.4 (telefonske številke) in 3.3 (konstrukcija grafa).

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, približno tako kot lani. Pri tem izstopa le naloga 1.5 (okoljevarstveni ukrepi), pri kateri se je besedilo veliko tekmovalcem zdelo predolgo.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot lani. Več pripomb glede dolgočasnosti nalog je bilo v tretji skupini, še posebej pri nalogah 3.2 (kompresija slike) in 3.3 (konstrukcija grafa).

Pripomb, da bi naloga vzela preveč časa, je bilo malo več kot lani, vendar podobno kot v letih pred tem, v tretji skupini pa celo še manj. Največ takih pripomb je bilo pri nalogah 1.5 (okoljevarstveni ukrepi) in 2.3 (kozarci); najbrž zato, ker je besedilo dolgo (1.5) ali pa tip naloge tekmovalcem ni tako domač (2.3), saj drugače z reševanjem teh dveh nalog ni tako zelo veliko dela.



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

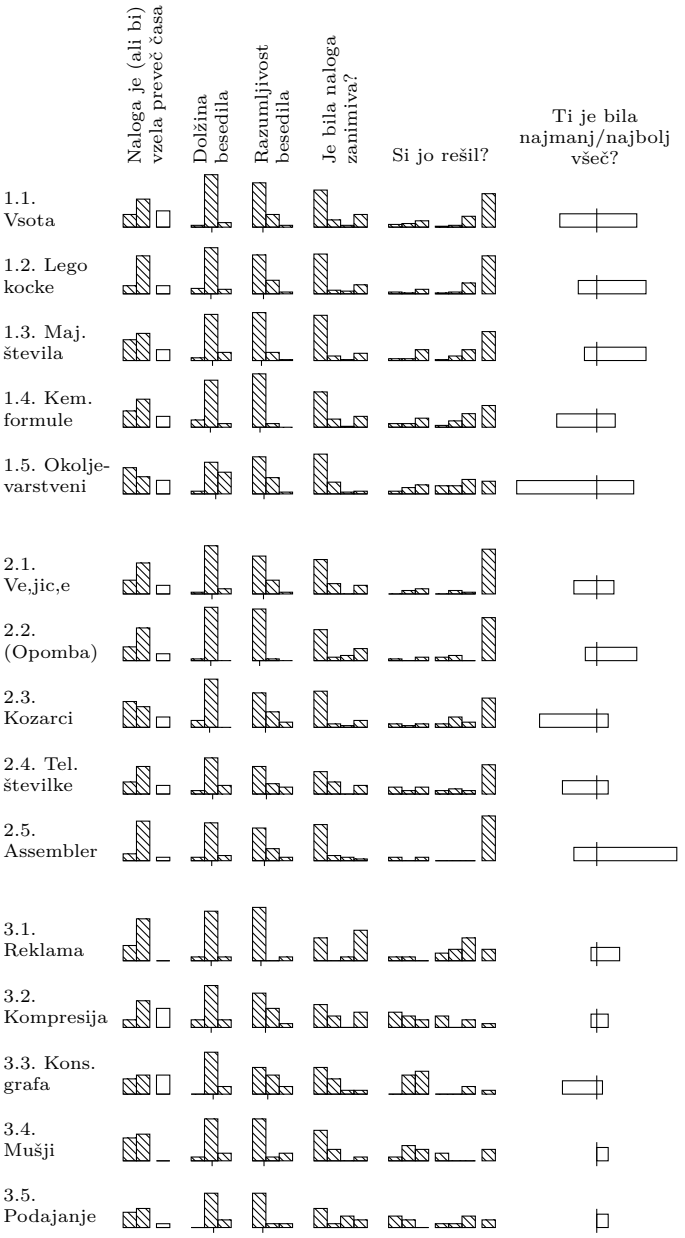
Pomen stolpcev v vsaki vrstici:

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?

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, sta bili letos v drugi skupini posebej priljubljeni nalogi 2.5 (assembler) in 2.2 (opomba), v prvi skupini pa so bili glasovi bolj razpršeni. Kot izrazito nepriljubljene izstopajo naloge 1.5 (okoljevarstveni ukrepi), 2.3 (kozarci) in 3.3 (konstrukcija grafa). Kot običajno se izkaže, da so tekmovalcem ponavadi bolj všeč lažje naloge, težje pa manj.

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.

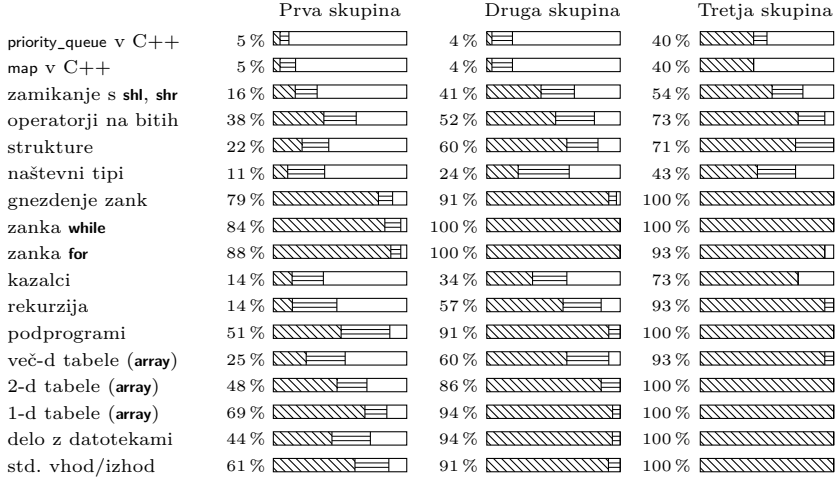


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let; v tretji skupini je pritrdilnih odgovorov še celo malo več kot v preteklosti. Tudi v izvorni kodi, ki so jo tekmovalci v tretji skupini oddajali med tekmovanjem, je videti, da jih že kar precej uporablja razne generične podatkovne strukture iz standardnih knjižnic svojih programskih jezikov (največkrat sicer generike za dinamične tabele, npr. vector, ArrayList in podobne); zato bomo vprašanji o uporabi razredov map in priority_queue v C++ prihodnje leto razširili še s podobnimi razredi v drugih programskih jezikih. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: rekurzija, kazalci, naštevni tipi in operatorji na bitih.

Uporaba programskih jezikov

Velika večina tekmovalcev tudi letos uporablja C in C++ (podobno kot lani je čisti C pravzaprav že redek), okrepila se je uporaba pythona (še posebej v prvi skupini) in C#, rahlo upadla pa je v primerjavi s prejšnjimi leti uporaba jave. Pascal se

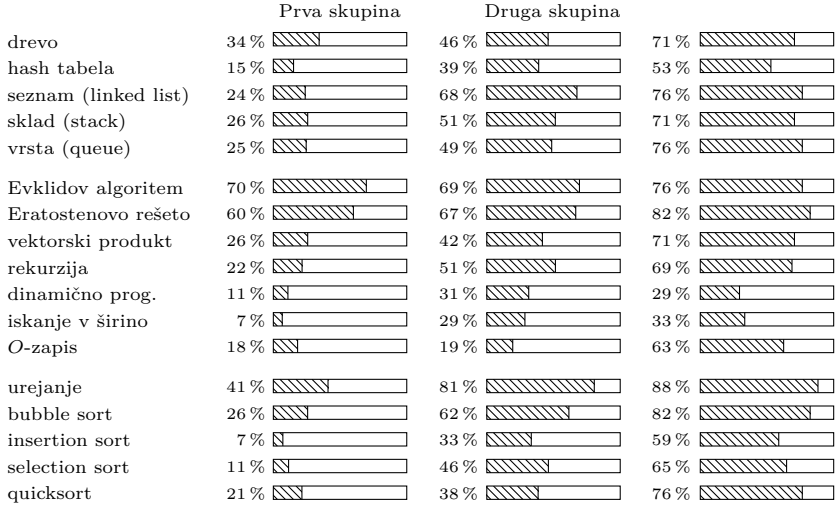


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

---

drži na približno istem nivoju kot prejšnja leta, basica pa tokrat ni uporabljal nihče. Podobno kot prejšnja leta se je tudi letos pojavilo nekaj število 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. Po eni strani to mogoče pomeni, da bi morali dati več nalog tipa „opiši postopek“ in manj „napiši podprogram“ (letos so bile v 1. skupini štiri naloge tipa „napiši podprogram“); po drugi strani pa se v praksi izkaže, da so odgovori tekmovalcev pri nalogah tipa „opiši postopek“ pogosto precej nejasni, tako da jih je težje ocenjevati.

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. 130. 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++, pythonu in javi. Zaradi komentarjev v prejšnjih letih smo tokrat v primerih, ki zahtevajo delo z nizi, ločili deklaracije v C++ od tistih v C-ju (in uporabili tip `string` namesto **`char *`**). Tekmovalce smo v anketi vprašali, če te deklaracije razumejo ali pa bi morale biti še v kakšnem drugem jeziku; veliki večini sedanje deklaracije zadostujejo (53/62 v prvi skupini in 32/34 v drugi). Najpogostejša dodatna želja so zdaj deklaracije v C#, ki jih je želelo kar precej tekmovalcev, tako da jih bomo na naslednjem tekmovanju tudi res uvedli.

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,

	Leto in skupina																	
	2011			2010			2009			2008			2007			2006		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	3	4	3	$4\frac{1}{2}$	5	2	4	2	1	$1\frac{1}{2}$	2	2	$8\frac{1}{2}$	2	1	6	5	5
C	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	$6\frac{1}{2}$	4	16	$1\frac{1}{2}$
C++	$23\frac{1}{2}$	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	$15\frac{1}{2}$	13	5	$10\frac{1}{2}$
java	6	5	3	5	9	4	8	8	11	$9\frac{1}{2}$	3			$2\frac{1}{2}$				3
PHP	$\frac{1}{2}$		–	1	1	–	2	1	–		2	–	1		–	1		–
basic			–			–			–			–		1	–		1	–
C#	4	2	3		$\frac{1}{2}$	1						3		$\frac{1}{2}$	–			–
python	20	6	–	12	2	–	4	$\frac{1}{2}$	–	6	1	–			–			–
psevdokoda	6		–	4		–	8		–	–	–	–	–	–	–	–	–	–
nič	1	1		1	5		1				1		3			1	2	

Število tekmovalcev, ki so uporabljali posamezni programski jezik.

Nekateri uporabljajo po dva različna jezika (pri različnih nalogah) in se štejejo polovično k vsakemu jeziku. „Nič“ pomeni, da tekmovalec ni napisal nič izvorne kode. Znak „–“ označuje jezike, ki se jih tisto leto v tretji skupini ni dalo uporabljati. 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“.

in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Večina je s C-jem zadovoljna (27/60 v prvi skupini, 20/31 v drugi, 14/17 v tretji), vendar je v prvi skupini zdaj že več kot polovica ljudi takih, ki pravijo, da rešitev v C-ju ne razumejo. (Mogoče nekateri od njih ne bi razumeli rešitev niti, če bi bile v kakšnem drugem jeziku, ker je njihovo znanje zaenkrat še prešibko.) Med jeziki, ki bi jih radi videli namesto C-ja, jih največ omenja C++ in C#, v prvi skupini pa tudi 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 povprečju pa so se tekmovalci letos spet rahlo pomladili. V prvi skupini so tekmovali tudi trije osnovnošolci (dva iz 8. in eden iz 9. razreda), ki jih pri izračunu povprečnega letnika v spodnji tabeli nismo upoštevali.

	Št. tekmovalcev po letnikih					Povprečni
Skupina	OŠ	1	2	3	4	letnik
prva	3	19	20	17	15	2,4
druga		4	6	11	18	3,1
tretja				10	7	3,4

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“; obojih je približno enako, v prvi skupini pa je skoraj

prav toliko tudi tekmovalcev, ki pravijo, da so se programirati naučili na krožkih (v prejšnjih letih je bilo takšnih manj).

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. Še posebej v prvi skupini pa je letos precej več kot lani 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	66	0	2	2	30	28	26	7	3	23	6	35	1	21	40	21	20	14	26	19	18	18	28
II	36	2	0	2	23	22	5	7	5	6	2	23	5	6	20	8	4	9	9	13	8	10	9
III	13	3	1	1	8	3	4	4	0	3	0	9	1	7	4	5	1	4	2	5	2	5	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 pomotoma bil na računalnikih dostopen interpreter pythona (mišljeno je, da so na računalnikih le urejevalniki besedil, ne pa tudi interpreterji in prevajalniki).

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) Pri nalogah tipa „opiši postopek“ se pogosto zgodi, da reševalec v resnici opiše program; rezultat tega je opis, ki je precej manj berljiv, kot če bi tisti program kar napisal (kar bi bil načeloma tudi čisto veljaven način opisa postopka pri takšni nalogi). Tule je odlomek iz enega od letošnjih primerov:

> Program s **for** zanko najprej primerja vsa števila, če je že kakšno večje od m. Če je, izpiše število in njegovo mesto. Če ni, uporabimo **while** zanko, da dodamo v **for** zanko še eno **for** zanko, ampak tako, ki gre od i + 1 do len(a) − 1.

(1.1) Komentar na koncu ene od rešitev:

> P.S. to naj bi bila psevdokoda

(1.1) Rešitev z obilico nepotrebnega dela:

> V a-ju pogledamo, če obstaja zaporedje, ki ima 1 člen in je večji od m. Če obstaja, ga dodamo v nov seznam b. Potem pogledamo zaporedje, ki ima 2 člena in ponovimo postopek. To naredimo za vse člene do n. Nato v seznamu b najdemo tako zaporedje, ki ima najkrajše število členov.

Ampak saj smo jih vendar pripravljali po naraščajoči dolžini! Zakaj se torej ne bi ustavili takoj, ko smo sploh karkoli dodali v b?!

(1.2) Dobra ideja za nov operator:

```
if (Tehtaj() > (TreTeza && 0)) { // če je teža večja od prejšnje oz. nič, jo shrani...
```

(1.2) Nek tekmovalec je poleg prireditvenega operatorja vpeljal še neprireditveni operator:

```
if i > a:        # če se teža vrečke veča
   i == a        # se jo zapisuje v a
elif i =< a:     # če se teža manjša
   i =! a        # a ostane enak
```

(1.2) Še en prispevek k širitvi sintakse primerjalnih operatorjev:

```
if (Tehtaj(i) < MinTeza[i] || > MaxTeza[i])
```

(1.2) Presenetljivo veliko ljudi komplicira, ko je treba izvesti neskončno zanko. Namesto da bi rekli

```
while True: ...
```

ali pa, če že ne poznajo logičnih spremenljivk:

```
while 0 < 1: ...
```

raje komplicirajo v stilu

```
b = 0
while b < 1: ...
```

(1.3) Nekateri tekmovalci so skušali prepoznavati majevsko številko kot celoto namesto po znakih. V ta namen so si morali pripraviti vse možne zapise posamezne številke; pri enem so nastale takšne reči:

```
if (char [0] = {'.|.')
if (char [0] = {'|..')
if (char [0] = {'..|')
   X = 7;
```

Če odmislimo nenavadno sintakso, je glavna težava tega pristopa v tem, da je možnih zapisov neugodno veliko. Koda za prepoznavanje številke je bila pri tem tekmovalcu dolga okoli 70 vrstic, pri čemer je pokril le tiste zapise, v katerih se dvopičja sploh ne pojavljajo, pik ni več kot štiri, črte pa (če jih je več) vse stojijo skupaj. Nek drug tekmovalec je za prepoznavanje številk na ta način porabil celo 110 vrstic, vendar je seveda še vedno izpustil ogromno možnih zapisov raznih številk. Kot smo videli na str. 49, je vseh možnih zapisov vseh številk od 0 do 19 kar 37 660.

(1.3) Zakaj bi označevali seštevanje s + in množenje s ·, če smo lahko bolj ekscentrični:

```
vsota = vsota · y : 20^n;       // seštevanje vsote glede na presledke v besedilu
```

(Ta rešitev je bila pisana ročno na papir, tako da si je te reči lažje privoščil.)

(1.3) Verjetno najdrznejša letošnja razširitev pomena operatorjev:

```
int main()
{
   char stevilo;
   scanf("%s" . stevilo);
   5 = ('|' || '.....' || '::.' || '.::' || ':.:');
   1 = '.';
}
```

(1.3) Še ena zanimiva razširitev jezika C++:

```
int . = 1, : = 2, .. = 2, | = 5, 0 = 0, stevilo;
cin ≪ stevilo;
```

(1.4) Eden od tekmovalcev je inicializiral spremenljivke v C++ na precej eksotičen način (ki pa je čisto veljaven):

```
bool je_ze (0);
for (int k (0); ...)
```

(1.4) Naporen način preverjanja, ali je nek znak črka:

```
if (s[i] == 'A' || s[i] == 'B' || ... || s[i] == 'Z') {
   // bolj elegantno bi bilo, če bi imel dostop do charmapa
```

Pri tem tam v resnici niso bile tri pike, ampak eksplicitno napisani pogoji še za vse ostale črke. Žalostno je, da „dostop do charmapa“ dejansko ima — standardna funkcija `isalpha()` — da ne govorimo o tem, da bi lahko ta pogoj zapisal tudi preprosto kot `if (s[i] >= 'A' && s[i] <= 'Z')`...

(1.4) Dekadenten pristop k preverjanju, ali je znak `i` številka:

```
if int(i) in range(0, 10):
```

Pri tem se iz rešitve ene od ostalih nalog jasno vidi, da zna uporabljati operator <.

(1.5) Komentar na začetku ene od rešitev:

```
// Koda bi bila lahko boljša, ampak kaj pa pričakujete od lačnega človeka?
```

Mogoče to, da bo pred odhodom na tekmovanje zajtrkoval?

(1.5) Dekadenten pristop k branju standardnega vhoda:

```
for (int i = 0; i < 12; i++)
{
  mesec[i] = new Scanner(System.in).nextInt();
}
```

(1.5) Odlomek, ki ilustrira slabe strani programiranja z ukazoma copy in paste:

```
double janPor = jan / 31;
double febPor = jan / 28;
⋮
double decPor = jan / 31;
```

V tej rešitvi so bile tudi skoraj vse druge reči podobno razmnožene po dvanajstkrat, celoten program pa je bil dolg kar 290 vrstic, s čimer je postal najdaljša letos prejeta rešitev. Še najbolj velikodušna razlaga te grozote bi bila, da avtor pač ni znal uporabljati tabel (arrayev), kar pa pade v vodo, ko opazimo, da je isti tekmovalec pri 3. nalogi uporabljal celo vektorje iz STLa.

Pri tej nalogi smo prejeli še več drugih dolgih rešitev. En tak primer je 226 vrstic dolg program, ki med drugim najprej deklarira 65 celoštevilskih spremenljivk in jih vsako v svoji vrstici postavi na 0; sledi 12 kopij neke 11-vrstične zanke **for**, program pa se konča z enormnim 9 vrstic dolgim aritmetičnim izrazom, v katerem tistih 65 spremenljivk sešteje. Težava je bila očitno v tem, da reševalec ni poznal tabel (arrayev).

(1.5) Rešitev s poenostavitvijo naloge do trivialnosti:

```
# program deluje samo, če se na dan porabi več kot 24 sodčkov
```

(2.2) Rešitev s fobijo do nekaterih operatorjev:

```
k += 1    # da ni treba imeti „<=“ v kodi in imaš lahko „<“
```

(2.2) Lani smo imeli nalogo, pri kateri se je bilo treba z uporabo zanke izogniti množenju. Tokrat je šel eden od tekmovalcev še korak dlje in se izognil tudi odštevanju:

```
for (j = poz1; j <= poz2; j++) i++;
```

(2.3) Neskončna zanka, ki ni tako zelo neskončna:

```
repeat
  ⋮
until 1 = 1; (* do neskončnosti *)
```

(2.3) Prijetno ekscentričen način za preverjanje, ali sta dve vrednosti enaki:

```
else if preveri − b <> 0 then
```

(2.3) Rešitev s sklicevanjem na nadnaravne sile:

```
int CudeznaMetoda(int barve[], int k[]) {
  int smer;
  // naredimo nekaj čudežnega
  return smer;
}
⋮
int smer = CudeznaMetoda(barve[], k[]); // s čudežno metodo, ki v C-ju žal ne obstaja,
                        // preverimo, če se tabeli barv na kozarcu in v programu skladata
```

(2.3) Rešitev s sklicevanjem na psevdo funkcije (ki se od pravih funkcij po vsem videzu sodeč ločijo predvsem po tem, da ne obstajajo):

```
def PreveriZaporedje():
  # psevdo-funkcija, ki naj bi preverila zaporedje barv
  # vrne True, če je zaporedje pravilno, drugače vrne False
```

Implementiral pa je seveda ni `:)`

(2.3) Rešitev z neodločno inicializacijo spremenljivk:

```
prazen = true;
barva = 0;
prazen = false;
```

(2.3) Glavni del ene od rešitev:

```
do
{
  Obrni();
  n−−;
  if (Preveri() == 0)
    Pobarvaj(Preveri() + 1);
} while (n >= 0);
```

(2.4) Filozofska rešitev:

> Drugače ustvarimo novo tabelo in število zapišemo tja, spet pa ugotavljamo podobnost števila v drugi tabeli, z števili ki so.

To je verjetno koristno; s števili, ki jih ni, bi bilo težje ugotavljati podobnost.

(2.4) Z začetka ene od rešitev:

> Najprej ko bi dobil telefonske številke bi jih razporedil po dolžini.

Na srečo s tem ne bi bilo veliko dela, saj naloga pravi, da so vse številke enako dolge.

(2.5) Koristen skok:

```
    CMP 0, x
    JL zacetek
    JE konec
konec:
```

Pri tem na labelo konec ne kaže noben drug skok. Podobne reči se pojavijo še pri več drugih tekmovalcih.

(3.1) Nagrado za najbolj nepotrebno rabo stavka **goto** letos dobi:

```
33:
  if (length(q) = 0) or (qpos = length(q)) then goto 34;
  ⋮
  if st_kupcev = n then goto 34;
  goto 33;
34:
```

Od drugod na tidve labeli ne skače. Z drugimi besedami, to bi bila lahko zanka **while** in mogoče še en stavek **break** na koncu...

(3.1) Prijetno ekscentrično kombiniranje dveh različnih načinov dostopanja do elementov tabele:

```
int ..., kx[1000], ky[1000], ...;
⋮
ky[a] = yy;
*(kx + a) = xx;
```

Po svoje je škoda, da ni uporabil še tretje možnosti, a[kx]... :)

(3.3) Nagrada za največjo izhodno datoteko: eden od tekmovalcev je generiral grafe z $O(n)$ točkami in $O(n)$ povezavami. Pri največjem testnem primeru (ki je imel n okoli 63 milijonov) je uspel zgenerirati malo čez 540 MB veliko izhodno datoteko, preden ga je ocenjevalni sistem prekinil zaradi prekoračitve časovne omejitve. Če bi ga pustili teči do konca, bi bila izhodna datoteka še približno 40 % daljša. Žalostno dejstvo je, da pri teh njegovih gromozanskih grafih večina točk sploh ni dosegljivih iz točke 1 in zato od 1 do 2 obstaja vedno natanko 28 poti, ne pa n, kot zahteva naloga.

(3.4) Eden od tekmovalcev je pri tej nalogi oddal program, ki izpisuje naključne odgovore.

```
Randomize;
for a := 0 to n − 1 do
  if Random(2) = 0 then WriteLn(outp, 'DA') else WriteLn(outp, 'NE');
for n := 0 to 1111111 do ;
```

Na to možnost smo pomislili tudi pri sestavljanju naloge in jo zato formulirali tako, da je vsak testni primer v resnici sestavljen iz več naborov točk, torej bi se odgovor programa pri posameznem testnem primeru štel za pravilnega le, če bi uganil pravilni rezultat pri vseh naborih točk v njem. Z drugimi besedami, program bi moral po

srečnem naključju uganiti pravo od 2^n možnosti, če je n število naborov točk pri opazovanem testnem primeru (saj imamo pri vsakem naboru dva možna odgovora, `DA` in `NE`). Pri naših testnih primerih je bil n povsod enak 7, razen pri najmanjšem primeru, ki je imel $n = 3$; naš tekmovalec je svojo hazardersko rešitev oddal kar osemkrat in enkrat celo pravilno rešil primer z $n = 3$, ostalih pa nikoli. Žal je zaradi velikega števila oddaj izgubil vse točke, ki bi jih dobil zaradi pravilnega odgovora na en testni primer. Z enakim uspehom je ta tekmovalec poskusil naključni pristop tudi pri drugi nalogi (kompresija slike).

(3.5) Nagrado za najgloblje gnezdenje template parametrov dobi:

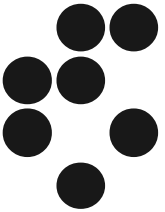
```
vector<pair<int, pair<int, pair<int, int> > > > sto;
```

Je pa vsekakor spodbudno, da letos v tretji skupini že kar precej tekmovalcev uporablja razne generične tipe in podatkovne strukture.

(tekmovanje programov) Posebno nagrado za najglobljo indentacijo pa si zasluži eden od prispevkov za tekmovanje programov, v katerem dosega najgloblje gnezdenje kar 23 nivojev. Iz očitnih razlogov tega primera tukaj ne moremo prikazati v celoti, lahko pa naštejemo, kaj se dogaja na vsakem nivoju gnezdenja: podprogram, if, for, if, if, if, for, if, if, if, if, if, if, for, if, if, if, if, if, if, for, if, if, klic podprograma.

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.

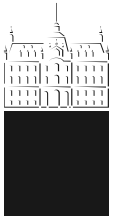
Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi 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.

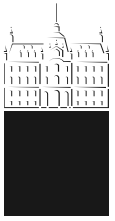
*



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.

SREBRNI POKROVITELJ

Our Space

Podjetje Our Space d. o. o. je del Our Space Skupine, ki z ločenimi podjetji na trgu že več kot 13 let ponuja zaokroženo celoto storitev in produktov s področja informacijskih tehnologij.

Vrsto let smo se ukvarjali z implementacijami poslovno informacijskih sistemov, podporo uporabnikov ter razvojem lastnih rešitev, zadnjih nekaj let pa smo vse bolj prisotni tudi na področju poslovnega svetovanja, upravljanja poslovnih procesov, vodenja projektov in zagotavljanja kakovosti.

BRONASTI POKROVITELJI

arnes

cosylab

CONTROL SYSTEM LABORATORY

ENVIGENCE

environmental intelligence

Fortheia

Pametne rešitve