8. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 23. marca 2013

Bilten

Bilten 8. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2014

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Primož Gabrijelčič, Boris Gašperin, Matija Grabnar, Tomaž Hočevar, Boris Horvat, Nace Hudobivnik, Jurij Kodre, Mitja Lasič, Mark Martinec, Polona Novak, Jure Slak, Boštjan Slivnik, Mitja Trampuš, Janez Brank.

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

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

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

37.091.27:004(497.4)(0.034.2)

TEKMOVANJE ACM v znanju računalništva (8 ; 2013 ; Ljubljana)
Bilten [Elektronski vir] / 8. tekmovanje ACM v znanju računalništva, Institut Jožef Stefan, Ljubljana, 23. marca 2013 ; avtorji nalog Nino Bašić ... [et al.] ; uredil Janez Brank. — El. knjiga. — Ljubljana : Institut Jožef Stefan, 2014

Način dostopa (URL): `http://rtk.ijs.si/2013/rtk2013-bilten.pdf`

ISBN 978-961-264-053-8 (PDF)
1. Bašić, Nino 2. Brank, Janez, 1979–
272322560

KAZALO

STRUKTURA TEKMOVANJA

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

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

V lažjih dveh skupinah traja tekmovanje tri ure; tekmovalci lahko svoje rešitve napišejo na papir ali pa jih natipkajo na računalniku, nato pa njihove odgovore oceni temovalna komisija. Naloge v teh dveh skupinah večinoma zahtevajo, da tekmovalec opiše postopek ali pa napiše program ali podprogram, ki reši določen problem. Pri pisanju izvorne kode programov ali podprogramov načeloma ni posebnih omejitev glede tega, katere programske jezike smejo tekmovalci uporabljati. Podobno kot v zadnjih nekaj letih smo tudi letos ponudili možnost, da tekmovalci v prvi in drugi skupini svoje odgovore natipkajo na računalniku; tudi tokrat so se zanjo odločili skoraj vsi tekmovalci.

V tretji skupini rešujejo vsi tekmovalci naloge na računalnikih, za kar imajo pet ur časa. Pri vsaki nalogi je treba napisati program, ki prebere podatke iz vhodne datoteke, izračuna nek rezultat in ga izpiše v izhodno datoteko. Programe se potem ocenjuje tako, da se jih na ocenjevalnem računalniku izvede na več testnih primerih, število točk pa je sorazmerno s tem, pri koliko testnih primerih je program izpisal pravilni rezultat. (Podrobnosti točkovanja v 3. skupini so opisane na strani 22.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C# in java.

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

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

Na začetku smo tekmovalcem razdelili tudi list z nekaj nasveti in navodili (str. 7–9 za 1. in 2. skupino, str. 21–23 za 3. skupino).

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

Poleg tekmovanja v znanju računalništva smo organizirali tudi tekmovanje v off-line nalogi, ki je podrobneje predstavljeno na straneh 163–164.

Podobno kot v zadnjih treh letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 25. januarja 2013. 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. 151–154). 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 243 tekmovalcev z 29 šol.

NASVETI ZA 1. IN 2. SKUPINO

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

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

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

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

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

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

Pri ocenjevanju so vse naloge vredne enako število točk. Svoje odgovore dobro utemelji. Prizadevaj si predvsem, da bi bile tvoje rešitve pravilne, ob tem pa je zaželeno, da so tudi čim bolj učinkovite; take dobijo več točk kot manj učinkovite (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). 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 (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

1. Vandali

Vandal bi iz javne table, ki vsebuje dolg napis T_1, rad naredil nek krajši napis T_2 (niza T_1 in T_2 sta dana) tako, da s flomastrom zakrije določene črke. Recimo:

```
  T2 :  dol z vlado
  T1 :  Še dobro, da lahko z volitvami vplivamo na dobrobit naroda.
izpis :  ###do########l#### z v###########l#######a#d###o###########
```

Napiši podprogram `Vandal(T1, T2)`, ki dobi niza T_1 in T_2 ter izpiše vandalizirano različico niza T_1, v kateri so nekateri znaki spremenjeni v `#` tako, da preostali znaki tvorijo ravno niz T_2.

Predpostavi, da se T_2 zagotovo pojavlja nekje znotraj napisa T_1; če je pojavitev več, je vseeno, katero uporabiš. (Na primer: pri T_2 = `abc` in T_1 = `cabdbc` lahko izpišemo `#ab##c` ali pa `#a##bc`.)

Tvoj podprogram naj bo takšne oblike:

```
procedure Vandal(T1, T2: string);                { v pascalu }
void Vandal(char *T1, char *T2);                 /* v C/C++ */
void Vandal(string T1, string T2);               // v C++
public static void Vandal(String T1, String T2); // v javi
public static void Vandal(string T1, string T2); // v C#
def Vandal(T1, T2): ...                          # v pythonu
```

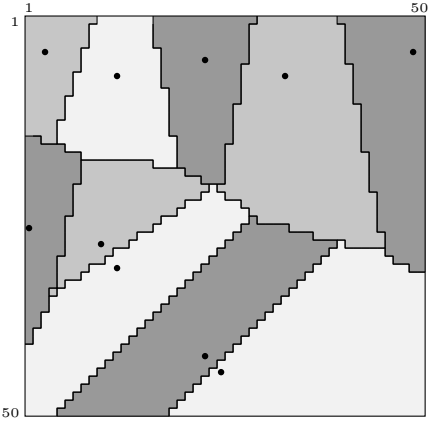
2. Kolera

V devetnajstem stoletju se še ni kaj dosti vedelo o načinu prenašanja nalezljivih bolezni. Leta 1854 so imeli v Londonu velik izbruh črevesne bolezni kolere. Zdravnik John Snow je takrat s svojo analizo pokazal na vzročno zvezo med lokacijo londonskih vodnjakov in lokacijo domovanja obolelih. Izkazalo se je, da so žrtve pile vodo iz istega okuženega vodnjaka na ulici Broad Street. Zdravnik si je narisal zemljevid Londona, vanj vrisal lokacije vodnjakov in za vsak vodnjak z drugo barvo pobarval območje zemljevida, ki je temu vodnjaku najbližje. Ko je vrisal še točke domovanja obolelih, je postala vzročna zveza precej očitna, saj so prebivalci večinoma hodili po vodo k njim najbližjemu vodnjaku.

Nekoliko si poenostavimo zemljevid velemesta in denimo, da nas zanima le območje, ki ga razdelimo na kvadratno mrežo 50×50 celic. V nekaterih celicah te mreže se nahajajo vodnjaki. Vseh vodnjakov je deset, njihove koordinate preberemo z vhodne datoteke: v vsaki vrstici sta dve celi števili, x in y (obe sta z območja od 1 do 50).

Napiši program, ki bo prebral koordinate vodnjakov, potem pa za vsak vodnjak izpisal, kolikšno površino zemljevida pokriva, t.j. kolikšnemu številu celic v tej pravokotni mreži je ta vodnjak najbližji. Za merjenje razdalje med dvema celicama uporabimo Pitagorov izrek: kvadrat razdalje med (x_1, y_1) in (x_2, y_2) je $(x_1 - x_2)^2 + (y_1 - y_2)^2$. Če je od neke celice razdalja do več vodnjakov enaka, je vseeno, h kateremu vodnjaku štejemo tako celico. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `kolera.txt` (kar ti je lažje).

Primer vhodnih podatkov:	Eden od možnih izpisov:
49 5	240
33 8	420
1 27	115
23 43	325
25 45	452
12 8	206
10 29	158
3 5	97
23 6	184
12 32	303



Prikazan je seveda le eden od številnih možnih izpisov pri teh vhodnih podatkih (saj naloga pravi, da če je neka celica enako oddaljena od dveh ali več vodnjakov, je

vseeno, h kateremu jo štejemo). Črne pike na sliki označujejo položaje vodnjakov, osenčeni liki pa kažejo dele mreže, ki so najbližje posameznemu vodnjaku.

3. Kino

Mirko rad hodi v kino (običajnega, z eno samo dvorano), kjer vrtijo filme brez prekinitev enega za drugim. Domov bo šel z avtobusom, vendar pa sovraži stati na postaji in čakati na avtobus. Po vsakem filmu se odloča, ali naj gre na avtobus ali naj pogleda še en film. Pred seboj ima dva seznama:

- vozni red avtobusov: npr. $[(0, 40), (1, 50), (2, 30), (3, 30), (4, 30), (5, 35), (6, 5)]$; to so pari (ure, minute), ki povedo, kdaj odpeljejo avtobusi s postaje pred kinom; pri tem se čas meri od nekega izbranega začetnega trenutka, zato lahko število ur sčasoma tudi preseže 24;
- seznam trajanj filmov, ki jih vrtijo v kinu: na primer $[(1, 30), (2, 10), (1, 40), (0, 50)]$, torej spet v obliki parov (ure, minute).

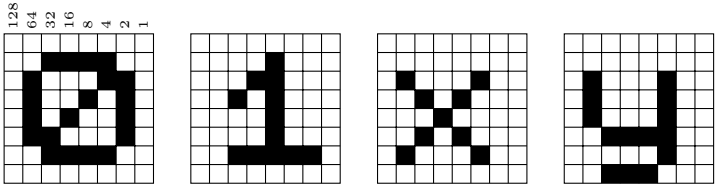
Prvi film se začne ob trenutku $(0, 0)$. Med filmi ni prekinitve. Mirko pogleda vedno vsaj prvi film. Vhodni podatki so taki, da zagotovo lahko dobi avtobus za domov, če pogleda samo prvi film. Rad bi minimiziral čas čakanja na postaji (to med drugim tudi pomeni, da seveda noče ostati v kinu tako dolgo, da bi zamudil še zadnji avtobus). **Opiši postopek**, ki izračuna, koliko filmov naj pogleda. Mirko lahko pride od kina do avtobusne postaje v trenutku, tako da lahko ujame avtobus tudi, če le-ta odpelje ob istem času, ob katerem se konča zadnji film, ki si ga je ogledal.

Pri zgornjem primeru se na primer izkaže, da je najbolje, če pogleda tri filme. V tem primeru mora čakati na avtobus 15 minut; če bi pogledal en film, bi moral čakati 20 minut, po drugem filmu bi moral čakati kar 50 minut, po četrtem filmu pa bi celo zamudil zadnji avtobus (in to za 5 minut).

4. Igličnik tiskalnik

Na nekaterih starejših računalnikih je bil vsak znak predstavljen s črno-belo sliko velikosti 8×8 točk. Vsaka slika je bila v pomnilniku predstavljena s tabelo 8 bajtov, pri čemer je n-ti bajt vseboval opis n-te vrstice (gledano od zgoraj navzdol). Vsaka točka (piksel) v vrstici je imela svojo utež, uteži posameznih točk od desne proti levi so bile potence števila 2, torej 1, 2, 4, 8, 16, 32, 64, 128. Vrstica je bila predstavljena z vsoto uteži črnih točk.

Znaki `0`, `1`, `x` in `y` so recimo predstavljeni takole:



V tretji vrstici (od zgoraj navzdol) slike znaka `0` so druga, tretja in sedma točka (od desne proti levi) črne. Skupna vsota njihovih uteži je $2 + 4 + 64 = 70$, zato ima tretji bajt v opisu znaka `0` vrednost 70. Znak `0` je bil torej predstavljen z naslednjim zaporedjem 8 bajtov: 0, 60, 70, 74, 82, 98, 60, 0.

Napiši podprogram `Izpisi(s)`, ki izpiše bitno sliko niza `s`. Pri tem naj predpostavi, da so bitne slike vseh možnih znakov že pripravljene in shranjene v globalni spremenljivki `znaki`; to je tabela 256×8 števil, pri čemer je prvi indeks številka znaka (od 0 do 255), drugi indeks pa številka vrstice (od 0 do 7). Podprogram naj niz izpiše tako, da vse črne točke v bitnih slikah izpiše kot znak „`#`“, vse bele točke pa kot pike „`.`“. Tako bi podprogram `Izpisi` na primer izpisal niz `001xy` kot

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

Še deklaracije v raznih programskih jezikih:

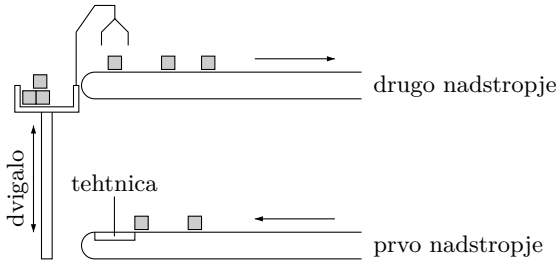
```
unsigned char znaki[256][8];                    /* v C/C++ */
void Izpisi(char *s);                           /* v C/C++ */
void Izpisi(string s);                          // v C++

var znaki: array [0..255, 0..7] of byte;        { v pascalu }
procedure Izpisi(s: string);

public static byte[,] znaki = new byte[256, 8]; // v C#
public static void Izpisi(string s);

public static byte[][] znaki = new byte[256][8]; // v javi
public static void Izpisi(String s);

znaki = [[...], [...], ...]                     # v pythonu
def Izpisi(s): ...
```

Nasvet: v pythonu in pascalu lahko številsko kodo znaka `c` dobiš tako, da pokličeš standardno funkcijo `ord(c)`.

5. Dvigalo

V dvonadstropni tovarni robotizirano dvigalo skrbi za prenos zabojev med nadstropji. Zaboji prihajajo po tekočem traku na tehtnico, ločen sistem pa trak zaustavi, kadar je na tehtnici zaboj, ter ga požene, ko se izprazni.

Del dvigala z nosilnostjo 500 kg je robotska roka, ki v spodnjem nadstropju prelaga zaboje s tehtnice v dvigalo, v zgornjem nadstropju pa jih razlaga iz dvigala na trak, po katerem se avtomatsko odpeljejo v nadaljnjo obdelavo.



Napiši program, ki se vrti v neskončni zanki in upravlja dvigalo. Poskrbeti moraš, da masa zabojev na dvigalu nikoli ne preseže nosilnosti dvigala. Predpostavi, da je dvigalo na začetku prazno in parkirano v prvem (torej spodnjem) nadstropju ter da imajo zaboji maso od 1 do 500 kg (vključno). Želimo, da je število premikov med nadstropji čim manjše. (Dvigalo torej vedno naložimo kar najbolj.)

Na voljo imaš naslednje podprograme (funkcije):

- **int** Stehtaj() — Vrne maso zaboja na tehtnici. Če na tehtnici ni zaboja, funkcija počaka, dokler se zaboj ne pojavi, nato ga stehta in vrne rezultat.
- **void** Nalozi() — Z robotsko roko prenese zaboj s tehtnice na dvigalo. Če pri tem masa zabojev na dvigalu preseže nosilnost, se dvigalo pokvari. Podprogram se vrne šele, ko je zaboj naložen. Če je tehtnica prazna ali pa je dvigalo v drugem nadstropju namesto v prvem, podprogram ne naredi ničesar in se takoj vrne.
- **void** Razlozi() — Z robotsko roko prenese enega od zabojev z dvigala na trak v drugem nadstropju. Če je bil trak še zaseden (ker je bil na njem nek prejšnji zaboj), Razlozi najprej počaka, da se prejšnji zaboj odpelje. Če je dvigalo v prvem nadstropju namesto v drugem, Razlozi ne naredi ničesar in se takoj vrne.
- **void** OdpeljiDvigalo(**int** StNadstropja) — Odpelje dvigalo v izbrano nadstropje. Podprogram se vrne šele, ko dvigalo prispe na cilj. Parameter StNadstropja ima lahko vrednost 1 ali 2.

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 (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

1. Binarni sef

Binarni sef ima na številčnici dve tipki: 0 in 1. Če ima sef šifro 011, se bo odprl takrat, ko bodo zadnje tri pritisnjene tipke zaporedno 0, 1 in 1. Če ne poznamo šifre, lahko poskusimo sef odpreti tako, da vtipkamo vseh 8 trimestnih kombinacij. Vendar gre tudi hitreje — če vtipkamo na primer zaporedje 0001011100, se bodo vrata zagotovo odprla, saj se v tem zaporedju pojavljajo kot podnizi kombinacije 000, 001, 010, 101, 011, 111, 110, 100 — kar je ravno vseh osem kombinacij. **Napiši podprogram** Sef(s, n), ki bo za dano zaporedje s in znano dolžino šifre (n znakov) preveril, ali zaporedje s zagotovo odpre sef. Predpostaviš lahko, da je n (dolžina šifre) največ 12, zaporedje s pa je dolgo največ 10 000 znakov.[1]

Tvoj podprogram naj bo takšne oblike:

```
function Sef(s: string; n: integer): boolean;   { v pascalu }
bool Sef(char *s, int n);                       /* v C/C++ */
bool Sef(string s, int n);                      // v C++
public static boolean Sef(String s, int n);     // v javi
public static bool Sef(string s, int n);        // v C#
def Sef(s, n): ...                              # v pythonu; naj vrne True/False
```

[1] Zanimiva je tudi težja različica naloge, pri kateri dobimo samo n, poiskati pa hočemo najkrajši tak niz s, ki vsebuje kot podnize vsa možna zaporedja n bitov.

2. Sumljiva imenovanja

Pravkar ustanovljena Komisija za nadzor zaposlovanja v javni upravi dobiva številne vloge za oceno zakonitosti imenovanja novih uradnikov. Vsaka vloga je sestavljena iz treh stvari:

- seznam kvalifikacij, ki so zahtevane za to delovno mesto;
- seznam kvalifikacij, ki jih je imel človek, ki je to delovno mesto dobil;
- nato pa še seznami kvalifikacij vseh ostalih kandidatov, ki so se potegovali za to delovno mesto, vendar ga niso dobili.

Komisija mora za vsako imenovanje ugotoviti, ali je *zakonito*, *sumljivo* ali celo *nezakonito*. Pri tem uporablja naslednja pravila:

- imenovanje je *nezakonito*, če kandidat nima vseh kvalifikacij, zahtevanih na tem delovnem mestu;
- imenovanje je *sumljivo*, če kandidat sicer ima vse potrebne kvalifikacije, vendar obstaja boljši kandidat; s tem je mišljen tak kandidat, ki ima vse kvalifikacije kot tisti, ki je bil imenovan, in še kakšno zraven;
- sicer je imenovanje *zakonito*.

Napiši program, ki prebere vlogo in izpiše, ali je bilo imenovanje nezakonito, sumljivo ali zakonito.

Vhodni podatki: privzemimo, da obstaja natanko n (za nek $n \leq 32$) možnih kvalifikacij. Tako lahko seznam zahtevanih kvalifikacij podamo kar z nizom oblike: $s = c_1c_2c_3 \ldots c_{n-1}c_n$, kjer je lahko vsaka izmed črk c_i le `D` ali `N` in kjer črka `D` na i-tem mestu pomeni, da je i-ta kvalifikacija zahtevana, črka `N` pa, da ni. Na primer, niz `DNDNDNNNDDDNNDNDNDNNDNNNDDDNNDNN` predstavlja delovno mesto, kjer so zahtevane kvalifikacije št. 1, 3, 5, 9, 10, ..., kvalifikacije št. 2, 4, 6, 7, 8, ... pa ne. Podobno predstavimo seznam kvalifikacij, ki jih imajo kandidati. Tako kot prej črka `D` na i-tem mestu pomeni, da kandidat i-to kvalifikacijo ima, črka `N` pa, da je nima.

Tvoj program lahko vhodne podatke bere s standardnega vhoda ali pa iz datoteke `vhod.txt` (kar ti je lažje). V prvi vrstici je niz, ki predstavlja zahtevane kvalifikacije za delovno mesto; druga vrstica predstavlja kvalifikacije kandidata, ki je bil zaposlen; vse naslednje vrstice pa predstavljajo kvalifikacije vseh ostalih kandidatov.

Tvoja rešitev naj deluje tudi v primerih, ko je število vrstic v vhodni datoteki zelo veliko — preveliko, da bi celotno vsebino vhodne datoteke naenkrat shranili v pomnilnik.

Predpostaviš lahko, da obstaja funkcija `vStevilo(s)`, ki spremeni niz kvalifikacij `s` v število, ki ima v binarnem zapisu na i-tem mestu 1, če je imel niz kvalifikacij tam črko `D` in 0, če je bila tam črka `N`. Ni pa nujno, da to funkcijo uporabiš (nalogo lahko čisto dobro rešiš tudi brez nje).

Izhodni podatki: izpiši, ali je bilo imenovanje nezakonito, sumljivo ali zakonito.

Primer vhodne datoteke:

```
NNDDDDDDNDDNDNNDNDDDNNDNDNNNDDNN
DDDDDDDDNDDDDDDDDDDDNDDDDDDDDDNN
DDDDDDDDNDDDDDDDDDDDDDDDDDDDDDND
NDNNDDDNNNNNNDNNNNDNDNNDDNNDNNNN
NNNNNDNDNDDNNDDNDNNNDNDNDDNDDDDD
NNNDDNDNDNDDDNDDDNNNNDDDDDNNNDND
NDDDDNNNDNNNNDDNNNNNNDNNNDNNDNDN
NNDDDNDDNDDNNDDDDDDNDDDDNNNDDDNN
NNNNNDNDNDDDNDNDDNDNDNNNNDNDNNNN
NDDDNNNDDDNNDNDDNNNDNDDDNNDNNNND
NNNNDNDDDDDDNNDNNDDDDDDNDNDNNDDN
```

Pripadajoč izpis:

```
sumljivo
```

(Pojasnilo: drugi kandidat ima vse kvalifikacije, ki jih ima prvi, in še dve taki, ki ju prvi nima, zato je imenovanje sumljivo.)

3. Dekodiranje nizov

Dan je naslednji postopek za kodiranje besedila (niza znakov):

- Niz začnemo brati od začetka, dokler ne pridemo do prvega samoglasnika. Nato vse prebrane znake, vključno s tistim samoglasnikom, napišemo v obratnem vrstnem redu. Ob tem si shranimo podatek, koliko znakov smo zamenjali pri tej prvi menjavi.

- Sedaj preberemo vse znake od prvega samoglasnika dalje (toda brez tega samoglasnika), dokler ne pridemo do drugega samoglasnika. Zopet vse prebrane znake, vključno z drugim samoglasnikom, napišemo v obratnem vrstnem redu. S tem je drugi samoglasnik prišel na drugo mesto v nizu, takoj za prvim samoglasnikom. Shranimo podatek, koliko znakov smo zamenjali pri tej drugi menjavi.

- Postopek ponavljamo, dokler ne pridemo do konca besedila.

Primer: oglejmo si postopek kodiranja niza „`SOBA Z RAZGLEDOM NA VRT`“. Za samoglasnike štejejo znaki `A`, `E`, `I`, `O` in `U`. V vsaki vrstici je podčrtan tisti del niza, ki ga bomo v tem koraku obrnili; številka na desni pa je dolžina tega dela.

```
SOBA Z RAZGLEDOM NA VRT   (2)
OSBA Z RAZGLEDOM NA VRT   (3)
OABS Z RAZGLEDOM NA VRT   (7)
OAAR Z SBZGLEDOM NA VRT  (10)
OAAELGZBS Z RDOM NA VRT  (11)
OAAEODR Z SBZGLM NA VRT  (14)
OAAEOAN MLGZBS Z RD VRT
```

Opiši postopek za dekodiranje tako zakodiranega niza. Postopek torej kot vhodne podatke dobi kodirani niz (v zgornjem primeru: `OAAEOAN MLGZBS Z RD VRT`) in seznam s številom zamenjanih znakov pri vsaki menjavi (v zgornjem primeru: 2, 3, 7, 10, 11, 14). Predpostavi, da niz vsebuje le velike črke angleške abecede (od `A` do `Z`) in presledke.

Dekodiranje za zgornji primer:

```
(14)  OAAEOAN MLGZBS Z RD VRT
(11)  OAAEODR Z SBZGLM NA VRT
(10)  OAAELGZBS Z RDOM NA VRT
 (7)  OAAR Z SBZGLEDOM NA VRT
 (3)  OABS Z RAZGLEDOM NA VRT
 (2)  OSBA Z RAZGLEDOM NA VRT
      SOBA Z RAZGLEDOM NA VRT
```

4. Silhuete

Neko mesto ima obliko pravokotne mreže, razdeljene na $w \times h$ parcel (w stolpcev, h vrstic). Na vsaki parceli stoji stolpnica, ki ima obliko kvadra. V tlorisu so si vsi ti kvadri enaki, razlikujejo pa se po višini; na parceli v r-ti vrstici in c-tem stolpcu stoji nebotičnik z višino a_{rc} (to je celo število, večje od 0).

Mesto si (recimo s fotoaparatom) „ogledamo“ od spredaj in od strani. Na tak način dobimo samo podatek o najvišji stolpnici v vsaki vrstici in vsakem stolpcu. Recimo, da je x_c višina najvišje stolpnice v c-tem stolpcu, y_r pa višina najvišje stolpnice v r-ti vrstici. Iz teh podatkov (torej $x_1, x_2, \ldots, x_w$ in $y_1, y_2, \ldots, y_h$) sicer v splošnem ne moremo natančno določiti višin posameznih stolpnic (torej posameznih vrednosti a_{rc}), lahko pa preverimo, ali podatki sploh predstavljajo možno konfiguracijo in izračunamo eno od možnih postavitev.

Napiši program, ki prebere vhodne podatke in izpiše eno od možnih postavitev ali pa „`taka postavitev ne obstaja`“, če podatki niso veljavni. Vhodni podatki so zapisani v treh vrsticah in sicer takole:

$$\begin{array}{cccc} w & h & & \\ x_1 & x_2 & \ldots & x_w \\ y_1 & y_2 & \ldots & y_h \end{array}$$

Tvoj program jih lahko bere s standardnega vhoda ali pa iz datoteke `silhuete.txt` (kar ti je lažje). Predpostavi, da velja $w \leq 1000$ in $h \leq 1000$.

Primer: recimo, da imamo mrežo 4×3 stolpnic z višinami

9	2	3	2
2	1	6	1
6	3	6	9

Maksimalne višine po stolpcih so torej (od leve proti desni) 9, 3, 6, 9; po vrsticah pa (od zgoraj navzdol) 9, 6, 9. Program bi v tem primeru kot vhodne podatke dobil:

```
4 3
9 3 6 9
9 6 9
```

Nekaj možnih razporedov pri teh vhodnih podatkih (program bi torej lahko izpisal katerega koli izmed njih, obstaja pa še veliko drugih, ki bi bili prav tako dobri):

```
9 2 3 2        9 3 6 9        9 1 2 1        4 3 5 9
2 1 6 1        5 3 6 6        2 3 1 6        4 3 6 5
6 3 6 9        9 2 6 9        9 1 6 9        9 2 2 9
```

Primer vhodnih podatkov, pri katerih veljavna predstavitev ne obstaja:

```
2 2
4 2
1 1
```

5. Ribič

Nadebuden ribič se je odpravil k potoku, v katerem namerava z mrežo dolžine d uloviti točno k rib, ki jih bo prinesel domov za kosilo. Voda je kristalno čista, zato točno ve, kje se nahaja katera od n rib. Za potrebe naloge lahko potok predstavimo z ravno črto, ribe pa s točkami x_i na njej. Mrežo bo vrgel tako, da se bo začela na neki celoštevilski koordinati x (ki je lahko tudi negativna). Pri tem bo ujel vse ribe, ki se nahajajo na koordinatah od vključno x do vključno $x + d - 1$. **Opiši postopek**, ki izračuna, na koliko načinov lahko ribič vrže mrežo, da bo ujel točno k rib.

Predpostavi, da so dolžina d in koordinate x_i cela števila v razponu od 1 do 10^9, k in n pa sta celi števili med 1 in 10^6. Zato naj bo tvoj postopek čim bolj učinkovit (bolj učinkovite rešitve dobijo več točk). Koordinate x_i so že podane v naraščajočem vrstnem redu in nobeni dve ribi se ne nahajata na isti koordinati.

Primer: če imamo $n = 6$ rib na koordinatah $1, 5, 6, 10, 12, 15$ in mrežo dolžine $d = 8$, s katero bi radi ujeli natanko $k = 3$ ribe, potem je primernih položajev mreže (torej primernih vrednosti x) devet (to so $-1, 0, 1, 3, 4, 6, 8, 9, 10$). Položaj $x = 5$ na primer ni primeren, ker pri njem ujamemo 4 ribe, ne 3; položaja $x = 2$ in $x = 7$ nista primerna, ker takrat ujamemo le 2 ribi, ne 3.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Delovno okolje

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

Na vsakem računalniku imaš na voljo imenik `U:\_Osebno`, v katerem lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C# ali java, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz JDK 1.7 in s prevajalnikom za C# iz Visual Studia 2008. Za delo lahko uporabiš `FP` oz. `ppc386` (Free Pascal), `GCC/G++` (GNU C/C++ — command line compiler), `javac` (za javo 1.7), Visual Studio 2010 in druga orodja.

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

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

```
rtk -name JNovak
```

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

Za oddajo rešitve uporabi enega od naslednjih ukazov:

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

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Datoteka z izvorno kodo, ki jo oddajaš, ne sme biti daljša od 30 KB. Na spletni strani boš dobil za vsak testni primer obvestilo o tem, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund ali pa porabil več kot 200 MB pomnilnika, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le

standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku, razen s predpisano vhodno in izhodno datoteko. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku), prenosnih računalnikov, prenosnih telefonov itd.

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. 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. Moderna umetnost (`umetnost.in`, `umetnost.out`)

Slikar bo na prihajajoči razstavi predstavil zbirko slik, ki bodo v celoti sestavljene iz raznobarvnih navpičnih črt različnih širin. Platno je razdelil na n enako širokih stolpcev in določil, kakšne barve mora biti kateri od njih. Za slikanje bo uporabil kar slikopleskarski valj, s katerim vsakič pobarva k sosednjih stolpcev (vse z isto barvo). Ker se mu mudi, bi rad svoje izdelke končal v čim manjšem številu navpičnih potegov z valjem. Okoli platna je podložil časopisni papir, da ni škode, če z valjem seže preko roba. Prav tako lahko isti stolpec prebarva večkrat. Koliko potegov potrebuje, če slika optimalno?

Vhodna datoteka: v prvi vrstici sta dve celi števili, najprej n in nato k, ločeni s presledkom. Veljalo bo $1 \leq k \leq n \leq 10^6$. Sledi n vrstic, ki opisujejo želene barve posameznih stolpcev (od leve proti desni). Vsaka od teh vrstic vsebuje le eno celo število, to je številka barve ustreznega stolpca. Barve so oštevilčene s števili od 1 do n (pri čemer seveda ni nujno, da se vseh n možnih barv dejansko pojavlja na sliki). Pri 40 % testnih primerov bo veljalo tudi $n \leq 100$.

Izhodna datoteka: vanjo izpiši eno samo celo število, namreč najmanjše število potez z valjem, ki jih slikar potrebuje, da pobarva sliko v skladu z zahtevami naloge.

Primer vhodne datoteke:

```
7 4
2
2
2
6
6
2
2
```

Pripadajoča izhodna datoteka:

```
3
```

Komentar primera: slikar lahko z eno potezo pobarva stolpca barve 6, z drugo potezo pobarva leve tri stolpce in s tretjo še desna dva. Pri tej tretji potezi seže valj preko roba (desno od slike) in pobarva tudi nekaj podloženega časopisnega papirja. Slikar nekatere stolpce pobarva večkrat, vendar je končen videz slike pravilen, ker se barve ne mešajo in je vidna le zadnje nanešena barva.

2. Kompleksnost števil (`kompleksnost.in`, `kompleksnost.out`)

Dano celo število n bi radi izrazili kot rezultat aritmetičnega izraza, v katerem nastopajo le oklepaji, zaklepaji, operatorja + in ·, kot operand pa le število 1. Pri tem + ni dovoljeno uporabljati v primerih, ko bi za izračun takega izraza morali kdaj sešteti dve števili, večji od 1 (+ smemo torej uporabljati le za prištevanje 1).[2] **Napiši program**, ki za dani n ugotovi, kakšno je najmanjše število enic, ki jih potrebujemo, da sestavimo takšen izraz z vrednostjo n.

Primer: $n = 12$ lahko izrazimo kot

$$\begin{array}{cc} 1+1+1+1+1+1+1+1+1+1+1+1 & \text{(12 enic)} \\ (1+1+1+1+1+1)\cdot(1+1) & \text{(8 enic)} \\ (1+1+1+1)\cdot(1+1+1) & \text{(7 enic)} \\ (1+1)\cdot(1+1)\cdot(1+1+1) & \text{(7 enic)} \\ (1+1+1)\cdot(1+1+1)+1+1+1 & \text{(9 enic)} \end{array}$$

in še na veliko drugih načinov. Najmanjše število enic, ki jih potrebujemo, da sestavimo izraz z vrednostjo 12, je torej 7.

Primer neveljavnega izraza za $n = 12$:

$$((1+1)\cdot(1+1+1))+1+1+((1+1)\cdot(1+1))$$

Ko ga računamo, namreč iz njega nastane $6+1+1+4$ in ne glede na to, v kakšnem vrstnem redu bi hoteli sešteti te štiri seštevance, bi prej ali slej morali sešteti dve števili, ki bi bili obe večji od 1.

Vhodna datoteka: v prvi vrstici je celo število t, ki pove, za koliko n-jev nas pri tem primeru zanima najmanjše potrebno število enic. Sledi t vrstic, vsaka od njih vsebuje po eno celo število, to je nek n, za katerega nas zanima najmanjše potrebno število enic. Ta števila so podana v padajočem vrstnem redu.

Veljalo bo $1 \le t \le 10$. Za vsak n velja $1 \le n \le 3\,000\,000$. V 80 % testnih primerov bo za vsak n veljalo veljalo tudi $n \le 300\,000$, v 40 % testnih primerov pa celo $1 \le n \le 100$.

Izhodna datoteka: vanjo izpiši t celih števil (vsako v svojo vrstico), ki po vrsti za vsakega od n-jev iz vhodne datoteke povedo najmanjše število enic, ki jih potrebujemo, da po pravilih iz besedila naloge sestavimo aritmetični izraz z vrednostjo n.

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:
`3`	`10`
`22`	`7`
`12`	`5`
`6`	

Komentar: 22 lahko na primer z desetimi enicami računamo takole:

$$(1+(1+1)\cdot(1+(1+1)+1+1))\cdot(1+1)=22.$$

[2] Možnih je še več zanimivih različic te naloge, ki se razlikujejo po tem, katere operacije dovolimo v naših aritmetičnih izrazih: (*a*) dovolimo množenje in seštevanje podizrazov s poljubno vrednostjo (torej brez omejitve, da mora imeti eden od seštevancev vrednost 1); (*b*) dovolimo seštevanje, množenje in potenciranje; (*c*) dovolimo seštevanje, množenje in odštevanje.

3. Požar (`pozar.in`, `pozar.out`)

Podjetje ima v najemu poslovne prostore, ki so razdeljeni v pravokotno mrežo s h vrsticami in w stolpci. Vsaka celica v tej mreži ustreza eni pisarni. Vsi zaposleni so že odšli na zaslužen nočni počitek, nesreča pa žal ne počiva. Pisarno, ki se nahaja v vrstici y_0 in stolpcu x_0, je zajel močan ogenj, ki se hitro širi na sosednje pisarne. V vsaki sekundi se ogenj z goreče pisarne razširi na sosednje pisarne, ki si z njo delijo katero od štirih sten (ne pa na tiste, ki si z njo delijo le enega od vogalov). V podjetju imajo protipožarni alarm, ki pa se sproži šele, ko gori vsaj k pisarn. **Napiši program**, ki bo izračunal, po koliko sekundah se bo sprožil alarm.

Vhodna datoteka: vsebuje več požarov, za katere mora tvoj program izračunati čas alarma. V prvi vrstici je podano število požarov p. Vsaka naslednja vrtica opisuje požar s števili w, h, x_0, y_0 in k. Veljalo bo $1 \leq p \leq 20$, $1 \leq w \leq 10^9$, $1 \leq h \leq 10^9$, $1 \leq x_0 \leq w$, $1 \leq y_0 \leq h$, $1 \leq k \leq w \cdot h$.

Pri 40 % testnih primerov bosta dimenziji mreže kvečjemu 1000. Poleg tega se bo v 60 % testnih primerov alarm pri vseh požarih oglasil v 10^6 sekundah ali prej.

Izhodna datoteka: za vsak požar izpiši v svoji vrstici število, ki pove, po najmanj koliko sekundah gori vsaj k pisarn.

Primer vhodne datoteke:

```
4
5 3 4 2 10
5 3 4 2 11
4 1 1 1 4
5 4 2 2 12
```

Pripadajoča izhodna datoteka:

```
2
3
3
3
```

Naslednja slika se nanaša na zadnji požar v gornjem primeru vhodne datoteke ($w = 5$, $h = 4$, $x_0 = y_0 = 2$). Slika za vsako celico mreže kaže, koliko sekund po začetku požara se vžge. Vidimo lahko, da po dveh sekundah gori šele 11 celic, po treh pa že 16 celic, zato je pri $k = 12$ pravilni odgovor 3.

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

4. Številčenje (`stevilcenje.in`, `stevilcenje.out`)

Zapisati hočemo naravna števila od vključno a do vključno b (seveda v desetiškem zapisu, kot ponavadi). **Napiši program**, ki ugotovi, kolikokrat se v tem številčenju pojavi posamezna števka (od 0 do 9).

Primer: pri $a = 9$, $b = 12$ imamo števila $9, 10, 11, 12$; tu se torej števka 1 pojavi štirikrat, števke 0, 9 in 2 po enkrat, ostale števke pa nikoli.

Vhodna datoteka: vsebuje le eno vrstico, v njej pa sta dve celi števili, najprej a in nato b, ločeni z enim presledkom. Veljalo bo $1 \leq a \leq b \leq 2^{50}$. V 40 % testnih primerov bo veljalo tudi $b \leq 10^6$.

Izhodna datoteka: vanjo izpiši 10 celih števil, vsako v svojo vrstico; ta števila naj po vrsti za vsako od števk od 0 do 9 povedo, kolikokrat se pojavlja v desetiškem zapisu števil od a do b.

Primer vhodne datoteke:

```
9 12
```

Pripadajoča izhodna datoteka:

```
1
4
1
0
0
0
0
0
0
1
```

5. Natrpan urnik (`urnik.in`, `urnik.out`)

Na planetu K-72 v glavni provinci C-01 imajo zelo zanimiv šolski sistem. Njihovo šolsko leto traja zelo dolgo, tam do več milijonov dni; po drugi strani pa imajo le 5 predmetov. Posledica velikega števila dni v letu je tudi to, da imajo čez celo leto zelo veliko testov. VX-48 iz okrožja H-18 je pridna učenka, ki si vsakič, ko pri kakšnem predmetu izve za datum testa, le-tega vestno zapiše v koledar. Tako se njen letni urnik kar polni in polni, njeni skrbni starši pa jo pogosto sprašujejo, koliko testov nekega predmeta ima v določenem obdobju. **Napiši program**, ki ji bo pomagal odgovarjati na takšna vprašanja.

Vhodna datoteka vsebuje zaporedje vnosov v beležko in poizvedb (lahko so poljubno premešane, torej ni nujno, da pridejo vsi vpisi pred vsemi poizvedbami). Da vhodna datoteka ne bo predolga, so vpisi in poizvedbe združeni v skupine (bloke) in vsaka vrstica vhodne datoteke predstavlja bodisi skupino vnosov bodisi skupino poizvedb.

Prva vrstica vhodne datoteke vsebuje dve celi števili, b in D, ločeni s presledkom; b je število skupin, D pa število dni v šolskem letu. Sledi b vrstic, vsaka od njih pa predstavlja bodisi skupino vnosov v beležko bodisi skupino poizvedb.

Vrstica, ki predstavlja skupino vnosov v beležko, se začne z znakom `V`, sledijo pa mu števila n, p, d in s, ločena s presledkom. To pomeni, da gre za skupino n vnosov, vsi se nanašajo na teste pri predmetu p, datumi teh testov pa so d, $d + s$, $d + 2s$, …, $d + (n - 1)s$.

Vrstica, ki predstavlja skupino poizvedb, se začne z znakom `P`, sledijo pa mu števila n, p, d_1, d_2, s_1 in s_2, ločena s presledkom. To pomeni, da gre za skupino n poizvedb, pri čemer i-ta od teh poizvedb (za vsak i od 1 do n) sprašuje po tem, koliko testov predmeta p je od vključno dne $d_1 + (i-1)s_1$ do vključno dne $d_2 + (i-1)s_2$.

Vsi datumi, ki nastopajo v vnosih in poizvedbah, so od 1 do D; končni datum vsake poizvedbe je večji ali enak od začetnega datuma tiste poizvedbe; nikoli se ne zgodi, da bi na nek datum večkrat vnesli test istega predmeta.

Dolžina leta je $1 \leq D \leq 10^6$; skupno število skupin, torej b, je $1 \leq b \leq 10^6$; za skupno število vseh vnosov in poizvedb (recimo mu N; to je vsota n-jev po vseh b skupinah iz vhodne datoteke) pa velja $N \leq 5 \cdot 10^6$. Pri vsaki skupini velja $n \geq 1$, $1 \leq p \leq 5$, $s > 0$, $s_1 > 0$, $s_2 > 1$.

Pri 20 % testnih primerov bo veljalo $N \leq 1000$ in $D \leq 1000$; pri 40 % testnih primerov bo veljalo $N \leq 10^5$ in $D \leq 10^5$; pri 80 % testnih primerov bo veljalo $N \leq 10^6$.

Izhodna datoteka: za vsako poizvedbo iz vhodne datoteke izračunaj število testov predmeta p v obdobju, na katerega se nanaša poizvedba, in izpiši vsoto vseh teh števil. (Pri posamezni poizvedbi štejejo seveda le tisti testi, ki so bili do te poizvedbe že vnešeni v beležko, ne pa morebitni testi, ki bodo vnešeni šele kasneje.) Nasvet: ta vsota je lahko večja od 2^{32}, zato je koristno uporabiti 64-bitne celoštevilske tipe (npr. **long long** v C/C++, **long** v C#/javi, `int64` v pascalu).

Primer vhodne datoteke:

```
6 20
V 3 2 3 4
V 2 4 6 3
P 5 4 2 5 2 3
P 3 2 1 7 4 3
V 1 2 6 1
P 3 2 1 7 4 3
```

Pripadajoča izhodna datoteka:

```
14
```

Pojasnilo: prva skupina v vhodni datoteki vnese tri teste za predmet 2 in to na dneve 3, 7 in 11; druga skupina vnese dva testa za predmet 4 in to na dneva 6 in 9; tretja skupina vsebuje pet poizvedb za predmet 4, in to za obdobja $[2, 5]$, $[4, 8]$, $[6, 11]$, $[8, 14]$ in $[10, 17]$ (odgovori na te poizvedbe so po vrsti: 0, 1, 2, 1, 0); četrta skupina vsebuje tri poizvedbe za predmet 2, in to za obdobja $[1, 7]$, $[5, 10]$ in $[9, 13]$ (odgovori so po vrsti 2, 1, 1); peta vnese še en test za predmet 2, namreč na dan 6; šesta skupina pa vsebuje iste tri poizvedbe kot četrta skupina, le da so odgovori zdaj 3, 2, 1. Skupno je torej v naši vhodni datoteki enajst poizvedb, vsota odgovorov nanje pa je $0+1+2+1+0+2+1+1+3+2+1 = 14$; to je tudi končni rezultat, ki ga moramo izpisati v izhodno datoteko.

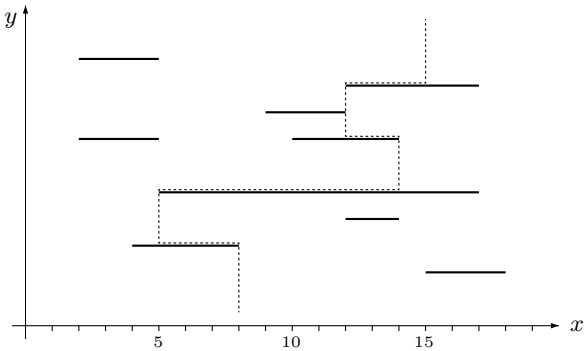
NALOGE ZA ŠOLSKO TEKMOVANJE

25. januarja 2013

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.

1. Lemingi

Naloga temelji na spremenjeni verziji starejše igre Lemingi. Pri tej verziji igre je igralno polje sestavljeno iz vodoravnih daljic, ki predstavljajo ploščadi, po katerih se lahko leming premika levo ali desno. Če stopi čez rob, pade naravnost navzdol na naslednjo ploščad, če ta obstaja. Ko leming prileti na neko ploščad, se mu smer gibanja glede na prejšnjo obrne. Leming se pri padcu ne poškoduje, ne glede na to, s kakšne višine je padel. Če leming pri padanju zgolj oplazi krajišče ploščadi, se to ne šteje za padec na to ploščad. Ploščadi se med seboj ne prekrivajo in ne dotikajo. Lemingi so manjši od najmanjšega možnega višinskega razmaka med ploščadmi, torej se v ploščad ne morejo zadeti od strani.



Opiši postopek, ki glede na dano igralno polje ter dano začetno x-koordinato leminga in smer gibanja (leming vedno začne na takšni višini, da je nad vsemi ploščadmi) določi koordinato v x smeri, kjer leming nazadnje pade čez rob kakšne ploščadi (ko torej pod njim ne obstaja nobena druga ploščad več, na kateri bi pristal). Če pa leming ne pade na nobeno ploščad, naj tvoj postopek vrne kar začetno x-koordinato leminga.

Predpostavi, da so vhodni podatki že shranjeni v nekih spremenljivkah oz. tabelah: n je število ploščadi, x_i in y_i sta koordinati levega krajišča i-te ploščadi (za $i = 1, 2, \ldots, n$), d_i je dolžina i-te ploščadi, x_0 je začetna x-koordinata leminga, s_0 pa

njegova začetna smer („L“ za levo ali „D“ za desno). Pri tem so ploščadi oštevilčene od višjih proti nižjim (z drugimi besedami, velja $y_1 \geq y_2 \geq \ldots \geq y_n$).

Gornja slika kaže primer, pri čemer debele črte predstavljajo ploščadi (teh je $n = 9$), črtkana črta pa pot leminga, če je $x_0 = 15$ in $s_0 =$ `D` (začetna smer leminga je v desno; ko prvič prileti na ploščad, se mu smer spremeni v levo, zato se po tej ploščadi premika v levo in tako naprej). Vidimo lahko, da je njegova končna x-koordinata (torej rezultat, ki ga mora poiskati tvoj postopek) v tem primeru $x = 8$.

2. 3-d šah

Predstavljajmo si trodimenzionalno šahovnico, v kateri posamezna polja niso kvadratki, ampak kockice, pa tudi celotna šahovnica je kocka, ki jo sestavlja $8 \times 8 \times 8$ polj. V to šahovnico postavimo tri kraljice. **Napiši program**, ki prebere položaj vseh treh kraljic in izpiše število polj, ki jih istočasno napadajo vse tri kraljice. Položaj vsake kraljice je opisan v svoji vrstici, v njej pa so tri cela števila od 1 do 8, ki podajajo koordinate kraljice. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `kraljice.txt` (kar ti je lažje).

Pravila za to, katera polja napada posamezna kraljica, so podobna kot pri običajnem šahu v dveh dimenzijah. Vsaka kraljica napada:

- 3 linije (v smeri vsake osi)
- 3×2 ravninski diagonali (v vsaki od ravnin, na katerih leži polje s kraljico)
- 4 prostorske diagonale.

To, ali neka kraljica napada neko polje ali ne, je neodvisno od tega, ali med to kraljico in tistim poljem stoji še kakšna druga kraljica ali ne (kraljice torej ne blokirajo napadov druga druge). Kraljica napada tudi polje, na katerem stoji.

3. Brisanje parov

Dan je niz, ki ga sestavljajo male in velike črke. Če sta mala in ustrezna velika črka druga za drugo, ju zbrišemo; to ponavljamo, dokler se da. Tako na primer iz `abbBAacCAdCcaBb` dobimo `abAdCcaBb`.

Napiši podprogram Okrajsaj(niz), ki izračuna ustrezno spremenjen (okrajšan) niz. Vseeno je, ali tvoj podprogram vrne okrajšani niz kot funkcijsko vrednost (npr. s stavkom **return**) ali pa kar spremeni vhodni parameter niz tako, da le-ta ob vrnitvi iz podprograma vsebuje skrajšano različico niza.

Predpostavi, da vemo, da so v nizu le male in velike črke (in nobenih drugih znakov) in da sta na voljo podprograma JeMalaCrka(znak) (ki vrne **true**, če je znak mala črka, sicer pa **false**) in DajVeliko(znak) (če je znak mala črka, vrne ustrezno veliko črko, drugače pa vrže izjemo). Tvoj podprogram naj bo čim bolj učinkovit, da bo deloval hitro tudi za zelo dolge vhodne nize; rešitev, ki ima časovno zahtevnost $O(n^2)$ namesto $O(n)$ (če je n dolžina vhodnega niza) lahko dobi pri tej nalogi največ 12 točk od 20.

4. Ta5nik

Pri pisanju kratkih sporočil si včasih prihranimo nekaj tipkanja tako, da kakšen podniz znakov v besedi nadomestimo s krajšim nizom po kakšnem preprostem ustaljenem pravilu. Tako lahko na primer namesto „`stric`“ zapišemo „`s3c`“, namesto

„`tapetnik`“ pa „`ta5nik`“. V nekaterih nizih je mogoče napraviti celo več kot eno zamenjavo, na primer: `1kostra0en`, `3gonome3ja`, `cen35alen`, `pr15`, `u01`. Če se v nekem nizu več možnih zamenjav prekriva, se dogovorimo, da vedno uporabimo najbolj levo med njimi; tako na primer iz besede „`petrijevka`“ dobimo „`5rijevka`“ in ne „`pe3jevka`“.

Podan imamo naslednji seznam dovoljenih zamenjav:

`nič` → `0`	`ena` → `1`	`dve` → `2`	`tri` → `3`	`štiri` → `4`
`pet` → `5`	`šest` → `6`	`sedem` → `7`	`osem` → `8`	`devet` → `9`

Napiši podprogram `Ta5nik(s)`, ki v nizu `s` opravi vse možne zamenjave, kot to določa tabela možnih zamenjav, nato pa, če je bilo možno opraviti več kot eno zamenjavo, predelani niz izpiše, sicer pa ne izpiše ničesar.

Primer: pri klicu `Ta5nik("enakostraničen")` naj izpiše `1kostra0en`; če pa pokličemo `Ta5nik("tapetnik")`, naj ne izpiše ničesar.

5. Koalicije

V nekem parlamentu sedijo poslanci n različnih strank (vsak poslanec pripada natanko eni stranki), in sicer je iz i-te stranke natanko n_i poslancev (za $i = 1, 2, \ldots, n$). Radi bi sestavili čim večjo koalicijo (torej skupino strank, ki imajo skupaj čim več poslancev), vendar z naslednjo omejitvijo: dan je seznam parov strank, ki se med sabo ne morejo prenašati, in od vsakega takega para hočemo imeti v naši koaliciji natanko eno stranko (ne pa obeh ali nobene od njiju). **Opiši postopek**, ki na podlagi teh podatkov izračuna velikost (skupno število poslancev) največje možne koalicije, ki ustreza danim omejitvam (ali pa ugotovi, da taka koalicija sploh ne obstaja).

Primer: recimo, da imamo $n = 6$ strank s takšnim številom poslancev: $n_1 = 5$, $n_2 = 10$, $n_3 = 3$, $n_4 = 4$, $n_5 = 2$, $n_6 = 1$; in da so pari nasprotujočih si strank naslednji: $(1, 2)$, $(3, 2)$, $(2, 6)$ in $(4, 5)$. Potem se izkaže, da ima največja primerna koalicija 14 poslancev (sestavljata jo stranki 2 in 4).

NEUPORABLJENE NALOGE IZ LETA 2011

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 6. tekmovanjem ACM v znanju računalništva (leta 2011), 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. 83–150) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Primerjava IPjev

Pri protokolu IPv6 je IP-številka načeloma 128-bitno celo število, ki ga lahko krajše zapišemo po naslednjih pravilih. Zapis je sestavljen iz 8 skupin s po 4 šestnajstiškimi števkami (pri tem lahko uporabljamo velike in/ali male črke). Skupine so ločene z dvopičji, na primer:

```
2001:0db8:85a3:0000:0000:8a2e:0370:7334
```

Vodilne ničle znotraj vsake skupine lahko izpustimo:

```
2001:db8:85a3:0:0:8a2e:370:7334
```

Največ eno zaporedje skupin, ki vsebujejo same ničle, lahko izpustimo; mesto, kjer smo izpustili skupine, je označeno z „`::`“:

```
2001:db8:85a3::8a2e:370:7334
```

Skupino ničel lahko opustimo tudi na začetku ali na koncu niza. Tako na primer vsi trije naslednji nizi predstavljajo isto številko:

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

Prav tako naslednji štirje nizi predstavljajo isto številko (čeprav drugo kot prejšnji trije):

```
0:0:0:0:0:0:0:0      ::      0:0:0::      ::0
```

Napiši podprogram, ki dobi dva niza znakov, ki predstavljata IP-številki, in vrne **true**, če predstavljata isto IP-številko, sicer pa vrne **false**. Tvoj podprogram lahko predpostavi, da v vhodnih nizih ni napak (torej da ustrezata zgoraj opisanim pravilom).

2. Dvigalo

V večnadstropni stavbi je zaposlenih z oseb, ki se vsak dan držijo enakega urnika. Posameznik i zaključi z delom ob času t_i, v nadstropju n_i pritisne na gumb dvigala, s katerim se želi odpeljati v pritličje (nadstropje 0), in počaka nanj. V podjetju uporabljajo tovorno dvigalo, ki ima nosilnost večjo od teže vseh zaposlenih skupaj in potrebuje za premik med sosednjima nadstropjema s časovnih enot. Čas, ki ga oseba porabi za vstop ali izstop iz dvigala, je tako majhen, da ga lahko zanemariš.

Vratar želi čim prej zaključiti s svojim delovnikom, zato bi rad programiral dvigalo tako, da bo zadnja oseba odložena v pritličju ob najmanjšem možnem času. Takrat lahko namreč zaklene stavbo in tudi sam odide domov. Vratar se ne ozira na to, koliko časa bo moral posamezen delavec čakati na dvigalo. **Opiši postopek**, ki izračuna, kdaj lahko vratar zaključi s svojim delovnikom. Dobro tudi utemelji, zakaj je rezultat tvojega postopka pravilen.

Omejitve: $z \leq 100$, $n_i \leq 100$, $t_i \leq 100\,000$. Vratarjev program sme dvigalo začeti premikati ob poljubnem času in sme predpostaviti, da je dvigalo ob začetku izvajanja stalo v pritličju.

3. Številski sestavi

Peter se je že v vrtcu naučil seštevanja. Sedaj je v osnovni šoli, kjer se uči seštevati števila v različnih številskih sestavih. Ker ve, da si dober programer, te prosi, da mu **napišeš program**, s katerim bo preveril pravilnost svojih izračunov.

Vhodni podatki: vsaka vrstica vhodnih podatkov bo vsebovala dve števili, b in n, pri čemer je b številski sestav, v katerem je zapisano število n. Podatki se zaključijo z vrstico „`0 0`“. Pri sestavih z osnovo več kot 10 se poleg števk od 0 do 9 uporabljajo še male črke angleške abecede od `a` naprej („števka“ `a` ima torej vrednost 10, števka `z` ima vrednost 35 ipd.).

Izhodni podatki: v edini vrstici izhoda izpiši rezultat seštevanja v desetiškem sistemu.

Primer vhodne datoteke:	Razlaga vhoda:
`2 0101110`	46 v desetiškem sistemu
`5 0244`	74 v desetiškem sistemu
`10 32`	32 v desetiškem sistemu
`3 02112`	68 v desetiškem sistemu
`0 0`	označuje konec podatkov

Pripadajoča izhodna datoteka:

```
220
```

4. Puškomitraljez

Podan imamo seznam besed (vsaka beseda je v svoji vrstici). **Napiši program**, ki prebere besede in ugotovi, koliko znakov je dolga najdaljša beseda, za katero velja, da se posamezna črka pojavlja največ enkrat (z drugimi besedami: v besedi se nobena črka ne pojavi dvakrat ali več kot dvakrat). Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `besede.txt` (kar ti je lažje). V besedah se pojavljajo samo male črke angleške abecede.

5. Disemvowelling

Na nekaterih forumih in blogih moderatorji poleg brisanja uporabljajo še en način za zatiranje neželenih sporočil: iz sporočila pobrišejo vse samoglasnike, ostale znake pa pustijo pri miru. Tako je sporočilo še vedno mogoče razvozlati, vendar le s precej truda in ga zato večina uporabnikov preskoči.

Napiši podprogram PobrisiSamoglasnike(s), ki izpiše niz, kakršen nastane, če v s pobrišemo vse samoglasnike in jih premaknemo na konec niza (v poljubnem vrstnem redu).

Primer: iz „`ena dve tri`“ dobimo `n dv treaei` (ali pa `n dv traeei` in še nekaj drugih možnosti).

Težja različica naloge: dobimo besedilo brez samoglasnikov; na voljo je tudi slovar besed (s samoglasniki) z znanimi frekvencami (pogostostmi) besed, pa tudi velika zbirka besedil (s samoglasniki), iz katere lahko oceniš še druge frekvence, če hočeš (npr. pogostost parov besed ali kakšnih daljših fraz). Opiši, kako bi se lotil čim boljše rekonstrukcije pobrisanih samoglasnikov.

6. Kemijske formule

Podana je formula kemijske spojine; v njej nastopajo simboli elementov, številke in oklepaji (npr.: NaCl, C_2H_5OH, C_{60}, $Ca(NO_3)_2$, $Co_3(Fe(CN)_6)_2$). Kot vidimo že iz zadnjega od teh primerov, so lahko oklepaji tudi gnezdeni. **Napiši podprogram** IzpisiAtome(s), ki v nizu niz s prejme formulo spojine in izpiše, koliko atomov katerega elementa je v spojini. Izpiše naj jih v abecednem vrstnem redu njihovih simbolov.[3]

Primer: če pokličemo

```
IzpisiAtome("C2H5OH");
IzpisiAtome("H2O");
IzpisiAtome("C60");
IzpisiAtome("Co3(Fe(CN)6)2");
```

naj se izpiše:

```
C 2
H 6
O 1

H 2
O 1

C 60

C 12
Co 3
Fe 2
N 12
```

7. Branje luknjanega traku

V 5-bitni teleprinterski kodi je vsak znak (črke, številke, posebni znaki, presledek) predstavljen z določeno kombinacijo petih luknjic v enem stolpcu traku. Imamo kodirno tabelo, v kateri je za vsak znak iz nabora možnih znakov zapisana kombinacija lukenj, ki ta znak predstavlja.

Bralnik luknjanega traku z zapisom v taki 5-bitni teleprinterski kodi pošilja vsebino prebranih trakov na računalnik. Vsebina vsakega prebranega traku je v

[3] To je težja različica naloge, ki smo jo na tekmovanju leta 2011 uporabili kot 4. nalogo v prvi skupini. Tam formule niso imele oklepajev, simboli elementov pa so bili le enočrkovni.

računalniku zapisana v svoji datoteki in sicer tako, da je vsak stolpec luknjanega traku (torej vsak znak) predstavljen s petimi zaporednimi biti v datoteki. Luknja na traku je predstavljena z 1, ne-luknja pa z 0. Med zapisano vsebino stolpcev traku ni nobenega ločilnega znaka.

Na računalniku smo našli pokvarjeno datoteko, pri kateri ne moremo prebrati ne začetnega in ne končnega dela vsebine. Vsebina je berljiva šele od nekega mesta dalje, vendar ne vemo, ali je to sredi znaka ali ne. Ker je tudi konec neberljiv, je tudi ne moremo prebrati „nazaj“.

Napiši program, ki bo prebral berljivi del okvarjene datoteke in s pomočjo kodirne tabele izpisal vse možne kombinacije znakov. Predpostavi, da dobiš podatke na standardnem vhodu kot niz znakov `'0'` in `'1'`; ta niz je dolg največ 1000 znakov. Da bo naloga lažja, predpostavi tudi, da je že na voljo funkcija `Znak(x)`, ki za dano 5-bitno kodo znaka (celo število od 0 do 31) vrne pripadajoči znak.

8. Pari besed

Napiši program, ki v danem besedilu poišče vse pare besed (dve sosednji besedi), ki se v enakem vrstnem redu pojavijo v celotnem besedilu vsaj dvakrat. (Vrstni red besed v paru je torej pomemben; na primer, v besedilu „`ena dve ena dve`“ se pojavi par besed „`ena dve`“ dvakrat, par „`dve ena`“ pa le enkrat.)

Predpostavi, da je celotno besedilo podano v datoteki `besedilo.txt`; besede so sestavljene le iz malih črk angleške abecede; ločil, velikih črk in drugih znakov v besedilu ni. Predpostaviš lahko, da je vsaka beseda v svoji vrstici, ali pa (če ti je lažje), da je celotno besedilo v eni sami vrstici in so besede ločene s po enim presledkom. Predpostavi tudi, da besedilo prihaja iz nekega naravnega jezika in ima temu primerne statistične značilnosti. Tvoja rešitev naj bo učinkovita tudi za primere, ko je vhodno besedilo zelo dolgo.

Težja različica naloge: kaj pa, če nas namesto parov, ki se pojavijo vsaj dvakrat, zanimajo npr. pari, ki se pojavijo vsaj desetkrat? In kaj, če nas namesto parov zanimajo skupine treh ali štirih zaporednih besed? **Opiši postopek**, ki za dana n in k poišče v besedilu vse take n-grame (skupine n zaporednih besed), ki se pojavijo v besedilu vsaj k-krat.

9. Pravokotnik

Nek programer je v enem svojem programu imel ogromen seznam pravokotnikov (v ravnini; stranice pravokotnika niso nujno vzporedne koordinatnima osema). Ker je želel prihraniti nekaj prostora, je shranil samo tri oglišča pravokotnika, ker je vedel, da je četrto mogoče rekonstruirati iz ostalih treh — zdaj pa tega ne zna več. **Napiši podprogram**, ki dobi koordinate treh oglišč pravokotnika in vrne koordinate četrtega oglišča.

10. Pismo iz zapora

FBI je po dolgih letih iskanja in skrbno narčtovanih akcijah končno ujel Unabomberja (s pravim imenom dr. Theodore Kaczynski). Unabomber[4] je zdaj v zaporu,

[4]Ozadje zgodbe: `http://en.wikipedia.org/wiki/Ted_Kaczynski`

„zunaj“ pa ima številne privržence in podpornike, ki bi zanj naredili marsikaj. Kot vsak zapornik ima tudi on možnost pisanja pisem iz zapora. Ta pisma pa čuvarji skrbno pregledajo, zato ne smejo vsebovati nič spornega.

Unabomber je izjemno zvit. Svoje pravo sporočilo bo skril v čisto nedolžno pismo, tako da bo nalašč naredil napake v tekstu. Pazniki tega ne bodo sprevideli, ker bodisi mislijo, da je dr. Kaczynski butast, bodisi so sami butasti in napak sploh ne bodo opazili.

(*a*) **Napiši podprogram**, ki prejme dva niza in neko celo število a. Drugi niz bo skrivno sporočilo in bo vseboval samo črke (nobenih ločil in presledkov). Podprogram naj drugi niz „skrije“ v prvi niz, tako da črke skrivnega sporočila vpisuje v prvi niz. Da ne bo preveč sumljivo, prvih a znakov pusti nespremenjenih. Prav tako med dvema spremenjenima črkama izpusti (vsaj) a znakov. Funkcija sme spreminjati samo črke v prvem nizu, kar pomeni, da je včasih treba izpustiti več kot a znakov (toliko, da pridemo do prve črke). Prav tako mora funkcija črko zares spremeniti, tj. črke ne moremo zamenjati za isto črko. Podprogram naj izpiše vsebino pisma, če je bilo skrivno sporočilo mogoče skriti v pismo, sicer pa naj izpiše „`Sporocila ni mogoce skriti :-(`“. (Skrivno sporočilo je lahko tudi predolgo.)

(*b*) *Različica naloge:* Pri zamenjavi črk naj ohrani „velikost“ prvotne črke (torej če je bila prvotna črka velika, naj bo tudi zamenjana črka velika, če pa je bila prvotna črka mala, naj bo tudi zamenjana črka mala).

Primer 1:

- Nedolžno sporočilo: `Everything is OK. Guards are nice and friendly. Prison food is not bad at all.`
- Skrivno sporočilo: `strikeatdawn`
- $a = 5$

Podprogram naj izpiše: `Everyshing ts OK. ruards ire nike and erienday. Prtson fdod is aot baw at anl.`

Primer 2:

- Nedolžno sporočilo: `I'm having a great time here in Alcatraz.`
- Skrivno sporočilo: `killthejudgeandjury`
- $a = 2$

Podprogram mora izpisati: `Sporocila ni mogoce skriti :-(`

(*c*) Napiši podprogram za dešifriranje, ki kot parameter dobi število a in dva niza: prvi je pismo pred šifriranjem, drugi pa je niz, ki je nastal z vpisovanjem skritega sporočila v drugi niz. Tvoj podprogram naj izpiše skrito sporočilo.

(*d*) Ali lahko šifrirani niz dešifriraš tudi, če ne poznaš vsebine pisma pred šifriranjem? Kakšne predpostavke si moral pri tem narediti o prvotnem pismu in skrivnem sporočilu?

(*e*) Napiši podprogram, ki kot parameter dobi število a in niz, ki je nastal s šifriranjem, in izpiše vsa možna skrita sporočila, ki bi lahko pripeljala do tega niza (če bi primerno izbrali pismo pred šifriranjem).

11. Knjige

Mirko rad bere knjige. Doma ima dolgo knjižno polico, na kateri ima zložene svoje najljubše knjige. Knjižna polica ob straneh nima ograjic, zato se včasih knjige ob robu prevrnejo in padejo dol. Ugotovil je, da so debele knjige bolj „stabilne“ od tankih in se ne prevrnejo tako zlahka. Zato se je odločil, da bo knjige zložil na naslednji način: Najdebelejša knjiga gre skrajno desno, druga najdebeleša skrajno levo, tretja po vrsti gre spet na desno, tik poleg najdebelejše, itd. Na primer, če je knjig 12, dobimo takšen vrstni red:

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

(Knjige so označene s števili po vrsti od najdebelejše naprej.)

(a) **Napiši podprogram** `void Izpisi(int n)`, ki izpiše ureditev knjig na polici. Kot parameter dobi naravno število n, ki pove skupno število knjig na polici.

(b) Ali bi znal ta podprogram napisati tako, da ne bo potreboval tabele?

(c) Njegova cimra Cilka si često sposodi kakšno knjigo, potem pa je ne vrne nazaj na pravo mesto. To Mirka zelo jezi, saj mora knjige ponovno zložiti. Njegov postopek je naslednji: gre od najdebeješe proti najtanjši knjigi in gleda, če je na ustreznem mestu (tj. ustrezno oddaljena od levega oz. desnega roba). Če ni, jo prestavi na ustrezno mesto (torej vzame knjigo s police in jo nato vrine v zaporedje knjig tako, da bo na pravi oddaljenosti od levega in desnega roba). Koliko knjig mora prestaviti Mirko, če Cilka knjigo vzame knjigo x in jo vrne na skrajno desno?

(d) Včasih mu oče kupi kakšno novo knjigo in jo položi na polico na skrajno levo. Koliko knjig mora prestaviti Mirko (po postopku iz (c)), če je nova knjiga tanjša od i-te, pa debelejša od $(i+1)$-ve? (Če je $i = 0$, je nova knjiga najdebelejša.)

(e) Napiši podprogram `int KolikoPrestavljanj(int n, int polica[])`, ki kot parameter dobi začetni razpored n knjig in vrne število prestavljanj, ki jih mora Mirko izvesti, ko po postopku iz (c) ureja knjige. Pri tem sme tabelo `polica`, ki jo je dobil kot parameter, tudi spreminjati.

(f) Mirko je opazil, da so na desni strani knjige v povprečju bolj debele, zato se je odločil, da jih bo zlagal tako, da po dve in dve najdebelejši (razen prve) položi na isto stran:

2 3 6 7 10 11 14 15 17 16 13 12 9 8 5 4 1.

Reši nalogi (a) in (b) še za ta način razporejanja knjig na polico.

12. Transfuzije

Človeška kri se razlikuje po različnih faktorjih, ki jih imenujemo sistemi krvnih skupin. Najbolj pomembna sistema krvnih skupin sta AB0 in Rh D (faktor rhesus). V prvem sistemu poznamo štiri različne krvne skupine (A, B, AB in 0), v drugem sistemu pa Rh D-pozitivno in Rh D-negativno krvno skupino. Če oba sistema skombiniramo, dobimo osem tipov krvi (A+, A−, B+, B−, AB+, AB−, 0+ in 0−).

Krvne skupine so pomembne pri transfuzijah, saj morata biti krvna skupina prejemnika krvi in krvna skupina darovalca kompatibilni. Pri tem veljajo naslednja pravila: pacient s krvno skupino 0 ne more sprejeti krvi skupine A, B ali AB; pacient s krvno skupino A ne more sprejeti krvi skupine B in obratno; pacient z negativno

skupino ne more sprejeti krvi pozitivne skupine. Vse kombinacije prejemnika in darovalca, ki ne zapadejo pod eno od teh omejitev, pa so dopustne.

(*a*) **Napiši podprogram**, ki dobi podatke o zalogi vseh osmih tipov krvi v bolnišnici in o tipu krvi pacienta, ki potrebuje transfuzijo; podprogram naj izpiše, kateri tipi krvi pridejo v poštev za transfuzijo (torej morajo biti na zalogi in kompatibilni s prejemnikom). Podatke lahko predstaviš tako, kot se ti zdi najbolj prikladno (vendar to svojo odločitev opiši).

(*b*) Recimo, da se od osmih prej omenjenih tipov krvi omejimo le na štiri: 0−, A−, B− in AB−. **Opiši postopek**, ki dobi za vsak tip krvi podatek o tem, koliko krvi tega tipa je na zalogi (torej za koliko transfuzij) in koliko pacientov s krvjo tega tipa potrebuje transfuzijo; postopek naj ugotovi, kako razdeliti kri med paciente tako, da bo transfuzijo dobilo čim več pacientov, pri čemer mora seveda vsakdo dobiti kri, ki je kompatibilna z njegovo. (Opomba: pri celi nalogi predpostavimo, da vsak pacient potrebuje enako količino krvi.)

Razmisli, ali je tvoj postopek primeren tudi, če dovolimo vseh osem tipov krvi namesto le prvih štirih.

(*c*) Reši nalogo (*b*) za splošnejši primer: recimo, da imamo n tipov krvi in da za vsak par tipov (i, j) vemo, ali pacient s krvjo tipa i lahko prejme kri tipa j ali ne. **Opiši postopek**, ki dobi za vsak tip krvi podatke o številu pacientov tega tipa in količini razpoložljive krvi tega tipa, nato pa razdeli kri med paciente tako, da jih bo čim več dobilo transfuzijo (pri čemer vsako dobi kri, ki je kompatibilna z njegovo).

13. KvadMars

Astronomi so odkrili do sedaj neznan planet KvadMars. Ker znanstvenike zanima, kakšno je to čudno nebesno telo — planet je kockast in na njem je vse čudno, kockasto in kvadratno — so poslali odpravo na planet. Tebi so naročili, da razviješ algoritme za avtomatskega robota „KvadMars surveyor“ za raziskovanje in premikanje po površini planeta.

Površino planeta KvadMars si lahko predstavljaš kot veliko karirasto mrežo, v kateri so vsa polja enako veliki kvadrati. Nekatera polja so prosta in se robot po njih lahko premika, na drugih pa so ovire in so zato tista polja neprehodna. Polja ob zunanjih robovih mreže so neprehodna, tako da mreža pravzaprav tvori labirint, iz katerega robot ne more pobegniti. V labirintu se ne pojavljajo kvadrati iz 2×2 prostih polj (ali še kakšna večja prosta območja).

Robot stoji vedno v enem od prostih polj in je obrnjen v eno od štirih možnih smeri (gor, dol, levo, desno). Lahko se premika za celo število polj naprej in nazaj (glede na smer, v katero je trenutno obrnjen) in se obrača na mestu v korakih po 90 stopinj levo ali desno.

Sistem v robotu pozna sledeče ukaze:

- **int** Tipalo(), pove kako daleč pred robotom je ovira (neprehodno polje); na primer, če vrne 0, to pomeni, da je ovira tik pred robotom (na polju, ki meji na polje, na katerem trenutno stoji robot); če vrne -1, pomeni, da je ovira dlje kot 10 polj daleč;

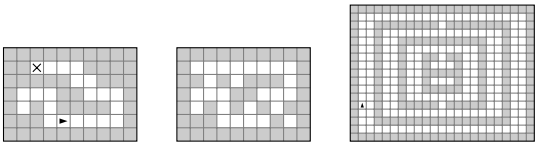
- **void** Naprej(**int** d), premakne robota naprej za d polj;

- **void** Nazaj(**int** d), premakne robota nazaj za d polj;
- **void** ObrniLevo() obrne robota za 90 stopinj v levo;
- **void** ObrniDesno() obrne robota za 90 stopinj v desno.

(*a*) Z uporabo navedenih funkcij opiši algoritem, ki bo samostojno prevozil labirinte na KvadMarsu do cilja in se vrnil na izhodišče — cilj označuje slepa ulica (kvadrat, ki ima na treh straneh steno) — ni poti ne levo, ne desno, ne naprej. Hkrati so astronomska opažanja potrdila, da se poti ne razvejujejo. Za primer glej levi labirint na spodnji sliki; če bi bil robot na začetku v položaju, ki ga označuje trikotnik (in obrnjen v desno), bi moral doseči polje, označeno z ×, in se od tam vrniti nazaj na začetni položaj.

(*b*) Kaj pa, če se lahko poti, ki jih tvorijo prosta polja v labirintu, tudi razvejujejo? (Še vedno pa ne tvorijo ciklov; iz enega prostega polja v drugo se dá torej priti po največ eni poti.) Opiši algoritem, s katerim robot obišče vsa prosta polja, ki so dosegljiva iz njegovega začetnega položaja, in se na koncu vrne nazaj na začetni položaj. Za primer glej srednji labirint na spodnji sliki; ne glede na to, na katerem prostem polju začne robot, bi moral obiskati vsa prosta polja razen tistega v spodnjem desnem kotu mreže.

(*c*) Kaj pa, če odpravimo vse omejitve razen tiste, da so polja na zunanjem robu mreže neprehodna? Opiši algoritem, s katerim robot tudi v tem primeru obišče vsa dosegljiva prosta polja in se vrne na začetni položaj. Za primer glej desni labirint na spodnji sliki; iz robotovega začetnega položaja so dosegljiva vsa prosta polja razen treh.



(*a*) (*b*) (*c*)

(*d*) Reši prejšnje naloge z dodatno omejitvijo, da ima robot zelo malo pomnilnika. Na primer, če je labirint velik $w \times h$, si robot ne more privoščiti tabele z $O(w)$ ali $O(h)$ elementi, kaj šele tabele z $O(w \cdot h)$ elementi.

(*e*) Ko je raketa pristala na KvadMarsu in se je robot izkrcal, nam je samodiagnostika sporočila, da se je menjalnik zataknil v vzvratni prestavi, tako da ne moremo uporabljati ukaza Naprej. Ker se misija odvija na KvadMarsu, robota ne moremo fizično popravljati, lahko pa ga preprogramiramo — zato popravi algoritme iz prejšnjih točk tako, da bo robot še vedno opravil zahtevano pot in pri tem uporabil preostale delujoče ukaze.

14. Torta

Dana je torta, ki ima v tlorisu obliko pravokotnika širine w in višine h. Torto prerežemo z več rezi; vsak rez je ravna črta, ki se začne na zgornji stranici pravokotnika in konča na spodnji stranici. Rezi so izbrani tako, da se nikjer ne sekajo trije ali več v isti točki. **Opiši postopek**, ki izračuna število točk, v katerih se sekata po

dva reza. Vhodni podatki so w, h, število rezov n in za vsak rez še par števil u_i in v_i, ki povesta, da se i-ti rez začne (na zgornjem robu) na razdalji u_i od levega roba, konča (na spodnjem robu) pa se na razdalji v_i od levega roba.

15. Trgovina

V trgovini so na začetku leta pričeli s prodajajo n različnih tipov izdelkov. Za vsak tip izdelka poznamo:

- začetno število kosov izdelkov v trgovini,
- minimalno število kosov izdelkov,
- polnilno število kosov izdelkov.

Kupci v trgovini kupujejo izdelke. Naenkrat lahko kupijo poljubno število različnih tipov izdelkov, pri posameznem tipu izdelka pa le največ toliko kosov, kolikor jih je tisti hip v trgovini na voljo za konkretni tip izdelka.

Računalniški program za spremljanje stanja v trgovini dobiva iz blagajn podatke o nakupih izdelkov — tip kupljenega izdelka in število kupljenih kosov tega izdelka.

S prodajo se manjša število preostalih kosov, ki so v trgovini še na zalogi. Ko pri nekem tipu izdelkov pade število preostalih kosov na ali pod minimalno število kosov za ta tip izdelka, skladiščnik iz skladišča takoj dopolni zalogo v trgovini in sicer za toliko kosov, kot je predpisano polnilno število kosov za ta tip izdelka.

Napiši program, ki bo na osnovi danih začetnih podatkov ter podatkov o nakupih kupcev preko celega leta izračunal pričakovano inventurno stanje za vsak izdelek na koncu leta.

16. T9

Pisanje sporočil SMS na mobilnih telefonih je lahko precej zamudno opravilo. Za izpis večine črk je potrebnih tudi več pritiskov na pripadajočo tipko. Tu priskoči na pomoč tehnologija T9 (*Text on 9 keys*), ki s pomočjo slovarja besed zmanjša število potrebnih pritiskov tipk za izpis želene besede. V tej nalogi bomo imeli opravka s približkom te tehnologije.

Predpostavimo, da ima telefon standardno raporeditev črk po tipkah (1: `abc`, 2: `def`, ...). Pri običajnem načinu pisanja sporočil izpisujemo posamezno besedo po črkah, z uporabo T9 pa besedo pravzaprav le opisujemo. S prvim pritiskom določimo tipko, na kateri se nahaja prva črka naše besede, z drugim pritiskom opišemo drugo črko besede itd. Sistem ob vsakem pritisku tipke primerja trenutni opis besede z naborom besed v slovarju in čim je beseda enolično določena, jo avtomatsko izpiše. Za oznako konca besede se uporablja tipka „`#`“. Na tem mestu naj opozorimo, da s tem načinom ni vedno možno izpisati vseh besed. Če slovar vsebuje besedi „`ae`“ in „`cd`“, ne moremo izpisati nobene od njiju, saj obe opišemo z enakim zaporedjem pritiskov na tipke: `12#`.

Na svojem telefonu želiš zmanjšati obseg slovarja, ker večine izmed n besed v njem nikoli ne uporabljaš. Pri manjšem slovarju pa je potrebnih tudi manj pritiskov tipk, da enolično določiš besedo. Vsaka beseda v slovarju je sestavljena iz največ b črk in ima prirejeno oceno pogostosti uporabe. Iz slovarja odstrani nekaj besed,

da bo vsaka beseda iz slovarja enolično določena z največ k pritiski na tipke. Najdi rešitev z največjo vsoto ocen pogostosti. **Opiši postopek**, ki čim bolj učinkovito reši dano nalogo, in oceni njegovo časovno zahtevnost.

Omejitve: $n \leq 100\,000$, $k \leq 10$, $b \leq 100$.

17. Spirala

Napiši podprogram `Spirala(s)`, ki dani niz `s` izpiše v spirali (v smeri urinega kazalca); pri tem seveda predpostavimo, da so vse črke (in presledki, ločila ipd.) enako široke. Primer: niz „`danesjelepdanitd.`“ naj se izpiše takole:

```
pdan
eani
ldet
ejsd
   .
```

18. Histogram

Podan je „histogram“ — stolpičast relief; vsi stolpci so enako široki, podane pa so njihove višine. Stolpcev je n (velja $n \leq 10^6$), višina i-tega stolpca pa je a_i. V ta histogram bi radi postavili čim višji pravokotnik, ki mora biti širok toliko kot w stolpcev, noben del pravokotnika ne sme štrleti ven iz histograma, spodnji rob pravokotnika pa mora ležati na spodnjem robu histograma. **Opiši postopek**, ki poišče tak pravokotnik, pri tem pa naj bo čim bolj učinkovit, da bo deloval hitro tudi za velike n.

19. Pikavost

Mali zeleni možici iz vesolja so se odločili preobraziti človeštvo po svoji podobi. Za začetek so s svojimi vesoljskimi žarki sprožili pri nekaterih ljudeh mutacijo, zaradi katere se človeku naredijo zelene pike po telesu. Ta mutacija se prenaša s staršev na potomce in to po naslednjih pravilih: ženska je pikasta natanko tedaj, če je pikast vsaj eden od njenih staršev; moški pa je pikast natanko tedaj, če je pikasta njegova mati, ne glede na to, ali je oče tudi pikast ali ne.

Napiši program, ki prebere družinsko drevo in na podlagi podatkov o tem, katere osebe v drevesu so vesoljci prežarčili (in jih s tem naredili pikaste), ugotovi, kateri potomci teh oseb bodo pikasti in kateri ne.

Vhodni podatki: v prvi vrstici so štiri števila, m, z, t in p. Števili m in z povesta, da je v drevesu m moških (imenujmo jih $M_1, M_2, \ldots, M_m$) in z žensk (imenujmo jih $Z_1, Z_2, \ldots, Z_z$). Sledi t vrstic, vsaka od njih pa je oblike $s\ i\ j\ k$; znak s je lahko `M` ali `Z`, takšna vrstica pa nam pove, da ima oseba s_i očeta M_j in mater Z_k. Sledi še p vrstic, od katerih je vsaka oblike $s\ i$, pri čemer je znak s spet lahko `M` ali `Z`; taka vrstica nam pove, da so vesoljci prežarčili osebo s_i (ki je zato postala pikasta). Predpostaviš lahko, da povezave med starši in otroci ne tvorijo ciklov. Če za neko osebo ni vrstice, ki bi povedala njene prednike, lahko predpostaviš, da so bili ti predniki ljudje, ki jih ni v drevesu in ki niso bili pikasti.

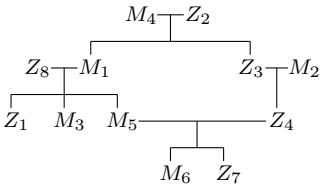
Izhodni podatki: program naj izpiše po eno vrstico za vsako pikasto osebo; ta vrstica naj bo oblike *s i*, pri čemer *s* pove spol te osebe (`M` ali `Z`), *i* pa njeno zaporedno številko.

Primer vhodne datoteke (za drevo na sliki):

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

Pripadajoča izhodna datoteka:

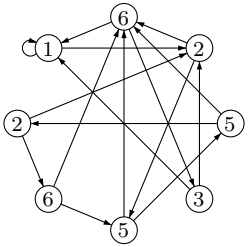
```
M 1
M 6
Z 1
Z 3
Z 4
Z 7
```



Razlaga primera: vesoljci prežarčijo M_1 in Z_3, ki zato postaneta pikasta. Z M_1 se prenese pikavost na hči Z_1, na sinova M_3 in M_5 pa ne (saj njuna mati, Z_8, ni pikasta). Z Z_3 se prenese pikavost na hčer Z_4, s te pa oba njena otroka, M_6 in Z_7. Tako so na koncu pikasti ljudje M_1, M_6, Z_1, Z_3, Z_4 in Z_7.

20. Neskončne stopnice

Imamo n polj, razporejenih v krog; na vsakem polju je napisano neko naravno število (glej primer na sliki). Začneš na označenem polju. V vsakem koraku lahko narediš dve različni potezi. Če označimo številko, na kateri trenutno stojiš, z m, se lahko premakneš za $m-1$ polj v pozitivni smeri (torej nasproti smeri urinega kazalca) ali za $m+1$ polj v negativni smeri (torej v smeri urinega kazalca). Obiskati moraš vsa polja, vsako samo enkrat, in končati na začetnem polju.



Napiši program, ki s standardnega vhoda prebere najprej število polj, nato pa (v naslednji vrstici) vrednosti številk na vseh poljih. Prva številka je hkrati začetno polje. Številke so navedene v smeri urinega kazalca.

Program naj izpiše premike, ki rešijo nalogo. Premike naj označi z „$-k$“ (premik za k polj v negativni smeri) ali s „$+k$“ (premik za k polj v pozitivni smeri).

Primer vhoda za nalogo s slike:

```
8
6 2 5 3 5 6 2 1
```

Pri takem vhodu mora program izpisati:

```
+5 -4 -2 -3 -6 +4 +1 +5
```

21. Najdaljši palindrom

Jaka potrebuje datoteko iz zašifriranega arhiva, pa je pozabil geslo. Spomni se samo, da je šlo za najdaljši palindrom iz Sonetnega venca. Pomagaj mu in **opiši postopek**, ki vrne najdaljši palindromni podniz v danem nizu znakov. (Štejejo samo strnjeni podnizi, nestrnjeni pa ne.) Tvoj postopek naj bo učinkovit, da bo deloval hitro tudi na nizih, dolgih več deset tisoč znakov.

22. Skakač na šahovnici

Skakač stoji na šahovskem polju (npr. A1). Na preostalem delu šahovnice določimo štiri poljubna polja (npr. C3, E5, A8, C8).

Napiši program, ki bo izračunal najmanjše število potez, v katerih lahko skakač iz svojega začetnega polja obišče vsa štiri določena polja. Svojo pot konča, ko prispe na zadnje od štirih določenih polj. Skakač lahko skače le po veljavnih skakačevih potezah.

23. Flomastri

V vsaki šoli z belimi šolskimi tablami prihaja do problema, da je učitelju v učilnici na voljo veliko flomastrov, a vsi skupaj vsebujejo tako malo barve, da mora učitelj na sredini ure hoditi k hišniku po nove. Zato ravnatelj od tebe, učitelja računalništva, pričakuje rešitev za ta problem.

Imaš podan seznam n flomastrov, ki jih ima hišnik vsak dan na zalogi. Vsak flomaster vsebuje eno izmed treh barv, označeno s številom od 1 do 3. Ker so pisala različnih proizvajalcev, ne vsebujejo vsi enake količine barve, prav tako pa se razlikuje tudi njihova cena.

Uporabi podatke o razpoložljivih flomastrih (i-ti flomaster ima barvo b_i, dolžino pisanja d_i in ceno c_i) in o porabi posamezne barve vsako šolsko uro ter izpiši seznam flomastrov, ki jih bodo učitelji ta dan potrebovali. Pri tem moraš za vse skupaj porabiti čim manj denarja.

(*a*) *Lažja različica naloge:* če nekega flomastra ne porabijo v celoti, ampak le delno, štejemo tudi njegovo ceno le delno, sorazmerno s tem, koliko črnila v njem so porabili. (Torej če npr. s flomastrom i zapišejo le $\lambda \cdot d_i$ enot dolžine za neko $\lambda \in (0, 1)$, naj se šteje, da ta flomaster k ceni izbora prispeva le $\lambda \cdot c_i$ denarnih enot.)

(*b*) *Težja različica:* ceno vsakega izbranega flomastra se v vsakem primeru upošteva v celoti (kot da bi morebitne delno izrabljene flomastre na koncu dneva vrgli v smeti).

Vhodni podatki: prva vrstica vsebuje število flomastrov n in število šolskih ur u. Sledi n vrstic, vsaka od njih pa vsebuje tri s presledkom ločena naravna števila b_i, d_i in c_i. Sledi u vrstic, vsaka od njih pa vsebuje tri s presledkom ločena naravna števila, ki povedo, kolikšno dolžino besedila je napisal profesor vsako uro. Podatki so za prvo, drugo in tretjo barvo, v tem vrstnem redu.

Izhodni podatki: izpiši indekse flomastrov, ki naj jih uporabijo ta dan. Indeks posameznega flomastra je številka vrstice v vhodnih podatkih, kjer je definiran (torej od 1 do n). Če rešitev ne obstaja, izpiši -1.

Primer vhodne datoteke:

```
5 3
1 7 5
3 12 1
1 4 3
2 10 1
2 20 15
3 2 1
8 0 3
0 10 5
```

Pripadajoča izhodna datoteka:

```
1 2 3 5
```

24. Zbiranje sličic

Imamo n otrok, ki zbirajo sličice; pri tem je A_i množica sličic, ki jih ima zbiratelj i. Zbiratelji so pripravljeni eden drugemu posoditi slike, da si jih drugi lahko fotokopira, vendar pa je i pripravljen posoditi j-ju neko sliko le, če tudi j posodi i-ju neko sliko, ki je i še nima. Zbiratelja i in j si torej lahko izmenjata največ $z_{ij} := \min\{|A_i - A_j|, |A_j - A_i|\}$ sličic (i posodi toliko sličic j-ju in j posodi prav toliko sličic, seveda ne enakih, i-ju). **Opiši postopek**, ki za vsakega i ugotovi, s katerim j bi si lahko izmenjal največ sličic. Predpostavi, da so sličice predstavljene s števili od 1 do s, pri čemer je $\cup_{i=1}^{n} A_i = \{1, \ldots, s\}$.

25. Črni kovčki

S potniškim letalom je na letališče Pručnik priletelo n agentov. Ob pristanku je vsak izmed njih zgrabil enega izmed črnih kovčkov s tajno vsebino, ki pa ni bil nujno njegov. Pomagaj jim in **opiši postopek**, ki ugotovi, povej, najmanj koliko menjav kovčkov je potrebnih, da bo vsak dobil svoj kovček.

Vhodni podatki: prva vrstica vsebuje število agentov n. Naslednjih n vrstic vsebuje število k, pri čemer i-ta vrstica vrstica pove, čigav kovček ima i-ti agent.

Izhodni podatki: izpiši najmanjše število menjav (pri čemer menjata kovčke med seboj dva agenta), ki jih potrebujemo, da dobi vsak agent svoj kovček.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
3
```

Težja različica naloge: izpiši primerno zaporedje menjav, ne le njihovega števila. Za vsako menjavo izpiši številki agentov, ki si pri tej menjavi izmenjata kovčka.

26. Tihotapec

Zapornik želi čim hitreje pobegniti iz labirinta, ki ima obliko kariraste mreže, sestavljene iz $w \times h$ kvadratnih celic. Celici sta *sosednji*, če imata skupno stranico. Sosednji celici sta bodisi ločeni z (neskončno tanko) steno bodisi povezani (torej v steni obstaja prehod). Zapornik ima na voljo naslednje premike:

- gre skozi prehod iz trenutne celice v sosednjo (če tak prehod obstaja), za kar porabi a sekund časa;
- previdno in potihem prevrta steno in se premakne skoznjo, za kar porabi b sekund;
- razbije steno in se premakne skoznjo, za kar porabi c sekund.

Težava pa je, da vsakič, ko na hitro razbije steno, postanejo pazniki bolj pozorni, zato sme to storiti največ p-krat. **Opiši postopek**, ki ugotovi, v kakšnem najkrajšem času lahko zapornik pobegne iz labirinta. Podana so števila a, b, c (veljalo bo $a \leq c \leq b$), p, začetni položaj zapornika in tudi celoten načrt labirinta (torej položaj vseh prehodov med celicami).

27. Vlaki

Na neki železniški postaji se ustavlja n vlakov, ki so oštevilčeni od 1 do n. Pri tem vlak i obišče postajo ob času a_i, nato spet ob času a_i+b_i, nato ob času a_i+2b_i in tako naprej. Na postaji stoji vandal, ki na vlake riše grafite. Grafit lahko nariše na vlak kadarkoli, ko ta vlak obišče postajo, vendar pa ima pri tem naslednjo omejitev: če postajo istočasno obišče več vlakov, lahko nariše grafit samo na enega od njih (lahko pa si sam izbere, katerega). Naš vandal bi rad v časovnem obdobju od trenutka 1 do trenutka T narisal grafite na čim več različnih vlakov. **Opiši postopek**, ki ugotovi, na koliko vlakov lahko pri teh omejitvah nariše grafite. (Vsi podatki o časih, torej a_i, b_i in T, so cela števila.)

28. Okvarjena ura

Imamo digitalno uro, ki kaže čas v obliki `HH:MM`, pri čemer je vsaka števka sestavljena iz 7 diod. V nekem dnevu smo m-krat pogledali na uro in si v vsakem od teh m trenutkov (ki so po vsaj 1 minuto narazen) zapisali stanje vseh 28 diod. Možno je, da so nekatere diode okvarjene (lahko tako, da so vedno prižgane, ali pa tako, da so vedno ugasnjene). **Opiši postopek**, ki iz danih m opažanj diod ugotovi, katere izmed 2^{28} kombinacij okvarjenih diod so konsistentne s temi opažanji.

29. Stolp iz kock

Na voljo imamo dva enaka kompleta n lesenih kvadrov, iz katerih želimo zgraditi čim višji stolp. Stolp mora biti stabilen, zato lahko kvader A z velikostjo osnovne ploskve $a_x \times a_y$ položimo na kvader B z osnovno ploskvijo $b_x \times b_y$ samo, če sta dimenziji

osnovne ploskve kvadra B strogo večji od dimenzij osnovne ploskve kvadra A (torej $a_x < b_x$ in $a_y < b_y$). Pri gradnji stolpa lahko kvadre poljubno orientiramo. **Opiši postopek**, ki izračuna, kako visok stolp lahko zgradimo iz vseh $2n$ razpoložljivih kvadrov. Veljalo bo $n \leq 4000$.

30. Ogled dirke

Na krožnem dirkališču tekmujeta dva avtomobila. Oba štartata istočasno in dirkata k krogov; prvi avtomobil vozi ves čas s konstantno hitrostjo v_1, drugi pa ves čas s konstantno hitrostjo v_2. Dolžina enega kroga je d; ob dirkališču na položajih $x_1, \ldots, x_n$ stojijo gledalci (gledalci so oštevilčeni po vrsti, tako da je $0 \leq x_1 < x_2 < \ldots < x_n \leq d$). Gledalca najbolj osreči, ko kateri od avtov pripelje mimo njega. **Opiši postopek**, ki poišče prvi enominutni interval, v katerem bo osrečeno maksimalno število gledalcev.

31. Paradižniki

V neki deželi imajo paradižnike (rastlina, ne sadež), ki zrastejo visoko v nebo, tudi po več kilometrov. Na taki rastlini zraste mnogo paradižnikov na različnih višinah. Hobit, ki rabuta vrtiček s paradižniki, bi si jih rad nabral kar največ. Ker mora plezati visoko po steblih, ne bo nujno uspel pokrasti vseh sadežev, saj bo prej omagal od plezanja. Skupno lahko prepleza h metrov navzgor, navzdol ni problema. **Opiši postopek**, ki ogotovi maksimalno število paradižnikov, ki jih lahko nabere.

Primer: denimo, da na neki rastlini rastejo paradižniki na višinah: 3, 7, 12, 15 in 20. Če se odloči splezati 12 metrov visoko, bo nabral 3 paradižnike in porabil 12 enot „energije“.

Opis vhodnih podatkov:

- h — količina „energije“ (koliko metrov lahko skupno prepleza navzgor)
- n — število rastlin, ki rastejo na vrtu
- a_i — število paradižnikov na i-ti rastlini
- $x_{i1}, x_{i2}, \ldots, x_{i,a_i}$ — višine, na katerih se nahajajo paradižniki na i-ti rastlini. Veljalo bo $x_{i1} \leq x_{i2} \leq \ldots \leq x_{i,a_i}$.

32. Binomski koeficienti

Spomnimo se, da je binomski koeficient $\binom{n}{k}$ definiran (za celi števili n in k, pri čemer je $0 \leq k \leq n$) kot

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

pri računanju teh koeficientov pa pride prav tudi zveza

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \text{za } k > 0.$$

Pri $k = 0$ pa je vedno $\binom{n}{0} = 1$.

Napiši funkcijo, ki dobi kot parametra števili n in k ter izračuna ostanek po deljenju $\binom{n}{k}$ s 6. Delovati mora učinkovito tudi v primerih, ko sta n in k velika (npr. n gre lahko do 10^{15}).

(*Opomba:* naslednja dva namiga naredita nalogo občutno lažjo, zato smo ju obrnili na glavo, da se jima bralec lažje izogne, če si noče pokvariti veselja ob reševanju naloge.)

Namig 1: spomnimo se, da lahko binomske koeficiente izpišemo v obliki trikotnika (za vsak n je ena vrstica, v njej pa so binomski koeficienti za ta n in za vse k od 0 do n), ki mu pravimo tudi Pascalov trikotnik. Za nekaj majhnih m si ogled, kaj se zgodi, če v Pascalovem trikotniku zamenjamo vsako število z njegovim ostankom po deljenju z m.

Namig 2: vzemimo dve praštevili p in q; za razna števila x izračunajmo ostanke po deljenju x s števili p, q in pq. Kakšna je zveza med temi ostanki?

REŠITVE NALOG ZA PRVO SKUPINO

1. Vandali

Iskanje podniza T_2 znotraj niza T_1 si lahko predstavljamo takole: najprej moramo poiskati kakšno pojavitev prve črke niza T_2 v T_1; nato moramo nekje kasneje v T_1 poiskati kakšno pojavitev druge črke niza T_2; in tako naprej. Spodnji podprogram se po nizu T_1 sprehaja z zanko **while**, medtem pa kazalec T2 kaže na tisti znak podniza, ki ga trenutno iščemo v T_1. Ko tak znak najdemo, ga izpišemo in se premaknemo naprej po T_2; namesto ostalih znakov niza T_1 pa izpisujemo #.

```
#include <stdio.h>

void Vandal(char *T1, char *T2)
{
  while (*T1)
  {
    if (*T1 == *T2) putchar(*T2++);
    else putchar('#');
    T1++;
  }
}
```

2. Kolera

Z gnezdenima zankama po x in y preglejmo vse točke in za vsako poiščimo najbližji vodnjak. V ta namen uporabimo še eno zanko, ki gre po vseh vodnjakih in za vsakega izračuna razdaljo do (x, y). To primerja z razdaljo do najbližjega doslej znanega vodnjaka (dNaj) in če je trenutni vodnjak še bližji, si ga zapomnimo (v spremenljivki vNaj). Na koncu te zanke vemo, kateri vodnjak je res najbližji, in povečamo za 1 njemu pripadajočo vrednost v tabeli povrsina. Na koncu se moramo le še sprehoditi z zanko po tej tabeli in jo izpisati.

```
#include <stdio.h>
#define N 50 /* velikost mreže */
#define StVodnjakov 10

int main()
{
  int povrsina[StVodnjakov], vx[StVodnjakov], vy[StVodnjakov];
  int x, y, d, v, dNaj, vNaj;

  /* Preberimo koordinate vodnjakov. */
  for (v = 0; v < StVodnjakov; v++) {
    scanf("%d %d", &vx[v], &vy[v]);
    povrsina[v] = 0; }

  /* Za vsako točko poiščimo najbližji vodnjak. */
  for (y = 1; y <= N; y++) for (x = 1; x <= N; x++)
  {
    for (v = 0; v < StVodnjakov; v++) {
      /* Izračunajmo razdaljo od (x, y) do v-tega vodnjaka. */
      d = (x - vx[v]) * (x - vx[v]) + (y - vy[v]) * (y - vy[v]);
      /* Če je bližji od doslej najbližjega vodnjaka, si ga zapomnimo. */
      if (v == 0 || d < dNaj) vNaj = v, dNaj = d; }
```

```
    povrsina[vNaj]++;
  }

  /* Izpišimo rezultate. */
  for (v = 0; v < StVodnjakov; v++) printf("%d\n", povrsina[v]);
  return 0;
}
```

V splošnem, če imamo s vodnjakov in mrežo velikosti $w \times h$ celic, je časovna zahtevnost tega postopka $O(s \cdot w \cdot h)$, ker moramo za vsako celico izračunati razdaljo do vseh s vodnjakov. Za majhno mrežo, kot smo jo imeli pri naši nalogi, je ta rešitev čisto dovolj dobra; vseeno pa si oglejmo še primer učinkovitejše rešitve. Pri njej bomo celice pregledovali od vodnjakov navzven, torej po naraščajoči oddaljenosti od najbližjega vodnjaka; zato takrat, ko prvič naletimo na neko celico, že tudi vemo, kateri vodnjak ji je najbližji, in nam ni treba računati še njene oddaljenosti do ostalih vodnjakov.

Kako lahko pregledamo celice po naraščajoči oddaljenosti od vodnjaka v? Recimo, da smo že pregledali vse celice, ki so od v oddaljene manj kot d enot. Ker oddaljenost merimo z evklidsko razdaljo, ima to območje približno obliko kroga. Naslednja celica, ki jo bomo morali pregledati, gotovo leži nekje na zunanjem robu tega kroga; med celicami na tem zunanjem robu moramo vzeti tisto, ki je najbližje vodnjaku v. Ta celica potem ne pripada več zunanjemu robu kroga, ampak postane del kroga, v zunanji rob pa morajo priti njene sosede (tiste, ki še niso bile pregledane). Tako se krog (pregledano območje) počasi povečuje in sčasoma pokrije celo mrežo. Postopek je torej takšen:

za vsako celico c naše mreže: $b[c] :=$ **false**;
$b[v] :=$ **true**; $H := \{v\}$;
while H ni prazna:
  naj bo c tista celica iz H, ki je najbližje vodnjaku v; pobriši c iz H;
  za vsako c-jevo sosedo c':
    if not $b[c']$:
      $b[c'] :=$ **true**; dodaj c' v H;

Množica H torej hrani celice na zunanjem robu pregledanega območja; to so tiste celice, ki jih še nismo pregledali, smo pa pregledali kakšno njihovo sosedo. Tabela b pa nam za vsako celico pove, ali smo jo že kdaj dodali v H ali še ne. Da bo postopek učinkovit, je množico H pametno predstaviti s kopico (*heap*), saj nam bo tako vsako dodajanje in brisanje elementa vzelo le $O(\log |H|)$ časa. (Mimogrede, gornji postopek si lahko predstavljamo kot malo prilagojeno različico Dijkstrovega algoritma za iskanje najkrajših poti po grafu.)

Ker imamo pri naši nalogi več vodnjakov, moramo takšno pregledovanje pognati iz vseh vodnjakov naenkrat; že pregledano območje zdaj ni več krog, ampak unija več takih krogov, množica H pa je zunanji rob te unije. Še vedno bomo, enako kot doslej, na vsakem koraku vzeli iz H tisto celico, ki je najbližja svojemu vodnjaku. Pomembno za učinkovitost postopka je, da vsako celico pregleda le tisti vodnjak, ki ji je najbližje; če za neko celico ugotovimo, da jo je že prej pregledal nek drug vodnjak, je ne pregledujemo še enkrat. V množici H je koristno namesto celic hraniti pare (c, v), ki nam povedo, da gre za celico c, do katere smo prišli pri pregledovanju z začetkom pri vodnjaku v.

```
1   za vsako celico c naše mreže: b[c] := NIL;
    H := {}; za vsak vodnjak v: b[v] := v; dodaj (v, v) v H;
2   while H ni prazna:
3     naj bo (c, v) tisti par iz H, pri katerem je razdalja med c in v najmanjša;
      pobriši ta par iz H;
4     if b[c] ≠ NIL then continue;
5     b[c] := v;
6     za vsako c-jevo sosedo c′:
7       if b[c′] = NIL:
8         dodaj (c′, v) v H;
```

Tabelo b zdaj uporabljamo za označevanje tega, kateri vodnjak je najbližji posamezni celici; to ugotovimo, ko iz H prvič vzamemo par, v katerem se pojavlja ta celica.

Kakšna je časovna zahtevnost tega postopka? Pogoj v vrstici 4 poskrbi za to, da se za vsako celico c izvedejo vrstice 5–8 le pri enem vodnjaku v. Notranja zanka v vrsticah 6–8 ima (pri vsakem c) največ štiri iteracije (ker ima vsaka celica v karirasti mreži največ štiri sosede), zato je skupno število dodajanj v množico H v vrstici 8 le $O(wh)$. Glavna zanka v vsaki iteraciji pobriše en element iz H, zato je tudi skupno število iteracij glavne zanke (in s tem skupno število izvajanj vrstic 2–4) le $O(wh)$. Tudi tokrat je pametno predstaviti H s kopico, tako da vsako dodajanje ali brisanje elementa porabi $O(\log |H|)$ časa; časovna zahtevnost celotnega postopka je zato $O(wh \log wh)$.

Faktor $O(\log wh)$ bi lahko teoretično še malo zmanjšali, če bi namesto običajne dvojiške kopice uporabili Fibonaccijevo. Še ena možnost za predstavitev množice H pa je, da se opremo na dejstvo, da je pri našem postopku razdalja med celico in vodnjakom vedno število oblike $\sqrt{\triangle x^2 + \triangle y^2}$, pri čemer je $-w \le \triangle x \le w$ in $-h \le \triangle y \le h$. Možne vrednosti $\triangle x^2 + \triangle y^2$ so torej od 0 do $w^2 + h^2$ in lahko bi imeli za vsako od njih po en seznam, v katerega bi vpisovali tiste elemente množice H, pri katerih je razdalja od celice do vodnjaka ravno koren te vrednosti. Za pregled teh seznamov bi porabili $O(w^2 + h^2)$ pomnilnika, tako da bi bila časovna zahtevnost našega postopka zdaj $O(wh + w^2 + h^2)$. To je lahko bolje od $O(wh \log wh)$, če je mreža približno kvadratne oblike (torej če sta w in h približno enaka).[5]

3. Kino

Za vsak film bi radi ugotovili, kateri je najzgodnejši avtobus, ki pripelje po koncu tega filma (če sploh obstaja kakšen tak avtobus); ko bomo vedeli to, bomo tudi zlahka izračunali, kako dolgo bi morali čakati nanj. Pri tem nam ni treba zaporedja avtobusov pregledovati vsakič od začetka, ampak lahko kar nadaljujemo pri tistem

[5] Problem, ki ga rešujemo pri tej nalogi, je pravzaprav diskretna različica Voronojevega diagrama, enega od najbolj znanih problemov v računski geometriji. V literaturi najdemo še razne druge ideje za njegovo reševanje; ena možnost je na primer, da iz vsakega vodnjaka širimo pregledovano območje v obliki kara ◇ namesto kroga, kar nam omogoča v praksi lažje doseči časovno zahtevnost $O(wh)$ (M. Velić *et al.*, *A fast robust algorithm for computing discrete Voronoi diagrams*, J. of Math. Modelling and Algorithms, 8(3):343–355, August 2009); žal pa za to rešitev obstajajo patološki primeri vhodnih podatkov, pri katerih približno polovico celic dodamo v H pri vseh vodnjakih, zato ima ta postopek v najslabšem primeru časovno zahtevnost $O(s \cdot wh)$. Obstajajo tudi ideje, kako naš prvotni (neučinkoviti) postopek z začetka te rešitve hitro izvajati na grafični kartici (K. Hoff III *et al.*, *Fast computation of generalized Voronoi diagrams using graphics hardware*, Proc. SIGGRAPH 1999, pp. 277–86).

avtobusu, kjer smo se ustavili pri prejšnjem filmu (saj je jasno, da tisti avtobusi, ki pridejo pred zaključkom k-tega filma, pridejo tudi pred zaključkom $(k+1)$-vega filma). Ko za nek film najdemo prvi naslednji avtobus, lahko izračunamo, kako dolgo bomo čakali nanj, najmanjši čas čakanja pa si zapomnimo (skupaj s filmom, pri katerem smo ga dosegli; spodnji postopek ima v ta namen spremenljivki $f^\star$ in $c^\star$).

Zapišimo naš postopek še s psevdokodo; čase prihodov avtobusov bomo označili z $p_1, p_2, \ldots, p_n$, trajanja filmov pa z $d_1, d_2, \ldots, d_k$. (Za lažje računanje s časi je koristno, če čase iz parov (ure, minute) sproti preračunavamo v minute.)

```
t := 0; a := 1; f* := 1;
for f := 1 to k:
  t := t + d_f;
  (* Film f se konča ob času t. Kateri je prvi naslednji avtobus? *)
  while a ≤ n:
    if p_a < t then a := a + 1 else break;
  if a ≤ n:
    (* Našli smo avtobus a; nanj bomo čakali t − p_a minut.
       Je to najmanjši čas čakanja doslej? *)
    if f = 1 or t − p_a < c*:
      f* := f; c* := t − p_a;
```

Na koncu tega postopka je v $f^\star$ število filmov, ki si jih moramo ogledati, da bomo morali na avtobus čakati najmanj časa.

4. Igličnи tiskalnik

Zahtevani izpis najlažje pripravimo po vrsticah. Posamezno vrstico izpisa dobimo tako, da gremo po vrsti po vseh znakih niza `s` in za vsak znak izpišemo ustrezno vrstico slike tega znaka (ki jo dobimo v tabeli `znaki`). Ostane nam še vprašanje, kako iz byta, s katerim je opisana vrstica znaka v tabeli `znaki`, ugotovimo, kateri piksli morajo biti prižgani, kateri pa so ugasnjeni. Še najlažje je, če uporabimo operatorje za delo z biti; bit 7 opisuje stanje najbolj levega piksla, bit 6 opisuje stanje drugega piksla z leve in tako naprej. Da preverimo, če je bit b prižgan, lahko na primer zamaknemo naš byte za b bitov navzdol (z operatorjem $\gg$) in nato preverimo, če v rezultatu prižgan bit 0 (po operaciji `& 1` nam ostane od števila le bit 0, torej moramo le še preveriti, če je ta rezultat ne ničeln).

```
#include <stdio.h>
unsigned char znaki[256][8];

void Izpisi(char *s)
{
  int y, x, i; unsigned char znak, vrsticaZnaka;
  for (y = 0; y < 8; y++)
  {
    /* Sprehodimo se po vseh znakih niza. */
    for (i = 0; s[i]; i++) {
      znak = (unsigned char) s[i];
      vrsticaZnaka = znaki[znak][y];
      /* V vrsticaZnaka je zdaj opis y-te vrstice znaka s[i]. */
```

```
        for (x = 0; x < 8; x++)
          if ((vrsticaZnaka >> (7 - x)) & 1) putchar('#'); else putchar('.'); }
      putchar('\n'); /* Izpišimo še znak za konec vrstice. */
    }
}
```

5. Dvigalo

Program v zanki tehta zaboje in jih dodaja v dvigalo; pri tem si v spremenljivki stZabojev zapomni, koliko zabojev je že naložil na dvigalo, v skupnaMasa pa vsoto njihovih mas. Skupno maso potrebujemo zato, da lahko preverimo, ali smemo naslednji zaboj še naložiti na dvigalo ali pa bi s tem že presegli nosilnost. Če naslednjega zaboja ne moremo več dodati na dvigalo, odpeljemo dvigalo v drugo nadstropje in raztovorimo vse zaboje (kličemo funkcijo Razlozi tolikokrat, kolikor je zabojev v dvigalu, torej stZabojev). Nato le še odpeljemo dvigalo nazaj v prvo nadstropje in že smo pripravljeni na natovarjanje novih zabojev. Opisani postopek je zato ovit še v eno (neskončno) zanko.

```
#include <stdbool.h>
#define Nosilnost 500

int main()
{
  int stZabojev = 0, skupnaMasa = 0, m;
  while (true)
  {
    /* Natovarjajmo zaboje, dokler se le da. */
    while (true)
    {
      m = Stehtaj();
      /* Naslednji zaboj ima maso m, ali ga lahko dodamo v dvigalo? */
      if (skupnaMasa + m > Nosilnost) break;
      /* Naložimo ga na dvigalo in si zapomnimo novo skupno maso. */
      Nalozi();
      stZabojev++; skupnaMasa += m;
    }
    /* Odpeljimo dvigalo v drugo nadstropje, ga raztovorimo in nato zapeljimo
       nazaj v prvo nadstropje. */
    OdpeljiDvigalo(2);
    while (stZabojev > 0) { Razlozi(); stZabojev--; }
    skupnaMasa = 0;
    OdpeljiDvigalo(1);
  }
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Binarni sef

Ker ima vsaka števka v kombinaciji dve možni vrednosti (0 in 1), obstaja skupno 2^n možnih n-mestnih kombinacij. Pri $n = 12$ je to na primer $2^{12} = 4096$; lahko si torej privoščimo tabelo 2^n logičnih vrednosti (**bool** oz. boolean), v kateri bomo za vsako možno n-mestno kombinacijo hranili podatek o tem, ali smo jo doslej v našem vhodnem nizu s že videli ali ne.

Ko se premikamo naprej po nizu s, lahko vse n-mestne kombinacije, ki se pojavljajo v njem kot podnizi, izračunamo takole: recimo, da smo trenutno na k-tem mestu v nizu s in da imamo trenutni podniz, $s[k-n+1]\ldots s[k]$, predstavljen v n-bitnem številu x (pri čemer bit b vsebuje vrednost števke $s[k-b]$). Če zdaj zamaknemo x za en bit v levo, bo dosedanji podniz $s[k-n+1]\ldots s[k]$ namesto v bitih od 0 do $n-1$ zapisan v bitih od 1 do n; če nato bit n ugasnemo, nam v bitih od 1 do $n-1$ ostane opis podniza $s[k-n+2]\ldots s[k]$; in če zdaj v bit 0 vpišemo vrednost $s[k+1]$, imamo zdaj v bitih od 0 do $n-1$ ravno opis podniza $s[k-n+2]\ldots s[k+1]$ — to pa je ravno naslednji podniz, ki nas bo zanimal (ko se premaknemo s k na $k+1$).

```
#include <stdbool.h>
#define MaxN 12

bool Sef(char *s, int n)
{
  bool prisotna[1 << MaxN];
  int x, i, stPrisotnih = 0;
  for (x = 0; x < (1 << n); x++) prisotna[x] = false;
  for (i = 0, x = 0; s[i]; i++)
  {
    x <<= 1; /* Zamaknimo x za eno mesto gor. */
    x &= ~(1 << n); /* Ugasnimo bit n. */
    if (s[i] == '1') x |= 1; /* Če je treba, prižgimo bit 0. */
    if (i < n - 1) continue; /* Mogoče še nimamo n bitov. */
    /* Označimo, da je kombinacija x prisotna. */
    if (! prisotna[x]) prisotna[x] = true, stPrisotnih++;
  }
  return stPrisotnih == (1 << n);
}
```

Razmislimo še o problemu, kako poiskati najkrajši niz bitov, ki vsebuje kot (strnjene) podnize vseh 2^n možnih zaporedij n bitov. Definirajmo graf, ki ima po eno točko za vsako možno zaporedje $n-1$ bitov (vseh točk je torej 2^{n-1}). Iz vsake točke naj gresta po dve usmerjeni povezavi, ena z oznako 0 in ena z oznako 1; iz točke $b_0b_1\ldots b_{n-1}$ nas povezava z oznako b pripelje v točko $b_1b_2\ldots b_{n-1}b$. Opazimo lahko, da vsakemu sprehodu po tem grafu ustreza neko (daljše) zaporedje bitov in obratno: če začnemo sprehod v točki $b_1b_2\ldots b_{n-1}$ in nato po vrsti sledimo povezavam z oznakami $b_n, b_{n+1}, \ldots, b_d$, lahko ta sprehod opišemo z nizom $b_1b_2\ldots b_d$ dolžine d; in po drugi strani, če si mislimo poljuben niz dolžine d, recimo $b_1b_2\ldots b_d$, vidimo, da za ta niz obstaja v našem grafu sprehod, ki se začne v točki $b_1b_2\ldots b_{n-1}$ in nato po vrsti sledi povezavam z oznakami $b_n, b_{n+1}, \ldots, b_d$.

Opazimo tudi, da niz, ki opisuje naš sprehod, vsebuje podniz $c_1c_2\ldots c_n$ natanko v primeru, če sprehod nekoč uporabi povezavo, ki gre iz točke $c_1c_2\ldots c_{n-1}$ in ima oznako c_n. Naš niz torej vsebuje vseh 2^n možnih podnizov dolžine n natanko tedaj, če pripadajoči sprehod uporabi v grafu vsako povezavo vsaj enkrat. Najkrajši možni niz bi torej dobili, če bi se dalo sestaviti tak sprehod, ki uporabi vsako povezavo *natanko* enkrat (takemu sprehodu pravimo *Eulerjev sprehod*).[6] Izkaže se, da tak sprehod res obstaja; še več, obstaja celo tak sprehod, ki uporabi vsako povezavo natanko enkrat in se začne in konča v isti točki (*Eulerjev obhod*). To ne velja le za naš graf, ampak za vsak tak usmerjen graf, ki je krepko povezan in je v njem vhodna stopnja vsake točke enaka njeni izhodni stopnji.

O tem se lahko prepričamo takole: začnimo sprehod v poljubni točki u in na vsakem koraku sledimo poljubni izhodni povezavi, ki je še nismo uporabili. Prej ali slej se zgodi, da iz trenutne točke (recimo ji v) poti ne moremo nadaljevati, ker so vse izhodne povezave že uporabljene. Označimo število teh povezav z d; po predpostavki ima potem v tudi natanko d vhodnih povezav. Recimo, da v ni ista točka kot u; ker sprehoda nismo začeli v v, zdajle pa se nahajamo v v, to pomeni, da je število vstopov v točko v za eno večje od števila izstopov iz te točke. Ker smo uporabili že vseh d izhodnih povezav, to pomeni, da je število izstopov enako d, zato mora biti število vstopov enako $d+1$; toda to je nemogoče, saj ima v le d vhodnih povezav. Torej možnost $v \neq u$ odpade, kar pomeni, da se je naš sprehod končal v isti točki, kjer se je začel; naš sprehod je torej v resnici obhod.

Če s tem obhodom še nismo uporabili vseh povezav grafa, ga lahko dopolnimo takole. Oglejmo si poljubno povezavo, ki je še nismo uporabili, recimo $x \to y$. Če smo točko x na našem dosedanjem obhodu že obiskali, lahko razmišljamo takole: namesto v u lahko naš obhod začnemo v x, naredimo en cel obhod, pridemo nazaj v x in od tam nadaljujemo v y. Zdaj imamo daljši sprehod kot prej in ga lahko nadaljujemo po enakem postopku kot v prejšnjem odstavku. Če pa x na prvotnem obhodu še nismo obiskali, mora biti gotovo dosegljiv (ker je naš graf povezan) iz neke točke z, ki smo jo na prvotnem obhodu že obiskali. Vzemimo poljubno pot od z do x in poiščimo na njej zadnjo točko, ki je še na našem prvotnem obhodu; recimo, da je to točka w; naš sprehod lahko torej namesto pri u začnemo pri w, naredimo cel obhod in nato nadaljujemo od w do x, od tam v y in tako naprej.

Vidimo torej, da ko se naš sprehod ustavi (ker nima več prostih izhodnih povezav), je to zagotovo obhod; in če obstaja še kakšna neuporabljena povezava, lahko obhod podaljšamo v še daljši obhod. Tako prej ali slej zagotovo pridemo do obhoda, ki uporabi vse povezave grafa, prav takega pa smo iskali. □

Zdaj torej znamo poiskati Eulerjev obhod za vsak tak usmerjen graf, ki je krepko povezan in ima v njem vsaka točka vhodno stopnjo enako izhodni. Hitro lahko vidimo, da tem pogojem ustreza tudi graf, ki smo ga sestavili pri naši nalogi. Definirali smo ga tako, da ima vsaka točka izhodno stopnjo 2 (iz nje gresta dve povezavi); hitro pa se lahko prepričamo, da ima vsaka točka tudi vhodno stopnjo 2: v točko $b_1b_2\ldots b_{n-1}$ kažeta povezavi z oznako b_{n-1}, ki se začneta v točkah $b_0b_1\ldots b_{n-2}$ za $b_0 \in \{0,1\}$. Poleg tega je naš graf krepko povezan: iz poljubne točke $b_1b_2\ldots b_{n-1}$ je mogoče priti v poljubno drugo točko $c_1c_2\ldots c_{n-1}$, če ne drugače, pa z $n-1$ koraki,

[6] Dolžina takega niza bi bila torej 2^n+n-1 bitov; za več o tem gl. zaporedje A005245 v *The On-Line Encyclopedia of Integer Sequences* in tam navedeno literaturo.

ki po vrsti sledijo povezavam z oznakami c_1, c_2 in tako naprej.

Za primer si oglejmo nekaj primernih nizov za majhne n:

n	Dolžina $2^n + n - 1$	Primer najkrajšega niza, ki vsebuje kot podnize vsa možna zaporedja n bitov
1	2	01
2	5	00110
3	10	0001011100
4	19	0000100110101111000
5	36	000001000110010100111010110111110000

2. Sumljiva imenovanja

Nalogo lahko rešimo zelo elegantno, če na vsak niz kvalifikacij gledamo kot na 32-bitno celo število (kot pravi besedilo naloge, lahko celo predpostavimo, da imamo pri roki funkcijo vStevilo za pretvorbo nizov v števila). Recimo, da število z predstavlja zahtevane kvalifikacije, k pa kvalifikacije nekega kandidata. Bit b je torej v z prižgan, če je zahtevana kvalifikacija b, v k pa je ta bit prižgan, če kandidat kvalifikacijo b dejansko ima.

Če hočemo preveriti, ali ima kandidat vse zahtevane kvalifikacije, moramo torej preveriti, ali so v k prižgani vsi tisti biti, ki so prižgani v z (poleg njih sme biti seveda prižgan še kakšen od bitov, ki so v z ugasnjeni). Pri tem nam pride prav operator &: v rezultatu $z \,\&\, k$ so prižgani tisti biti, ki so prižgani tako v z kot v k. Torej, če je imel k prižgane vse bite, ki so bili prižgani tudi v z, potem je $z \,\&\, k$ kar enak z; s tem lahko preverimo, če je imel nek kandidat vse zahtevane kvalifikacije.

Na enak način lahko preverimo tudi, če ima nek drugi kandidat vse kvalifikacije, ki jih je imel sprejeti kandidat; če je $k' \,\&\, k$ enako k, potem vemo, da ima k' vse tiste kvalifikacije, ki jih ima k; in če poleg tega k' in k nista enaka, potem vemo, da ima k' še kakšno kvalifikacijo, ki je k nima (tako lahko odkrivamo sumljive zaposlitve).

```
#include <stdio.h>

int main()
{
  char s[34]; unsigned int z, k, kk;

  gets(s); z = vStevilo(s); /* Preberimo zahtevane kvalifikacije. */
  gets(s); k = vStevilo(s); /* Preberimo kvalifikacije sprejetega kandidata. */

  /* Preverimo, ali ima kandidat vse zahtevane kvalifikacije. */
  if ((z & k) != z) { printf("nezakonito\n"); return 0; }

  /* Preglejmo ostale kandidate. */
  while (gets(s)) {
    kk = vStevilo(s);
    /* Ali ima kk vse kvalifikacije, ki jih ima k? */
    if ((k & kk) == k)
      /* Ali ima še kakšno kvalifikacijo, ki je k nima? */
      if (kk != k) { printf("sumljivo\n"); return 0; }}
  /* Če smo prišli do sem, je z imenovanjem vse v redu. */
  printf("zakonito\n"); return 0;
}
```

3. Dekodiranje nizov

Opazimo lahko, da se začetek podniza, ki ga obračamo, po vsakem koraku premakne

za eno mesto v desno: pri k-tem obračanju smo imeli podniz, ki se je začel s k-tim znakom vhodnega niza. Naloga pravi, da pri dekodiranju kot vhodni podatek dobimo tudi seznam dolžin obrnjenega podniza pri vseh obračanjih; recimo, da so to dolžine $d_1, d_2, \ldots, d_m$. Iz tega torej vidimo, da je bilo obračanj m, torej se je moralo zadnje od teh obračanj — to je tisto, s katerim je kodirani niz dobil svojo končno podobo (iz katere ga mi zdaj skušamo dekodirati) — začeti z m-tim znakom niza. Vemo pa tudi, da je to obračanje zajelo podniz dolžine d_m, tako da zdaj točno vemo, za kateri podniz je šlo, in ga lahko obrnemo nazaj. Nadaljujemo lahko na enak način: predzadnje obračanje se je moralo začeti pri $(m-1)$-vem znaku in je zajelo podniz dolžine d_{m-1}, tako da lahko tudi tega obrnemo nazaj itd.

Zapišimo naš postopek še s psevdokodo:

vhod: niz $s = s_1 s_2 \ldots s_n$ in seznam dolžin $d_1, d_2, \ldots, d_m$;

for $k := m$ **downto** 1:
  (* *Obrni podniz* $s_k s_{k+1} \ldots s_{k+d_k-1}$. *)
  $i := k$; $j := k + d_k - 1$;
  while $i < j$:
    $t := s_i$; $s_i := s_j$; $s_j := t$; $i := i + 1$; $j := j - 1$;

4. Silhuete

Naloga pravi, da je x_c višina najvišje stolpnice v c-tem stolpcu; če poiščemo maksimum vrednosti x_c po vseh stolpcih (torej po vseh c od 1 do w), mora biti to potemtakem višina najvišje stolpnice sploh. Podobno mora biti tudi maksimum vrednosti y_r po vseh r (od 1 do h) višina najvišje stolpnice sploh. Če v naših vhodnih podatkih tadva maksimuma nista enaka, lahko takoj zaključimo, da taka postavitev ne obstaja.

Če pa sta maksimuma enaka, lahko razmišljamo takole. Stolpec, pri katerem x_c doseže svoj maksimum, označimo s C, podobno pa tudi vrstico, pri kateri y_r doseže svoj maksimum, označimo z R. Vemo torej, da velja $x_C = y_R$. Postavimo zdaj v vrstico R stolpnice z višinami $x_1, \ldots, x_w$ (torej $a_{Rc} = x_c$ za vsak c), v stolpec C pa stolpnice z višinami $y_1, \ldots, y_h$ (torej $a_{rC} = y_r$ za vsak r). Za stolpnico a_{RC} smo torej zahtevali, da mora biti visoka x_C in hkrati tudi y_R, kar pa na srečo ni problem, saj sta x_C in y_R enaka. Vse ostale stolpnice v mestu lahko zdaj postavimo na najmanjšo možno višino, torej 1.

Prepričajmo se, da tako dobljena postavitev ustreza našim omejitvam. V R-ti vrstici imamo zdaj stolpnice z višinami $x_1, \ldots, x_w$, najvišja je torej visoka x_C, kar pa je isto kot y_R; torej y_R res pove višino najvišje stolpnice v R-ti vrstici. V kateri koli drugi vrstici, recimo r-ti, pa imamo stolpnice z višino 1, razen tiste v C-tem stolpcu, ki je visoka y_r; torej nam y_r res pove višino najvišje stolpnice v r-ti vrstici. Na analogen način bi se lahko prepričali tudi, da je x_c (za vsak c) res višina najvišje stolpnice v c-tem stolpcu.

```
#include <stdio.h>

#define MaxW 1000
#define MaxH 1000

int main()
```

```
{
  int w, h, x[MaxW], y[MaxH], r, c, a, R, C;
  /* Preberimo vhodne podatke. */
  scanf("%d %d", &w, &h);
  for (c = 0; c < w; c++) scanf("%d", &x[c]);
  for (r = 0; r < h; r++) scanf("%d", &y[r]);
  /* Poiščimo maksimalno višino. */
  for (C = 0, c = 1; c < w; c++) if (x[c] > x[C]) C = c;
  for (R = 0, r = 1; r < h; r++) if (y[r] > y[R]) R = r;
  if (x[C] != y[R]) { printf("taka postavitev ne obstaja\n"); return 0; }
  /* Izpišimo primerno postavitev. */
  for (r = 0; r < h; r++)
    for (c = 0; c < w; c++)
      printf("%d%c", (r == R ? x[c] : c == C ? y[r] : 1), (c == w - 1 ? '\n' : ' '));
  return 0;
}
```

Sestavljanja razporeda se lahko lotimo tudi še kako drugače (in dobimo drugačen, vendar tudi veljaven razpored); lahko bi na primer vzeli $a_{rc} = \min\{x_c, y_r\}$.

5. Ribič

Recimo, da mrežo počasi premikamo od leve proti desni (z drugimi besedami: počasi povečujemo x, torej koordinato levega konca mreže) in opazujemo, pri katerih x se število ujetih rib spremeni. Vidimo lahko, da so spremembe možne le v primerih, ko se (za nek i) vrednost x poveča z x_i na $x_i + 1$ (takrat i-ta riba pade iz mreže) ali pa se x poveča z $x_j - d$ na $x_j - d + 1$ (takrat pride j-ta riba v mrežo). Pri vseh drugih premikih mreže ostane število ujetih rib nespremenjeno.

Ker imamo seznam koordinat rib že urejen naraščajoče, se lahko zdaj po njem sprehajamo z i in j in gledamo, katera je naslednja koordinata začetka mreže (torej x), pri kateri pride do spremembe v številu ujetih rib. Če je bila prejšnja sprememba pri x, naslednja pa nastopi pri x', to pomeni, da je za vse položaje mreže od vključno x do vključno $x' - 1$ (tu je torej $x' - x$ možnih položajev) ulov enak; če je to ravno ulov, ki ga iščemo (torej natanko k rib), potem vemo, da teh $x' - x$ ugodnih položajev šteje k našemu končnemu rezultatu (spodnji postopek ga računa v r).

$i := 1$; $j := 1$; $x := x_1 - d$; $u := 0$; $r := 0$; **while** $i \leq n$:
  (* *i je naslednja riba, ki bo padla iz mreže, j pa naslednja riba, ki bo prišla v mrežo. Kaj od tega dvojega se zgodi prej?* *)
  if $j \leq n$ **then** $x_\oplus := x_j - d + 1$ **else** $x_\oplus := \infty$;
  $x_\ominus := x_i + 1$;
  $x' := \min\{x_\oplus, x_\ominus\}$; (* *Tu nastopi naslednja sprememba.* *)
  (* *Od x do $x' - 1$ je $x' - x$ možnih položajev mreže, pri katerih ujamemo natanko u rib; je to iskani ulov?* *)
  if $u = k$ **then** $r := r + x' - x$;
  (* *Izračunajmo novi ulov in se premaknimo po seznamu rib.* *)
  $x := x'$;
  if $x = x_\oplus$ **then** $u := u + 1$; $j := j + 1$;
  if $x = x_\ominus$ **then** $u := u - 1$; $i := i + 1$;
return r;

REŠITVE NALOG ZA TRETJO SKUPINO

1. Moderna umetnost

Recimo, da morata biti v končni verziji slike dva stolpca, x_1 in x_3, iste barve, nekje med njima pa je še nek stolpec neke druge barve; recimo, da je to stolpec x_2, pri čemer seveda velja $x_1 < x_2 < x_3$. Ali je mogoče, da bi x_1 in x_3 dobila svojo končno podobo v isti potezi z valjem? Pa recimo, da je to res. Poteza, ki pobarva x_1 in x_3, seveda pobarva tudi vse stolpce vmes, med drugim x_2; ker pa bo moral ta na koncu biti druge barve kot x_1 in x_3, to pomeni, da bomo morali x_2 pobarvati še z neko kasnejšo potezo. Ker se je dalo x_1 in x_3 pobarvati v eni potezi, sta gotovo manj kot k enot narazen (torej $x_3 - x_1 < k$). Tista poteza, ki kasneje pobarva x_2 z njegovo pravo barvo, pa seveda pobarva nek blok k stolpcev, ki med drugim vsebuje tudi x_2. Ker leži x_2 nekje med x_1 in x_3 in ker je med x_1 in x_3 največ $k-2$ stolpcev, je nemogoče, da bi tista poteza, ki bo pobarvala x_2, v celoti ležala med x_1 in x_3 — če nočemo, da nam ta poteza pokvari barvo stolpca x_1, bo gotovo pokrila stolpec x_3 in obratno. Tako torej vidimo, da je nemogoče, da bi x_1 in x_3 dobila svojo končno podobo v isti potezi.

Sliko si lahko torej predstavljamo kot sestavljeno iz blokov, pri čemer je vsak blok strnjeno zaporedje stolpcev iste barve. Primer iz besedila naloge ima na primer tri bloke: prvi blok je širok tri stolpce in ima barvo 2, drugi blok je širok dva stolpca in ima barvo 6, tretji blok pa je širok dva stolpca in ima spet barvo 2. Označimo širine blokov z $w_1, w_2, \ldots, w_m$. Širina i-tega bloka je w_i, tako da potrebujemo vsaj $\lceil w_i/k \rceil$ različnih potez, da dobijo vsi stolpci tega bloka svojo končno barvo. V prejšnjem odstavku smo tudi ugotovili, da nobena poteza ne more dati končne barve stolpcem iz dveh ali več različnih blokov; skupno število potez, ki jih potrebujemo, je torej $\sum_{i=1}^{m} \lceil w_i/k \rceil$.

Takšno barvanje lahko izvedemo čisto sistematično: gremo po sliki od leve proti desni, pri vsakem stolpcu preverimo, če ima že pravo barvo, in če je nima, izvedemo novo potezo z začetkom pri tem stolpcu in z barvo, ki jo mora na koncu ta stolpec imeti. Pri tem nam sploh ni treba vzdrževati tabele s trenutnim stanjem slike; za tisto, kar je levo od našega trenutnega položaja, že tako ali tako vemo, da je vse pravilno pobarvano in teh delov slike ne bomo več spreminjali; tisto, kar je desno od našega trenutnega položaja, pa je bilo bodisi pobarvano v zadnji doslej narejeni potezi (moramo si le zapomniti, katere barve je bila in do katerega stolpca je segla; v spodnjem programu imamo zato spremenljivki barvaDo in xPrazno), desno od tam pa je vse še nepobarvano.

```
#include <stdio.h>

#define MaxN 1000000

int main()
{
  int nPotez, barvaDo, xPrazno, x, n, k, b;
  FILE *f = fopen("umetnost.in", "rt");
  fscanf(f, "%d %d", &n, &k);

  xPrazno = 0; nPotez = 0;
  for (x = 0; x < n; x++)
```

```
  {
    fscanf(f, "%d", &b);
    if (x < xPrazno && b == barvaDo)
      continue; /* stolpec x je že prave barve */
    nPotez++; barvaDo = b; xPrazno = x + k;
  }
  fclose(f);

  f = fopen("umetnost.out", "wt"); fprintf(f, "%d\n", nPotez); fclose(f); return 0;
}
```

2. Kompleksnost števil

Označimo s $f(n)$ najmanjše število enic, ki jih potrebujemo, da sestavimo izraz z vrednostjo n. Zadnja operacija, ki jo pri tem izračunu izvedemo, je bodisi seštevanje bodisi množenje. Če je seštevanje, mora imeti eden od seštevancev vrednost 1 (tako zahteva besedilo naloge), drugi pa torej vrednost $n-1$, da bo vsota potem n. Naš izraz bo torej oblike $1+E$, pri čemer mora imeti podizraz E vrednost $n-1$. Po definiciji funkcije f vemo, da za izraz z vrednostjo $n-1$ potrebujemo najmanj $f(n-1)$ enic, torej bo imel naš izraz oblike $1+E$ (z vrednostjo n) vsega skupaj $f(n-1)+1$ enic.

Druga možnost je, da je zadnja operacija v našem izrazu množenje; naš izraz je torej oblike $E \cdot E'$. Naj bo k vrednost podizraza E; k mora biti torej nek delitelj števila n, drugi podizraz E' pa mora imeti vrednost n/k. Vemo, da za izraz z vrednostjo k potrebujemo najmanj $f(k)$ enic, za izraz z vrednostjo n/k pa najmanj $f(n/k)$ enic. Naš izraz oblike $E \cdot E'$ (z vrednostjo n) bo imel torej $f(k)+f(n/k)$ enic. Število n ima lahko seveda precej deliteljev k, zato moramo med njimi izbrati tistega, pri katerem bo skupno število enic najmanjše. Lahko pa se omejimo na primere, ko je $k \le \sqrt{n}$, kajti če je $k > \sqrt{n}$, potem je $n/k \le \sqrt{n}$ in bi dobili isti izraz kot pri nekem manjšem k, le vrstni red faktorjev bi bil zamenjan (na primer $6 \cdot 2$ namesto $2 \cdot 6$).

Tako torej vidimo, da lahko funkcijo f računamo takole:

$$f(n) = \min\{1+f(n-1), \min\{f(d)+f(n/d) : 2 \le d \le \sqrt{n},\ d \text{ deli } n\}\}.$$

To formulo lahko precej neposredno predelamo v rekurzivni podprogram (funkcijo), ki računa f in pri tem kliče samega sebe, da pride do vrednosti $f(n-1)$, $f(d)$ in podobno.

```
int KolikoEnic(int n)
{
  int f, d, c;
  if (n == 1) return 1;
  f = KolikoEnic(n - 1) + 1; /* število enic v izrazu 1 + (n - 1) */
  for (d = 2; d * d <= n; d++) {
    if (n % d != 0) continue; /* mogoče d sploh ni delitelj n-ja */
    c = KolikoEnic(d) + KolikoEnic(n / d); /* število enic v izrazu d * (n / d) */
    if (c < f) f = c; }
  return f;
}
```

Slabost take rešitve je, da bi se funkcija pri takšnem izračunu večkrat klicala z enako vrednostjo argumenta n, tako da bi po nepotrebnem izgubljala čas z večkratnim

izračunom ene in iste vrednosti funkcije f. Časovna zahtevnost takšne funkcije bi bila eksponentna v odvisnosti od n in v sprejemljivo hitrem času bi lahko obdelali n-je velikosti nekaj 100, za $n = 1000$ pa bi bila že prepočasna. Pri testnih primerih z našega tekmovanja bi ta rešitev dobila 40 % točk.

Do učinkovite rešitve pridemo, če si vsako vrednost $f(n)$, čim jo izračunamo, zapomnimo v neki tabeli; če jo nekoč kasneje spet potrebujemo, jo poberemo iz tabele, namesto da bi jo računali še enkrat. Zdaj lahko vrednosti funkcije $f(n)$ računamo čisto sistematično od manjših n proti večjim:

```
void KolikoEnic2(int n, int *f)
{
  int d, c, m;
  f[0] = 0; f[1] = 1;
  for (m = 2; m <= n; m++) {
    f[m] = f[m - 1] + 1;
    for (d = 2; d * d <= m; d++) {
      if (m % d != 0) continue;
      c = f[d] + f[m / d];
      if (c < f[m]) f[m] = c; }}
}
```

Zdaj imamo pri vsakem m-ju $O(\sqrt{m})$ dela (zaradi notranje zanke po d), zunanja zanka po m pa gre do n, tako da je časovna zahtevnost celotnega postopka $O(n\sqrt{n})$. Besedilo naloge pravi, da v enem testnem primeru pravzaprav dobimo več n-jev, vendar so urejeni padajoče; torej, ko bomo z gornjim podprogramom izračunali $f(n)$ za prvega od teh n-jev, bodo v tabeli f že pripravljene tudi vrednosti funkcije f za vse možne manjše n-je, tako da moramo za vse nadaljnje n-je iz vhodne datoteke le pogledati v tabelo in izpisati ustrezen odgovor. Pri testnih primerih z našega tekmovanja bi takšna rešitev dovolj hitro rešila 80 % točk.

Videli smo, da imamo pri posameznem m-ju $O(\sqrt{m})$ dela z iskanjem deliteljev m-ja; marsikateri m ima precej manj kot toliko deliteljev, tako da naša rešitev porabi precej časa s preizkušanjem neobetavnih vrednosti d. Veliko lažje je, če problem obrnemo: namesto da bi se pri danem m spraševali, kateri d so njegovi delitelji, se lahko pri danem d vprašamo, kateri m so njegovi večkratniki; in tu je odgovor zelo enostaven: to so $2d, 3d, 4d$ in tako naprej. Tako pridemo do naslednje rešitve:

```
void KolikoEnic3(int n, int *f)
{
  int m, k, mk;

  /* Na začetku postavimo vse f[m] na neko veliko vrednost, ki jo bomo kasneje zmanjševali,
     ko bomo odkrivali boljše načine za izražavo števila m z enicami. */
  for (m = 0; m <= n; m++)
    f[m] = m; /* to je gotovo večje ali enako od prave vrednosti f(m) */

  for (m = 1; m <= n; m++) {
    /* Trenutno je v f[m] najmanjše število enic, ki jih potrebujemo,
       če hočemo m izraziti kot zmnožek d * (m/d).
       Poglejmo še, s koliko enicami ga lahko izrazimo kot vsoto 1 + (m - 1). */
    if (1 + f[m - 1] < f[m]) f[m] = 1 + f[m - 1];
    /* Zdaj imamo v f[m] pravo vrednost funkcije f(m). Izraz za število m lahko zdaj
       uporabimo kot enega od faktorjev v izrazih oblike m * k; poglejmo, če lahko na ta
       način izboljšamo kakšno od dosedanjih vrednosti f[m * k]. */
```

```
    for (k = 2, mk = m + m; k <= m && mk <= n; k++, mk += m)
      if (f[m] + f[k] < f[mk]) f[mk] = f[m] + f[k]; }
}
```

Pri notranji zanki smo šli s k le do m, saj za večje k še ne poznamo prave vrednosti $f(k)$; produkte, pri katerih je $k > m$, bomo obravnavali takrat, ko bomo z zunanjo zanko prišli do k.

Kakšna je časovna zahtevnost tega postopka? Notranja zanka (po k) izvede največ n/m iteracij, vsaka od teh iteracij pa nam vzame $O(1)$ časa. Zunanja zanka gre z m od 1 do n; skupno število iteracij notranje zanke je torej kvečjemu $n/1+n/2+n/3+\ldots+n/n = n\sum_{m=1}^{n}(1/m)$. Vsota $\sum_{m=1}^{n}(1/m)$ je v matematiki znana kot n-to harmonično število H_n in je približno enaka $\ln n$. Časovna zahtevnost našega postopka je torej $O(n \log n)$. S tem bomo lahko dovolj hitro rešili tudi največje testne primere.

Zapišimo še preostanek programa:

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

void KolikoEnic3(int n, int *f) { ... } /* glej zgoraj */

int main()
{
  FILE *g = fopen("kompleksnost.in", "rt"), *h = fopen("kompleksnost.out", "wt");
  int n, t, *f;
  fscanf(g, "%d", &t); /* število n-jev v tej vhodni datoteki */
  fscanf(g, "%d", &n); /* največji n v tej vhodni datoteki */

  /* Alocirajmo pomnilnik in izračunajmo f(1), ..., f(n). */
  f = (int *) malloc(sizeof(int) * (n + 1));
  KolikoEnic3(n, f);

  /* Izpišimo vse rezultate, po katerih spražuje vhodna datoteka. */
  fprintf(h, "%d\n", f[n]);
  while (--t > 0) { fscanf(g, "%d", &n); fprintf(h, "%d\n", f[n]); }

  /* Pospravimo za sabo. */
  free(f); fclose(g); fclose(h); return 0;
}
```

(a) Razmislimo zdaj še o različici naloge, pri kateri je dovoljeno seštevanje tudi v primerih, ko sta oba seštevanca večja od 1. Najmanjše število enic, ki jih potrebujemo za izraz z vrednostjo n pri tej novi različici naloge, bomo označili s $f'(n)$. Naš podprogram KolikoEnic2 lahko dopolnimo, da bo preizkusil vse možne načine, kako izraziti n kot vsoto dveh števil, podobno kot že zdaj preizkuša vse mogoče načine za izražavo n-ja kot produkta. Dodati bi torej morali takšno zanko:

```
    for (d = 2; d <= m - d; d++) {
      c = f[d] + f[m - d];
      if (c < f[m]) f[m] = c; }
```

Vidimo lahko, da ta zanka pri vsakem m-ju porabi $O(m)$ časa; in ker mora iti m do n, bo časovna zahtevnost te rešitve zdaj $O(n^2)$. Zato nam tudi izboljšava, ki smo jo uvedli v KolikoEnic3, ne pomaga kaj dosti (ta nam sicer zmanjša čas preverjanja

zmnožkov z $O(n\sqrt{n})$ na $O(n \log n)$, vendar program zdaj večino časa tako ali tako porabi za preverjanje vsot).[7]

Vsak aritmetični izraz, ki je ustrezal zahtevam prvotne naloge, seveda ustreza tudi zahtevam naše nove različice, zato je $f'(n) \le f(n)$. Zanimivo vprašanje je, koliko se ti dve funkciji sploh razlikujeta; z drugimi besedami, koliko manjše aritmetične izraze lahko dosežemo, če dovolimo seštevanje poljubnih vrednosti. Izkaže se, da pri majhnih n-jih ne pridobimo ničesar, torej je tam $f'(n) = f(n)$; najmanjši n, za katerega to ne velja, je $n = 353\,942\,783$, pri katerem ima najmanjši primerni aritmetični izraz obliko $n = 6 + (n - 6)$.[8]

(b) Če hočemo poleg množenja in seštevanja dovoliti tudi potenciranje, lahko pri vsakem m pregledamo njegove potence, podobno kot že zdaj pregledujemo njegove večkratnike:

```
if (m >= 2 && m <= sqrt(n))
  for (k = 2, mk = m * m; k <= m && mk <= n; k++, mk *= m)
    if (f[m] + f[k] < f[mk]) f[mk] = f[m] + f[k]; }
```

Težava pri tej zamisli pa je, da lahko računamo potence m^k le do $k = m$, saj za večje k še ne poznamo prave vrednosti $f(k)$. Pri večkratnikih to ni bil problem; zmnožek mk za $k > m$ smo obdelali kasneje, kot večkratnik k-ja (ko je m prišel do sedanje vrednosti k) namesto kot večkratnik m-ja. Potenciranje pa ni komutativno, zato potence m^k (za nek $k > m$) ne moremo kasneje obdelati kot k^m, saj ti dve vrednosti v splošnem nista enaki. Ta težava se sicer pojavlja le v začetku našega postopka: potence m^k nas zanimajo le, če so $\le n$; in ker je $m \ge 2$, je to mogoče le pri $k \le \log_2 n$. Čim je torej $m \ge \log_2 n$, nas omejitev $k \le m$ ne bo več prizadela, saj bo zanko že pred tem končal pogoj $m^k \le n$. Pri majhnih m lahko za manjkajoče potence poskrbimo tako, da pri vsakem m ne računamo le potenc oblike m^k, pač pa tudi potence oblike k^m (za $k < m$). To je sicer bolj zamudno, ker moramo računati vsako od začetka (medtem ko lahko m^{k+1} zelo preprosto izračunamo iz m^k, pa potence $(k + 1)^m$ ne moremo tako preprosto izračunati iz k^m), vendar v praksi to ne bo problem, saj moramo to početi le pri majhnih m (do $\log_2 n$).

Še ena možnost za obravnavanje potenc pa je, da pri vsakem m pregledamo vse možne izražave oblike $m = s^k$ (za celoštevilske s in k); pri vsakem od teh sta s in k manjša od m, torej zanju že poznamo pravo kompleksnost; lahko torej preverimo, če je $f[s] + f[k]$ manjša od dosedanje $f[m]$, in če je, vpišemo to vsoto v $f[m]$. Kako lahko učinkovito poiščemo primerne s in k? Za začetek lahko m razcepimo

[7] Opisano rešitev bi se dalo še izboljšati; glej zaporedje A005245 v *The On-Line Encyclopedia of Integer Sequences* in tam navedeno literaturo. Srinivas Vivek V. in Shankar B. R. (*Integer complexity: breaking the* $\Theta(n^2)$ *barrier*, World Academy of Science, Engineering and Technology, 17:690–691, May 2008) sta pokazala, da je dovolj, če se (za $n \ge 65$) pri seštevanju omejimo na primere, ko je eden od seštevancev manjši od $(n/2)(1-\sqrt{r_n})$ za $r_n = 1-4\cdot 3^{1/3}(n-1)^{\log_2 3}/n^2)^{1/2}$. S to omejitvijo se časovna zahtevnost postopka zmanjša na $O(n^{\log_2 3})$, kar je približno $O(n^{1,59})$).

[8] M. N. Fuller (2008) v OEIS A005245; gl. tudi V. Wang, *A counterexample to the prime conjecture of expressing numbers using just ones*, J. of Number Theory, 133(2):391–397, February 2013. J. Iraids *et al.* (*Integer complexity: experimental and analytical results*, 2012) so pregledali vse n do 10^{12} in ugotovili, da je med njimi le približno 20 000 takih, pri katerih je $f'(n) < f(n)$, pa še tam se je vedno dalo najti izraze (s $f'(n)$ enicami in vrednostjo n), pri katerih je bil manjši od obeh seštevancev pri vsakem seštevanju zelo majhen (namreč 1, 6, 8 ali 9). Izkaže se tudi, da pri vsakem n velja $3\log_3 n \le f'(n) \le f(n) \le 3\log_2 n$.

na prafaktorje: $m = \prod_{i=1}^{t} p_i^{d_i}$. To je torej k-ta potenca nekega celega števila s le v primeru, če so vse stopnje $d_1, \ldots, d_t$ večkratniki k-ja. Torej pridejo v poštev le takšni k, ki delijo $d_1, \ldots, d_t$; oz. še drugače, izračunamo lahko $d := \gcd(d_1, \ldots, d_t)$ in potem naštejemo vse njegove delitelje: to so ravno k-ji, ki jih moramo pregledati (pri vsakem k potem izračunamo $s = n^{1/k}$ in preverimo, ali je $f[k] + f[s]$ manjši od dosedanje $f[m]$).

Delitelje d-ja bi se dalo načeloma našteti tako, da bi tudi d faktorizirali, $d = \prod_{i=1}^{u} q_i^{c_i}$, in potem bi vedeli, da so njegovi delitelji natanko vsa tista števila, ki so oblike $\prod_{i=1}^{u} q_i^{b_i}$ za $0 \le b_i \le c_i$; toda ker je d skupni delitelj števil $d_1, \ldots, d_t$, ta pa so eksponenti v faktorizaciji števila m, ki je $\le n$, iz tega sledi, da so vsi $d_i \le \log_2 n$, zato je tudi $d \le \log_2 n$, torej je d majhen in lahko njegove delitelje poiščemo preprosto tako, da pregledamo v zanki vsa števila od 1 do d in preverimo, ali delijo d ali ne.

Poskusi kažejo, da je s pomočjo potenciranja mogoče skoraj vsak n izračunati z manj enicami kot brez njega. Najmanjše število, pri katerem s potenciranjem nekaj pridobimo, je $n = 8$, ki ga lahko zdaj sestavimo s petimi enicami ($8 = (1+1)^{1+1+1}$) namesto s šestimi ($8 = (1+1)(1+1)(1+1)$). Najmanjše število, pri katerem moramo uporabiti potenciranje vsaj dvakrat, da porabimo minimalno število enic, pa je $88 = (1+1)^{1+1+1}(1+1+(1+1+1)^{1+1})$.

(c) Recimo zdaj, da poleg seštevanja in množenja dovolimo tudi odštevanje. Naš dosedanji postopek je računal vrednosti $f(m)$ po naraščajočih m, kar je bilo koristno, saj se je dalo m izračunati le iz števil, ki so bila manjša od njega; če smo torej računali $f(m)$ šele takrat, ko smo že poznali $f(1), \ldots, f(m-1)$, smo imeli vse podatke, potrebne za izračun $f(m)$. Ko pa dovolimo odštevanje, to ne velja več: m lahko zdaj izračunamo kot $(m+k)-k$, pri čemer je zdaj mogoče, da je prvi od teh operandov večji od m; lahko sta celo oba večja od m; lahko pa bi bil prvi operand manjši od m, drugi pa negativen.

Zdaj je torej $f(m)$ v nekem smislu odvisen tudi od vrednosti $f(m+1), f(m+2), \ldots$ in ni ravno očitno, kdaj o teh večjih vrednostih vemo dovolj, da zares lahko izračunamo $f(m)$.[9] Nalogo bomo lažje rešili tako, da si problem obrnemo: namesto da se pri danem številu sprašujemo, kakšna je njegova kompleksnost, se raje pri dani kompleksnosti vprašajmo, katera števila imajo takšno kompleksnost. Označimo to z $A(c) = \{n \in \mathbb{Z} : f(n) = c\}$; to je torej množica vseh števil s kompleksnostjo c oz. še drugače, to je množica vseh števil, ki se jih dá izračunati s c enicami (in operacijami $+$, $-$, $\cdot$), ne pa z manj kot c enicami. Če nas torej zanima kompleksnost $f(n)$ za nek izbrani n, moramo poiskati tisti c, za katerega je $n \in A(c)$.

Množice $A(c)$ lahko elegantno računamo po naraščajočih c:

$A(1) := \{1\}$; $c := 1$;
while $n \neq A(c)$:
  $c := c + 1$; $A(c) := \{\}$;
  for $k := 1$ **to** $c - 1$:
    za vsak $x \in A(k)$ in vsak $y \in A(c-k)$:

[9] Natančneje povedano, če smo pri nalogi brez odštevanja imeli rekurzivno zvezo oblike $f(n) = \min\{\min\{f(k) + f(n-k) : 1 \le k < n\}, \min\{f(k) + f(n/k) : k|n\}\}$, bi morali zdaj dodati še en minimum, namreč $\min\{f(n+k) - f(k) : k \in \mathbb{Z}\}$, zaradi katerega se zdaj naša rekurzivna zveza pravzaprav prelevi v nekakšen neskončen sistem enačb.

dodaj v $A(c)$ števila $x+y$, $x-y$ in $x \cdot y$;
for $k := 1$ **to** $c-1$:
$A(c) := A(c) - A(k)$;

Ko računamo $A(c)$, se torej opremo na dejstvo, da je izraz s $c > 1$ enicami gotovo oblike $x \circ y$, pri čemer je $\circ$ eden od operatorjev $+$, $-$ ali $\cdot$, podizraza x in y pa imata manj kot c enic; recimo, da ima levi podizraz k enic (za nek $1 \le k < c$), desni pa ima potem pač preostalih $c-k$ enic. Ker že vemo, kakšna števila lahko sestavimo s k enicami (to je ravno množica $A(k)$) ali s $c-k$ enicami (to je množica $A(c-k)$), lahko z njihovo pomočjo tudi sestavimo vsa možna števila s c enicami. Na koncu še pobrišemo iz množice $A(c)$ vsa tista števila, ki jih je mogoče sestaviti z manj kot c enicami (torej tista, ki so že v eni od množic $A(1), \ldots, A(c-1)$).

Postopek se ustavi, ko opazimo, da se je v trenutni množici $A(c)$ znašlo tudi število n; takrat imamo torej v c ravno kompleksnost števila n.

Kot zanimivost omenimo, da je najmanjše naravno število, ki ga lahko po uvedbi odštevanja izračunamo z manj enicami kot prej, $n = 23$. Brez odštevanja zanj porabimo 11 enic: $23 = 1 + (1+1)(1+(1+1)(1+1+1+1+1))$; z odštevanjem pa je dovolj že 10 enic: $23 = (1+1)(1+1)(1+1)(1+1+1) - 1$. Pri večjih n se pojavijo tudi primeri, ko je prihranek enic zaradi odštevanja še večji.

3. Požar

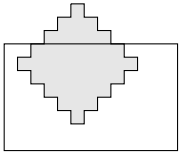
Oglejmo si najprej preprosto, vendar neučinkovito rešitev, ki bi bila dobra za majhne testne primere. Če se požar začne v (x_0, y_0), bo celica (x, y) zagorela $|x-x_0|+|y-y_0|$ sekund po začetku požara. V največ $w+h-2$ sekundah bodo torej zagorele vse celice (v tem času bi se na primer požar razširil iz enega vogala mreže v diagonalno nasprotni vogal). Lahko gremo torej po vseh celicah mreže in za vsako izračunamo, kdaj bo zagorela; v tabeli g si za vsak časovni trenutek vzdržujemo podatek o tem, koliko celic zagori v tistem trenutku. Nato se moramo le še sprehoditi po tej tabeli in seštevati števila v njej, dokler vsota ne preseže k; takrat vemo, da smo našli prvi časovni trenutek, v katerem gori t celic.

```
typedef long long int64;

int Pozar(int w, int h, int x0, int y0, int64 k)
{
  int x, y, t; int64 *g = (int64 *) malloc((w + h - 1) * sizeof(int64)), vsota;
  for (t = 0; t <= w + h - 2; t++) g[t] = 0;
  /* Preglejmo mrežo, za vsako celico izračunajmo, kdaj zagori,
     in povečajmo ustrezni element tabele g. */
  for (y = 1; y <= h; y++) for (x = 1; x <= w; x++)
    g[abs(x - x0) + abs(y - y0)]++;
  /* Seštejmo prvih nekaj elementov tabele g, dokler vsota ne doseže k. */
  for (t = 0, vsota = g[t]; vsota < k; ) vsota += g[++t];
  free(g); return t;
}
```

Na našem tekmovanju bi dobila ta rešitev 40 % točk; pri velikih mrežah je prepočasna, pa tudi preveč pomnilnika bi porabila, saj za tabelo g potrebujemo $O(w+h)$ prostora.

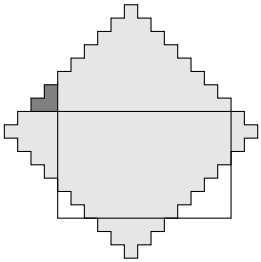
Razmislimo malo bolj geometrijsko. Če si izberemo nek čas t, koliko celic gori t sekund po začetku požara? Načeloma so to vse celice (x, y), za katere velja $|x-x_0|+|y-y_0| \leq t$. Na mreži tvorijo obliko kara $\diamond$ z oglišči $(x_0 \pm t, y)$ in $(x_0, y \pm t)$. Števila celic v tem karu ni težko izračunati: če jih seštevamo kar po vrsticah, vidimo, da jih je $1+3+5+\ldots+(2t-1)+(2t+1)+(2t-1)+\ldots+5+3+1$; to je skupaj $(t+1)^2+t^2$.

Ta karo se vsako sekundo malo poveča; stvari se zapletejo, ko trči ob robove mreže. Takrat nam formula iz prejšnjega odstavka daje prevelike rezultate, saj šteje tudi tiste dele kara, ki segajo čez rob mreže. Oglejmo si na primer spodnjo sliko, pri kateri del kara štrli nad zgornji rob mreže:



Vemo, da je zgornji vogal kara $(x_0, y_0 - t)$; najmanjša veljavna y-koordinata pa je 1. Torej nad zgornji rob mreže sega $d := 1 - (y_0 - t)$ vrstic našega kara. (Če je $d \leq 0$, to pomeni, da naš karo ne štrli čez zgornji rob.) Te vrstice tvorijo trikotnik s ploščino $1+3+5+\ldots+(2d-1) = d^2$; to moramo torej odšteti od ploščine našega kara. Podoben razmislek moramo opraviti še za ostale tri stranice mreže.

Še dodaten zaplet pa nastopi, ko se karo poveča do te mere, da nek del kara leži tako nad zgornjim robom kot levo od levega roba, na primer takole:



Temno osenčeno območje pripada tako trikotniku, ki štrli nad zgornji rob, kot trikotniku, ki štrli levo od levega roba. Ker smo doslej od ploščine kara odšteli ploščino obeh trikotnikov, to pomeni, da smo ploščino temnega območja neupravičeno odšteli dvakrat namesto samo enkrat. Zdaj jo moramo torej enkrat prišteti nazaj. Kolikšno je to območje? Celica v zgornjem levem kotu mreže je $(1, 1)$; prva celica našega temnega območja (zunaj meja mreže), ki jo požar doseže, je torej tista diagonalno stran od nje, to je $(0, 0)$. Ta celica zagori $|0 - x_0| + |0 - y_0|$ sekund po začetku požara; po tistem se požar širi še $d := t - (|0 - x_0| + |0 - y_0|)$ sekund. (Če je $d < 0$, pomeni, da požar temnega območja sploh ne doseže.) Temno območje je zdaj trikotnik s ploščino $1+2+3+\ldots+(d+1) = (d+1)(d+2)/2$. To moramo prišteti dosedanji ploščini našega gorečega območja. Podoben razmislek moramo opraviti

še za ostale tri vogale, razlika je le v tem, da namesto koordinat $(0, 0)$ vzamemo $(w + 1, 0)$, $(0, h + 1)$ in $(w + 1, h + 1)$.

```
int64 KolikoGori(int64 w, int64 h, int64 x0, int64 y0, int64 t)
{
  int64 a, d;
  a = (t + 1) * (t + 1) + t * t; /* karo */
  /* Odštejmo tisto, kar štrli čez robove. */
  d = 1 - (y0 - t); if (d > 0) a -= d * d;
  d = 1 - (x0 - t); if (d > 0) a -= d * d;
  d = (y0 + t) - h; if (d > 0) a -= d * d;
  d = (x0 + t) - w; if (d > 0) a -= d * d;
  /* Prištejmo tisto, kar smo neupravičeno odšteli dvakrat. */
  d = t - abs(0 - x0) - abs(0 - y0); if (d >= 0) a += (d + 1) * (d + 2) / 2;
  d = t - abs(w + 1 - x0) - abs(0 - y0); if (d >= 0) a += (d + 1) * (d + 2) / 2;
  d = t - abs(0 - x0) - abs(h + 1 - y0); if (d >= 0) a += (d + 1) * (d + 2) / 2;
  d = t - abs(w + 1 - x0) - abs(h + 1 - y0); if (d >= 0) a += (d + 1) * (d + 2) / 2;
  return a;
}
```

Vidimo torej, da smo gorečo površino pri danem času t izračunali v $O(1)$ časa. Zdaj bi načeloma lahko t postopoma povečevali, dokler goreča površina ne doseže k:

```
int Pozar2(int w, int h, int x0, int y0, int64 k)
{
  int t = 0;
  while (KolikoGori(w, h, x0, y0, t) < k) t++;
  return t;
}
```

To je že precej bolje od prejšnje rešitve; še vseeno pa se lahko zgodi, da je potrebni čas gorenja tja do $w + h - 2$ (npr. če se požar začne v enem od vogalov mreže in hočemo, da zagori celotna mreža). Časovna zahtevnost te rešitve je torej $O(w + h)$, kar je za največje testne primere še vedno prepočasi; bi pa dovolj hitro rešili tistih 60 % primerov, za katere besedilo naloge zagotavlja, da bo čas gorenja $\leq 10^6$ sekund.

Bolje pa je, če čas gorenja določamo z bisekcijo: 0 sekund je premalo (razen če je $k = 1$), $w + h$ sekund je gotovo dovolj, pravilna rešitev je nekje vmes; zdaj pa ta interval na vsakem koraku razpolovimo, dokler se ne zoži na eno samo sekundo; tisto je potem čas gorenja, ki ga iščemo.

```
int Pozar3(int w, int h, int x0, int y0, int64 k)
{
  int64 tL = -1, tD = w + h, t, g;
  while (tD - tL > 1) {
    /* Invarianta: tL sekund je premalo, tD sekund je dovolj. */
    t = (tL + tD) / 2;
    g = KolikoGori(w, h, x0, y0, t);
    if (g < k) tL = t; else tD = t; }
  /* Zdaj je tL = tD - 1; torej je tD - 1 sekund še premalo, tD pa že dovolj. */
  return (int) tD;
}
```

Zapišimo še preostanek rešitve:

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

int Pozar3(int w, int h, int x0, int y0, int64 k) { ... } /* glej zgoraj */

int main()
{
  int p, w, h, x0, y0; int64 k;
  FILE *f = fopen("pozar.in", "rt"), *g = fopen("pozar.out", "wt");
  fscanf(f, "%d", &p); /* preberimo število požarov */
  while (p-- > 0) { /* obdelajmo vse požare */
    fscanf(f, "%d %d %d %d %lld", &w, &h, &x0, &y0, &k);
    fprintf(g, "%d\n", Pozar3(w, h, x0, y0, k)); }
  fclose(f); fclose(g);
}
```

4. Številčenje

Naloga sprašuje po pogostosti števk v zapisu števil od a do b, vendar lahko to predelamo v malo preprostejši problem: kolikokrat se neka števka pojavi, če zapišemo vsa števila, manjša od n (torej od 1 do $n-1$)? Če znamo izračunati to za poljuben n, bomo morali le vzeti $n = b+1$ in $n = a$ ter izračunati razliko tako dobljenih pogostosti — to bodo potem ravno pogostosti v zapisu števil od a do b.

Zapišimo zdaj naš n v desetiškem zapisu: $n = n_{k-1}n_{k-2}\ldots n_2n_1n_0$, pri čemer je vodilna števka n_{k-1} seveda večja od 0 (sicer je sploh ne bi pisali). Zanima nas pogostost števk v številih, manjših od n. Ta števila lahko razdelimo na tista, ki so enako dolga kot n (torej imajo k) števk, in tista, ki so krajša (torej imajo t števk za nek t od 1 do $k-1$).

Vseh t-mestnih števil je seveda $9 \cdot 10^{t-1}$ (za prvo števko je 9 možnosti, ker mora biti različna od 0, za vsako od ostalih $t-1$ števk pa je 10 možnosti). Na spodnjih $t-1$ mestih je torej skupaj $9 \cdot 10^{t-1} \cdot (t-1)$ števk in ker se vse števke od 0 do 9 pojavljajo na teh mestih enako pogosto, odpade na vsako od njih ena desetina teh pojavitev, torej $9 \cdot 10^{t-2} \cdot (t-1)$. Poleg tega se vsaka števka razen 0 pojavlja še enkrat kot vodilna števka v 10^{t-1} številih. Opisani razmislek ponovimo za vse t od 1 do k in tako preštejemo pojavitve vseh števk v vseh številih, ki so krajša od n.

Razmislimo zdaj še o k-mestnih številih; tako število (recimo x) je enako dolgo kot n in se mogoče celo ujema z njim v nekaj vodilnih števkah, prej ali slej pa mora nastopiti neujemanje, kjer ima naš x manjšo števko od istoležne števke n-ja. Naj bo zdaj t indeks najvišje števke, kjer nastopi neujemanje med x in n. Koliko je pri tem t-ju možnih x-ov? Števka x_t mora biti manjša od n_t, zato pridejo zanjo v poštev vrednosti $0, 1, \ldots, n_t - 1$; rahlo poseben primer je $t = k$, ko je x_t vodilna števka, zato ne sme imeti vrednosti 0. Naj bo torej n'_t število možnih vrednosti števke x_t (torej $n'_t = n_t$ za $t < k$ in $n'_k = n_k - 1$). Števke levo od x_t morajo biti enake kot pri n, zato nimamo glede njih nobene izbire; števke desno od x_t pa so lahko poljubne, torej imamo zanje 10^t možnosti.

Tako torej vidimo, da je pri danem t možnih $n'_t \cdot 10^t$ primernih x-ov. Na spodnjih t mestih (desno od x_t) je vsega skupaj $n'_t \cdot 10^t \cdot t$ števk in ker se vse vrednosti pojavljajo tu enako pogosto, odpade na vsako od njih $n'_t \cdot 10^{t-1} \cdot t$ pojavitev. Na mestu t dobi vsaka števka od 0 do $n_t - 1$ (ali pa od 1 do $n_t - 1$, če je $t = k$) po 10^t pojavitev. Na mestih levo od t pa moramo le pogledati, kolikokrat se vsaka števka pojavi v tem

delu n-ja, in to pomnožiti z $n'_t \cdot 10^t$, saj so pri vseh x-ih (pri tem t) te števke enake kot pri n.

```
#include <stdio.h>

#define MaxStevk 17
typedef long long int64;

void Kolikokrat(int64 n, int64 *p)
{
  int ns[MaxStevk], pn[10], k = 0, t, d, nt; int64 pow10[MaxStevk];
  /* Inicializirajmo tabelo p. */
  for (d = 0; d <= 9; d++) p[d] = 0;
  if (n <= 0) return;
  /* Razbijmo n na števke. */
  while (n > 0) { ns[k++] = n % 10; n /= 10; }
  /* Izračunajmo si tabelo potenc števila 10. */
  for (pow10[0] = 1, t = 1; t < MaxStevk; t++) pow10[t] = 10 * pow10[t - 1];
  /* Preštejmo pojavitve števk v številih, krajših od n. */
  for (t = 1; t < k; t++) { /* Gledamo t-mestna števila. */
    /* Vodilne števke. */
    for (d = 1; d <= 9; d++) p[d] += pow10[t - 1];
    /* Števke na nižjih mestih. */
    if (t > 1) for (d = 0; d <= 9; d++) p[d] += 9 * pow10[t - 2] * (t - 1); }
  /* Preštejmo pojavitve števk v številih x, enako dolgih kot n. */
  for (d = 0; d <= 9; d++) pn[d] = 0;
  for (t = k - 1; t >= 0; t--) { /* t je najvišje mesto neujemanja med n in x. */
    nt = ns[t] - (t == k - 1 ? 1 : 0); /* možne vrednosti števke x_t */
    /* Vsi x se v mestih levo od t ujemajo z n in tam se števka d pojavi pn[d]-krat. */
    for (d = 0; d <= 9; d++) p[d] += pn[d] * nt * pow10[t];
    /* Prištejmo pojavitve na mestu x_t. */
    for (d = (t == k - 1 ? 1 : 0); d < ns[t]; d++) p[d] += pow10[t];
    /* Prištejmo še pojavitve na nižjih mestih. */
    if (t > 0) for (d = 0; d <= 9; d++) p[d] += nt * pow10[t - 1] * t;
    pn[ns[t]]++; }
}

int main()
{
  int64 a, b, pa[10], pb[10]; int d;
  FILE *f = fopen("stevilcenje.in", "rt");
  fscanf(f, "%lld %lld", &a, &b);
  fclose(f);
  Kolikokrat(a, pa); Kolikokrat(b + 1, pb);
  f = fopen("stevilcenje.out", "wt");
  for (d = 0; d <= 9; d++) fprintf(f, "%lld\n", pb[d] - pa[d]);
  fclose(f); return 0;
}
```

5. Natrpan urnik

Te naloge se lahko lotimo na veliko različnih načinov, od preprostejših in počasnejših

do učinkovitejših, vendar bolj zapletenih. Za vsak predmet moramo vzdrževati množico dni, v katerih nastopijo testi tistega predmeta; vsaka od teh množic je podmnožica množice $\{1, \ldots, D\}$, vsebuje pa največ N elementov; operaciji, ki ju moramo na tej množici podpirati, pa sta dodajanje novega elementa in poizvedba, pri kateri moramo prešteti, koliko elementov množice leži na intervalu $[d_1, d_2]$. Za število predmetov bomo uporabljali simbol P; pri naši nalogi je $P = 5$. Vse spodaj opisane rešitve dajejo pravilne rezultate in bi na našem tekmovanju dovolj hitro rešile vsaj 40 % testnih primerov.

(1) Zelo preprosta rešitev je, da množico predstavimo preprosto kot (neurejen) seznam, lahko v tabeli ali pa kot verigo, povezano s kazalci (*linked list*). Dodajanje je v tem primeru zelo preprosto in nam vzame le $O(1)$ časa, pri poizvedbi pa se moramo sprehoditi po celotnem seznamu in za vsak datum v njem preverjati, ali pripada intervalu $[d_1, d_2]$ ali ne; cena poizvedbe je zato $O(N)$. Tudi poraba pomnilnika je $O(N)$. Takšna rešitev je primerna za manjše testne primere, sploh če je poizvedb malo (dodajanj pa je lahko veliko).

(2) Še ena preprosta rešitev je, da imamo tabelo D logičnih vrednosti, ki nam za vsak dan povedo, ali je tisti dan kakšen test iz tega predmeta ali ne. Na začetku postavimo vse elemente tabele na **false**, pri vsakem dodajanju pa postavimo ustrezni element tabele na true. Dodajanje torej vzame le $O(1)$ časa. Za poizvedbo pa moramo pregledati del tabele od indeksa d_1 do indeksa d_2 in šteti, koliko elementov ima vrednost **true**; zato traja poizvedba $O(D)$ časa. Tudi poraba pomnilnika je zdaj $O(PD)$ namesto $O(N)$. Ta rešitev je torej slabša od prejšnje, če je testov veliko manj kot dni (testov gotovo ne more biti več kot dni, saj naloga pravi, da je na en dan lahko le en test posameznega predmeta). V praksi pa v naših testnih primerih število testov ni bilo tako majhno in rešitev s tabelo D logičnih vrednosti se ne obnese nič slabše od prejšnje.

(3) Malo boljša različica te rešitve je, da namesto tabele elementov tipa **bool** oz. boolean naredimo tabelo D bitov (oz. tabelo $\lceil D/8 \rceil$ bytov). Štetje testov je zdaj še hitrejše, malo zato, ker porabi tabela manj pomnilnika in zato bolje izkoristi procesorjev predpomnilnik, malo pa zato, ker si lahko za vsako možno vrednost enega byta vnaprej potabeliramo število prižganih bitov, zato med odgovarjanjem na poizvedbo lahko pregledamo po osem dni v enem koraku. Asimptotično sicer nismo ničesar pridobili, smo pa postopek pospešili za nek konstantni faktor, ki nam lahko omogoči rešiti kakšen testni primer več kot prej. Na našem tekmovanju bi dobra implementacija te rešitve lahko dosegla do 60 % točk.

(4) Namesto neurejenega seznama testov (za vsak predmet) imamo lahko seznam testov, urejen naraščajoče po datumu. Pri poizvedbi zdaj z bisekcijo poiščemo prvi test od dne d_1 naprej in prvi test po dnevu d_2; razlika med njima je zdaj ravno število testov v poizvedovalnem obdobju. Časovna zahtevnost poizvedbe je tako le $O(\log N)$. Po drugi strani pa je dodajanje zdaj drago: ko vrinemo nov test v urejen seznam, moramo kasnejše teste pomakniti za eno mesto naprej po tabeli, tako da dodajanje traja $O(N)$ časa. V primeru, ko pride med dvema poizvedbama več dodajanj, si lahko privoščimo majhno izboljšavo: elemente začasno dodajajmo v nek pomožni neurejen seznam; tik pred naslednjo poizvedbo pa ta seznam uredimo in z zlivanjem dodajmo njegove elemente v glavni urejeni seznam. Cena bloka b dodajanj je tako le $O(b \log b + N)$ namesto $O(bN)$. Tudi ta rešitev bi na našem

tekmovanju lahko dosegla do 60 % točk.

(5) Leto lahko razdelimo na približno $\sqrt{D}$ „mesecev“ s po $\sqrt{D}$ dnevi. Za vsak mesec (in vsak predmet) imejmo neurejen seznam testov tega predmeta v tem mesecu, poleg tega pa v neki tabeli hranimo tudi dolžino tega seznama. Dodajanje nam še vedno vzame le $O(1)$ časa, saj moramo dodati element v enega od seznamov in povečati podatek o njegovi dolžini v tabeli. Pri poizvedbi pa je zdaj tako: za tiste mesece, ki v celoti ležijo znotraj poizvedovalnega območja, moramo le sešteti število testov v njih (za vsak mesec preberemo število testov iz tabele); s tem je le $O(\sqrt{D})$ dela, saj je mesecev le $\sqrt{D}$. Ostaneta nam še mesec na začetku in na koncu poizvedovalnega območja; tadva mogoče ne ležita v celoti v našem poizvedovalnem območju, zato moramo iti po seznamu testov za vsakega od teh dveh mesecev in za vsak test preverjati, ali leži na našem poizvedovalnem območju ali ne. (Pri tem moramo paziti tudi na primere, ko celotno poizvedovalno območje leži znotraj enega samega meseca.) Ker ima vsak mesec le $\sqrt{D}$ dni in ker na vsak dan pride največ en test, sta tadva seznama dolga le $O(\sqrt{D})$. Tako torej vidimo, da je cena ene poizvedbe $O(\sqrt{D})$, poraba pomnilnika pa $O(N + p\sqrt{D})$. Na naših testnih primerih bi ta rešitev dobila 80 % točk.

(6) Leto lahko predstavimo s tabelo D elementov, ki za vsak dan povedo, ali je takrat test ali ne; nato razdelimo leto na $D/2$ dvodnevnih obdobij in imejmo tabelo $D/2$ elementov, ki za vsako tako dvodnevno obdobje povedo, koliko testov je v tem dvodnevnem obdobju; podobno naredimo še eno tabelo z $D/4$ elementi, eno z $D/8$ elementi in tako naprej. Vse te tabele skupaj imajo približno $2D$ elementov, tako da je poraba pomnilnika zdaj $O(PD)$, vendar z veliko večjim konstantnim faktorjem kot pri rešitvah (2) in (3). Pri dodajanju novega testa moramo povečati za 1 ustrezni element vsake od teh tabel; dodajanje zato traja $O(\log D)$ časa, ker je tudi tabel toliko.

Pri poizvedbi pa lahko razmišljamo takole: če nas zanima, koliko je testov v prvih d dnevih leta, lahko zapišemo d v dvojiškem zapisu, torej kot vsoto potenc števila 2: na primer $81 = 64 + 16 + 1$; to pomeni, da lahko vzamemo prvo 64-dnevno obdobje (iz tabele, ki se nanaša na 64-dnevna obdobja), ki pokriva dneve 1..64, nato vzamemo eno od 16-dnevnih obdobij (tisto, ki pokriva dneve 65..80; iz tabele, ki se nanaša na 16-dnevna obdobja) in končno eno enodnevno obdobje (namreč dan 81). Tako torej vidimo, da moramo iz vsake od naših $\log_2 D$ tabel uporabiti kvečjemu en element, da dobimo število dni v pripadajočem obdobju; nato jih moramo le še sešteti, pa dobimo število testov v prvih d dnevih. Če nas zanima število testov od dneva d_1 do d_2, lahko najprej izračunamo število testov v prvih d_2 dnevih in nato od njega odštejemo število testov v prvih $d_1 - 1$ dnevih.

Ta in vse nadaljnje rešitve že rešijo dovolj hitro vse testne primere z našega tekmovanja.

(7) Zelo elegantna rešitev je tudi Fenwickovo drevo; tu imamo za vsak predmet tabelo z D elementi, pri čemer element na indeksu k vsebuje število testov v obdobju od vključno dneva $f(k)+1$ do vključno dneva k. Pri tem je $f(k)$ število, ki ga dobimo, če v k ugasnemo najnižji prižgani bit. Pokazati je mogoče, da lahko s takšno tabelo tako dodajanja kot poizvedbe izvajamo v času $O(\log n)$. Prednost v primerjavi z rešitvijo (6) je v tem, da porabimo pol manj pomnilnika.

(8) Ena od slabosti prejšnjih dveh rešitev je, da je poraba pomnilnika sorazmerna

s številom dni v letu namesto s številom testov. Ker je lahko v letu največ N testov, ima vsaka od tabel iz rešitve (6) lahko največ N neničelnih elementov. Lahko bi jo torej predelali v razpršeno tabelo in hranili samo neničelne elemente; s tem se poraba pomnilnika zmanjša na $O(PN \log D)$, dodajanje in poizvedba pa sta še vedno možni v času $O(\log D)$, čeprav z večjim konstantnim faktorjem kot prej. Še en pogled na to rešitev je, da si družino tabel v (6) predstavljamo kot polno binarno drevo z $\log_2 D$ nivoji in D listi; če ga predstavimo eksplicitno kot drevo (namesto s tabelami), si ni težko predstavljati, da lahko iz njega porežemo prazna poddrevesa (taka, ki ne vsebujejo nobenega testa).

(9) Za predstavitev množice datumov lahko uporabimo kakšno od binarnih iskalnih dreves, pazimo le na to, da se nam ne bo izrodilo (uporabimo na primer rdeče-črno drevo ali pa AVL-drevo). V vsako vozlišče dodajmo še podatek o tem, koliko je vozlišč v celotnem poddrevesu z začetkom pri tem vozlišču. Tako lahko v času $O(\log N)$ ne le dodajamo in iščemo elemente, ampak tudi ugotovimo, koliko elementov leži na danem poizvedovalnem intervalu. Prednost v primerjavi s (6), (7) in (8) je, da porabimo zdaj le $O(N)$ pomnilnika, ne več $O(PD)$ ali $O(PN \log D)$.

Oglejmo si podrobneje rešitev (7), torej Fenwickovo drevo. To je podatkovna struktura, s katero lahko vzdržujemo zaporedje števil $a_1, a_2, \ldots, a_n$ tako, da lahko učinkovito (v $O(\log n)$ časa) izvajamo naslednji dve operaciji: spreminjanje posameznega elementa in računanje vsote prvih k elementov. Fenwickovo drevo je pravzaprav bolj tabela $T = [t_1, \ldots, t_n]$ kot drevo, pri čemer pa ta tabela ne vsebuje neposredno vrednosti a_i, pač pa delne vsote: element t_k vsebuje vsoto $a_{f(k)+1} + \ldots + a_k$, pri čemer je $f(k)$ število, ki ga dobimo, če k zapišemo v dvojiškem zapisu in ugasnemo najnižji prižgani bit.[10] Na primer: če vzamemo $k = 44$, je to v dvojiškem zapisu enako 101100_2; če ugasnemo najnižjo enico, dobimo 101000_2, kar je v desetiškem zapisu 40. Torej je $f(44) = 40$, kar pomeni, da v naši tabeli element t_{44} vsebuje vsoto $a_{41} + a_{42} + a_{43} + a_{44}$.

Kako lahko s pomočjo tabele T učinkovito izračunamo vsoto prvih k elementov, torej $s_k := a_1 + a_2 + \ldots + a_k$? V t_k najdemo zadnji del te vsote, namreč $a_{f(k)+1} + \ldots + a_k$; do s_k nam potem manjka še $a_1 + \ldots + a_{f(k)}$, kar je ravno $s_{f(k)}$. Torej je $s_k = t_k + s_{f(k)}$; s tem razmislekom lahko potem nadaljujemo pri $f(k)$, kjer je $s_{f(k)} = t_{f(k)} + s_{f(f(k))}$ in tako naprej. Prej ali slej pridemo do s_0, ki je po definiciji enak 0 in lahko pri njem naš razmislek končamo. Opazimo lahko, da ima vsak naslednji člen v zaporedju $k, f(k), f(f(k)), \ldots$ en prižgan bit manj kot prejšnji, zato ima ta zanka le $O(\log k)$ iteracij (saj ima k največ toliko prižganih bitov).

Oglejmo si spet primer $k = 44$. Zgoraj smo videli, da je $f(44) = 40$; v nadaljevanju se izkaže, da je $f(40) = f(101000_2) = 100000_2 = 32$ in nato $f(32) = 0$. Torej bi s_{44} izračunali kot $t_{44} + t_{40} + t_{32}$. (Ta vsota ima toliko členov, kolikor je prižganih

[10] Za več o Fenwickovem drevesu glej P. M. Fenwick, *A new data structure for cumulative frequency tables*, Software: Practice and Experience, 24(3):327–336, March 1994. Fenwickovo drevo je tudi v tesni povezavi z našo rešitvijo (6): tamkajšnji sistem tabel si lahko predstavljamo kot binarno drevo, v katerem vsako vozlišče hrani vsoto vseh listov v svojem poddrevesu; to tudi pomeni, da je vsako vozlišče vsota svojih dveh otrok, kar pomeni, da je ena od teh treh vrednosti (tista v staršu in v obeh otrocih) odveč, saj jo lahko izračunamo iz ostalih dveh; odpovejmo se na primer vrednosti v desnem otroku in vanjo preselimo vrednost iz staršа. Če ta postopek izvedemo sistematično od zgoraj navzdol, se nam notranja vozlišča popolnoma izpraznijo, najnižji nivo drevesa (tisti, v katerem so listi) pa se nam spremeni v točno tisto tabelo, ki jo uporablja Fenwickovo drevo.

bitov v dvojiškem zapisu števila k. To opažanje velja tudi v splošnem, za poljuben k.)

Izkaže se tudi, da je funkcijo $f(k)$ mogoče računati zelo preprosto in učinkovito s pomočjo operacij za delo z biti. Zapišimo k v dvojiškem zapisu kot $k = (k_{b-1} \ldots k_2 k_1 k_0)_2$, torej $k = \sum_{i=0}^{b-1} k_i 2^i$. Najnižjih nekaj bitov je mogoče ugasnjenih, prej ali slej pa mora biti nek bit prižgan; recimo, da je k_i najnižji prižgani bit. Torej je

$$\begin{aligned} k &= k_{b-1}\ k_{b-2} \ldots k_{i+1}\ 1\ 0\ 0 \ldots 0 \\ k-1 &= k_{b-1}\ k_{b-2} \ldots k_{i+1}\ 0\ 1\ 1 \ldots 1 \\ k \textbf{ and } (k-1) &= k_{b-1}\ k_{b-2} \ldots k_{i+1}\ 0\ 0\ 0 \ldots 0 \end{aligned}$$

Če primerjamo tretjo vrstico s prvo, vidimo, da se razlikujeta ravno v tem, da je najnižji prižgani bit prve vrstice v tretji vrstici ugasnjen. Torej je $f(k) = k$ **and** $(k-1)$. Še en način za računanje $f(k)$ pa je naslednji (črtica nad bitom pomeni njegovo negacijo, torej $\bar{x} = 1 - x$):

$$\begin{aligned} k &= k_{b-1}\ k_{b-2} \ldots k_{i+1}\ 1\ 0\ 0 \ldots 0 \\ \textbf{not } k &= \bar{k}_{b-1}\ \bar{k}_{b-2} \ldots \bar{k}_{i+1}\ 0\ 1\ 1 \ldots 1 \\ \textbf{not } k+1 &= \bar{k}_{b-1}\ \bar{k}_{b-2} \ldots \bar{k}_{i+1}\ 1\ 0\ 0 \ldots 0 \\ k \textbf{ and } (\textbf{not } k+1) &= \quad 0 \quad\ 0 \quad \ldots \quad 0 \quad 1\ 0\ 0 \ldots 0 \end{aligned}$$

Tako smo torej izolirali najnižji prižgani bit v k; zdaj lahko $f(k)$ dobimo tako, da od k odštejemo vrednost tega bita: $f(k) = k - k$ **and** $(\textbf{not } k+1)$.[11]

Zdaj moramo razmisliti še o tem, kako popraviti tabelo T, ko se ena od vrednosti a_r spremeni; recimo, da se poveča za $\triangle$. Katere vrednosti se zaradi tega spremenijo v tabeli T? Spomnimo se, da t_k vsebuje vsoto $a_{f(k)+1} + \ldots + a_k$, torej se t_k spremeni natanko tedaj, ko ta vsota vsebuje tudi člen a_r, to pa je pri $f(k) < r \leq k$. Poiskati moramo torej vse k-je, ki ustrezajo temu pogoju, in njihove t_k povečati za $\triangle$. Oglejmo si še enkrat dvojiški zapis števil k in $f(k)$ ter razmislimo, kakšen mora biti r, da leži med njima:

$$\begin{aligned} k &= k_{b-1}\ k_{b-2} \ldots k_{i+1}\ 1 \quad 0 \quad\ 0 \quad \ldots 0 \\ r &= k_{b-1}\ k_{b-2} \ldots k_{i+1}\ r_i\ r_{i-1}\ r_{i-2} \ldots r_0 \\ f(k) &= k_{b-1}\ k_{b-2} \ldots k_{i+1}\ 0 \quad 0 \quad\ 0 \quad \ldots 0 \end{aligned}$$

Ker se $f(k)$ in k ujemata v vseh bitih od $i+1$ naprej, se mora tam z njima ujemati tudi r. Glede bita i opazimo naslednje: če ima tam r enico, je lahko $\leq k$ le v primeru, če ima na vseh nižjih mestih ničle, torej če velja kar $r = k$. Če pa ima r tam ničlo, potem sme imeti na nižjih bitih (od 0 do $i-1$) poljubne vrednosti, samo da niso vse enake 0 (v tem primeru bi imeli namreč $r = f(k)$, mi pa zahtevamo $r > f(k)$). Če torej odmislimo primer, ko je $r = k$, lahko vse ostale primerne k-je dobimo tako, da

[11] Če naš računalnik za negativna cela števila uporablja predstavitev z dvojiškim komplementom (*two's complement representation*, kar je dandanes običajno), lahko izraz v oklepajih računamo tudi kot $-k$. O tem se lahko prepričamo takole: recimo, da so naše celoštevilske spremenljivke b-bitne; če jih gledamo kot b-bitna cela števila, imajo torej lahko vrednost od 0 do $2^b - 1$, vendar pa se vrednosti od 2^{b-1} do $2^b - 1$ uporabijo za predstavitev negativnih števil, tako da je $-k$ predstavljeno z vrednostjo $2^b - k$. Po drugi strani je **not** k vrednost, v kateri so prižgani natanko tisti biti, ki so v k ugasnjeni; če torej k in **not** k seštejemo, dobimo vrednost, v kateri so prižgani vsi biti, to pa je $2^b - 1$. Torej je **not** $k = 2^b - 1 - k$, zato pa **not** $k + 1 = 2^b - k$, torej ravno ista vrednost, s katero je predstavljeno število $-k$.

v r poiščemo nek tak ugasnjen bit, ki ima nekje desno od sebe vsaj kakšen prižgan bit; naš ugasnjeni bit prižgemo, vse prižgane bite desno od njega pa ugasnemo. To je najlažje početi od desne proti levi: za začetek poiščimo najbolj desni ugasnjeni bit, ki je primeren za prižiganje (torej ki ima desno od sebe še kakšen prižgan bit):

$$\begin{aligned} r &= r_{b-1}\ r_{b-2} \ldots r_{i+1}\ 0\ 1 \ldots 1\ 1\ 0 \ldots 0 \\ r \textbf{ and } (\textbf{not } r+1) &= r_{b-1}\ r_{b-2} \ldots r_{i+1}\ 0\ 0 \ldots 0\ 1\ 0 \ldots 0 \\ r + (r \textbf{ and } (\textbf{not } r+1)) &= r_{b-1}\ r_{b-2} \ldots r_{i+1}\ 1\ 0 \ldots 0\ 0\ 0 \ldots 0 \end{aligned}$$

Ideja je torej naslednja: naj bo j najnižji prižgani bit v r in naj bo i najnižji ugasnjeni bit levo od j. Na mestih od j do $i-1$ so torej v r same enice; če r-ju prištejemo vrednost bita j (ki jo lahko, kot smo videli, izračunamo kot r **and** (**not** $r+1$)), pride pri seštevanju do prenosa naprej pri vseh mestih od j do i, kjer se ničla spremeni v enico in prenašanje se konča. Tako smo torej dosegli ravno to, kar smo želeli: enice na mestih od j do $i-1$ smo ugasnili, prvo naslednjo ničlo (na mestu i) pa prižgali. Tako smo dobili najmanjši primerni k (torej takega, za katerega je $f(k) < r < k$). Iz njega lahko zdaj po enakem postopku dobimo naslednjega: če mu prištejemo vrednost bita i, bomo na mestu i dobili spet ničlo, hkrati se bodo tudi morebitne enice na mestih od $i+1$ naprej ugasnile, vse do prve naslednje ničle, ki se bo spremenila v enico. Ta postopek ponavljamo, dokler ne pademo ven iz tabele (torej dosežemo indeksa, večjega od n). Ker je najnižji prižgani bit po vsaki iteraciji te zanke bolj levo kot prej, je jasno, da smo naredili največ $O(\log n)$ iteracij.

Kako si bomo s Fenwickovim drevesom pomagali pri rešitvi naše naloge? Omejimo se za začetek le na en predmet; naj bo a_i število, ki pove, ali je na dan i kak test iz tega predmeta ($a_i = 1$) ali ne ($a_i = 0$). Število testov v obdobju od d_1 do d_2 potem ni nič drugega kot vsota $a_{d_1} + a_{d_1+1} + \ldots + a_{d_2}$. S Fenwickovim drevesom lahko učinkovito izračunamo s_{d_2} (število testov do vključno datuma d_2) in s_{d_1-1} (število testov pred datumom d_1), razlika med njima pa je ravno rezultat, ki nas zanima (število testov od datuma d_1 do datuma d_2). Tudi začetna inicializacija Fenwickovega drevesa je zelo preprosta: če so na začetku vse vrednosti a_i enake 0, so tudi vse vsote več a_i-jev enake 0, torej moramo tudi v tabeli T postaviti vse elemente na 0. Ker imamo pri naši nalogi več predmetov, bomo za vsakega vzdrževali po eno Fenwickovo drevo.

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

#define P 5
#define MaxD 1000000
int D, t[P][MaxD + 1];

void Dodaj(int *tabela, int d) { /* doda test na dan d */
  while (d <= D) { tabela[d]++; d += d & -d; }}

int KolikoDo(int *tabela, int d) { /* vrne število testov v obdobju 1..d */
  int rezultat = 0;
  while (d > 0) rezultat += tabela[d], d -= d & -d;
  return rezultat; }

int main(int argc, char** argv)
{
  int nSkupin, n, p, d1, d2, s1, s2; char ukaz[2];
```

```
  long long rezultat = 0;
  /* Inicializirajmo Fenwickova drevesa. */
  FILE *f = fopen(argc > 1 ? argv[1] : "urnik.in", "rt");
  fscanf(f, "%d %d", &nSkupin, &D);
  for (p = 0; p < P; p++) for (d1 = 0; d1 <= D; d1++) t[p][d1] = 0;

  while (nSkupin-- > 0) /* Izvedimo operacije iz vhodne datoteke. */
  {
    fscanf(f, "%s", ukaz);
    if (ukaz[0] == 'V') { /* Vnos. */
      fscanf(f, "%d %d %d %d", &n, &p, &d1, &s1);
      while (n-- > 0) { Dodaj(t[p - 1], d1); d1 += s1; }}
    else if (ukaz[0] == 'P') { /* Poizvedba. */
      fscanf(f, "%d %d %d %d %d %d", &n, &p, &d1, &d2, &s1, &s2);
      while (n-- > 0) {
        rezultat += KolikoDo(t[p - 1], d2) - KolikoDo(t[p - 1], d1 - 1);
        d1 += s1; d2 += s2; }}
  }
  /* Izpišimo rezultate. */
  fclose(f);
  f = fopen(argc > 2 ? argv[2] : "urnik.out", "wt"); fprintf(f, "%lld\n", rezultat); fclose(f);
  return 0;
}
```

Poleg učinkovitosti je pri Fenwickovem drevesu lepo še posebej to, da ga je, ko enkrat razumemo idejo za njim, izredno preprosto implementirati; vidimo lahko, kako kratka in preprosta sta podprograma Dodaj in KolikoDo v gornji rešitvi.

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Lemingi

Nalogo lahko rešimo tako, da simuliramo gibanje leminga. Njegovo trenutno x-koordinato hranimo v spremenljivki x, smer pa v s. Na začetku ju inicializiramo na x_0 oz. s_0 (začetni položaj in smer), nato pa pregledujemo ploščadi od višjih proti nižjim (besedilo naloge pravi, da so tako ali tako že podane v tem vrstnem redu) in pri vsaki pogledamo, ali bi leming padel nanjo ali ne; če pade na ploščad, spremenimo njegovo smer in izračunamo x-koordinato krajišča, pri kateri bo padel z nje. Ob koncu tega postopka imamo v spremenljivki x ravno zadnjo x-koordinato, ki jo leming na ta način doseže. S psevdokodo bi lahko naš postopek opisali takole:

```
x := x0; s := s0;
for i := 1 to n:                (* Pregledujmo ploščadi od višjih proti nižjim. *)
  if xi < x and x < xi + di:    (* Ali pade leming na i-to ploščad? *)
    if s = D:                   (* Spremenimo smer iz desne v levo in *)
      s := L; x := xi;                          (* pademo z levega krajišča. *)
    else:                       (* Spremenimo smer iz leve v desno in *)
      s := D; x := xi + di;                     (* pademo z desnega krajišča. *)
return x;
```

Zapišimo še rešitev v C-ju:

```
int Leming(int x0, char s0, int n, const int xi[], const int yi[], const int di[])
{
  int i, x = x0; char s = s0;
  for (i = 0; i < n; i++)
    if (xi[i] < x && x < xi[i] + di[i])
      if (s == 'D') s = 'L', x = xi[i];
      else s = 'D', x = xi[i] + di[i];
  return x;
}
```

2. 3-d šah

Šahovnico predstavimo s trodimenzionalno tabelo $8\times8\times8$ celih števil, v kateri vsaka celica predstavlja eno od polj šahovnice, vrednost te celice pa nam pove, koliko kraljic napada to polje. Na začetku torej postavimo vsa števila v tabeli na 0, nato pa se za vsako kraljico sprehodimo po vseh poljih, ki jih napada, in pripadajoče elemente tabele povečamo za 1. Na koncu moramo tako le še prešteti, koliko elementov tabele ima vrednost 3 (kar je znak, da tisto polje napadajo vse tri kraljice).

Ostane še vprašanje, kako za neko kraljico ugotoviti, katera polja napada (v spodnjem programu se s tem ukvarja podprogram ObdelajKraljico). Tega se lahko lotimo na več načinov, na primer takole. Smer napada kraljice lahko opišemo s trojico števil $(\triangle x, \triangle y, \triangle z)$, pri čemer je vsako od njih lahko -1, $+1$ ali 0 in nam pove, kako se v tej smeri spreminja ena od koordinat (torej $\triangle x = 1$ pomeni, da se x-koordinata povečuje, $\triangle y = -1$ pomeni, da se y-koordinata zmanjšuje in podobno). Ne glede na položaj kraljice in opazovano smer pa vemo, da kraljica ne bo napadala več kot sedem polj v tisti smeri, saj bi potem zagotovo že padli čez rob šahovnice.

Naš podprogram lahko torej z nekaj zankami pregleda vse možne smeri napada in se v vsaki smeri sprehodi sedem polj daleč ter vsa tako dosežena polja označi kot napadena (v ta namen imamo še pomožni podprogram Oznaci, ki pred tem še preveri, ali ne leži zahtevano polje mogoče že zunaj šahovnice).

```
#include <stdio.h>

int a[8][8][8]; /* a[x][y][z] pove, koliko kraljic napada polje (x, y, z) */

/* Označi dano polje kot napadeno (če ne leži zunaj šahovnice). */
int Oznaci(int x, int y, int z) {
  if (0 <= x && x < 8 && 0 <= y && y < 8 && 0 <= z && z < 8) a[x][y][z]++; }

/* Obišče vsa polja, ki jih napada kraljica s položaja (x, y, z). */
int ObdelajKraljico(int x, int y, int z)
{
  int dx, dy, dz, d;
  Oznaci(x, y, z); /* Kraljica napada tudi polje, na katerem stoji. */
  /* Preglejmo vse možne smeri napada. */
  for (dx = -1; dx <= 1; dx++) for (dy = -1; dy <= 1; dy++)
  for (dz = -1; dz <= 1; dz++)
    /* Smer dx = dy = dz = 0 je neveljavna, saj se koordinate takrat ne spreminjajo. */
    if (dx != 0 || dy != 0 || dz != 0)
      /* Obiščimo 7 polj v tej smeri in jih označimo kot napadena. */
      for (d = 1; d <= 7; d++) Oznaci(x + dx * d, y + dy * d, z + dz * d);
}

int main()
{
  int x, y, z, n;
  /* Inicializirajmo vse celice tabele a na 0. */
  for (x = 0; x < 8; x++) for (y = 0; y < 8; y++) for (z = 0; z < 8; z++)
    a[x][y][z] = 0;
  /* Preberimo položaj vseh treh kraljic in označimo napadena polja. */
  for (n = 0; n < 3; n++) {
    scanf("%d %d %d", &x, &y, &z); ObdelajKraljico(x - 1, y - 1, z - 1); }
  /* Preštejmo, koliko polj napadajo vse tri kraljice. */
  for (n = 0, x = 0; x < 8; x++) for (y = 0; y < 8; y++) for (z = 0; z < 8; z++)
    if (a[x][y][z] == 3) n++;
  /* Izpišimo rezultat. */
  printf("%d\n", n); return 0;
}
```

Zelo elegantna pa je tudi naslednja rešitev. Recimo, da imamo kraljico na položaju (x_k, y_k, z_k) in da nas zanima, če ta kraljica napada polje (x, y, z). Izračunajmo absolutne vrednosti razlik: $\triangle x = |x_k - x|$, $\triangle y = |y_k - y|$ in $\triangle z = |z_k - z|$. Če pogledamo definicijo tega, katera polja kraljica napada, vidimo, da je polje napadeno natanko v naslednjih primerih: (1) če sta dva od $\triangle x$, $\triangle y$ in $\triangle z$ enaka 0 (tretji pa ima lahko v tem primeru poljubno vrednost) — to je napad v smeri, vzporedni z eno od koordinatnih osi; (2) če je eden od njih enak nič, druga dva pa imata poljubno vrednost, vendar oba enako — to je napad v smeri ene od ravninskih diagonal; (3) če imajo vsi trije enako vrednost — to je napad v smeri ene od diagonal kocke. Na

podlagi tega razmisleka lahko za katerokoli polje šahovnice preprosto preverimo, ali ga posamezna kraljica napada ali ne; v spodnjem programu to počne funkcija Napadeno. Glavni del našega programa se z nekaj gnezdenimi zankami sprehodi po vseh poljih in za vsako preveri, če ga napadajo vse tri kraljice.

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

bool Napadeno(int dx, int dy, int dz)
{
  int d = dx; if (d == 0) d = dy; else if (dy != 0 && dy != d) return false;
  return (d == 0 || dz == 0 || d == dz);
}

int main()
{
  int x, y, z, i, n, xk[3], yk[3], zk[3];
  /* Preberimo položaj vseh treh kraljic. */
  for (i = 0; i < 3; i++) scanf("%d %d %d", &xk[i], &yk[i], &zk[i]);
  /* Za vsako polje poglejmo, če ga napadajo vse kraljice. */
  for (n = 0, x = 1; x <= 8; x++) for (y = 1; y <= 8; y++) for (z = 1; z <= 8; z++) {
    i = 0; while (i < 3 && Napadeno(abs(x - xk[i]), abs(y - yk[i]), abs(z - zk[i]))) i++;
    if (i == 3) n++; }
  /* Izpišimo rezultat. */
  printf("%d\n", n); return 0;
}
```

3. Brisanje parov

Pri tej nalogi si je koristno pomagati s skladom. Na začetku imejmo prazen sklad. Vhodni niz berimo črko za črko od leve proti desni. ko naletimo na malo črko, jo odložimo na vrh sklada; pri veliki črki pa pogledamo, če je na vrhu sklada pripadajoča mala črka; če je, jo pobrišemo, sicer pa veliko črko dodamo na sklad. Na koncu tega postopka je vsebina sklada ravno okrajšani niz, po katerem sprašuje naloga. Oglejmo si delovanje tega postopka na primeru iz naloge: vhodni niz je torej `abbBAacCAdCcaBb`.

Prebrani znak	Stanje sklada (vrh je na desni)
a	a
b	ab
b	abb
B	ab
A	abA
a	abAa
c	abAac
C	abAa
A	abA

in tako naprej.

Ker na sklad po vsakem prebranem znaku vhodnega niza dodamo največ en znak, včasih pa s sklada tudi kaj pobrišemo, sledi, da je po k prebranih vhodnih znakih

niz na skladu dolg največ k (ali pa še manj). Zato lahko naš spodnji podprogram za shranjevanje sklada uporablja kar isto tabelo niz, v kateri je dobil vhodni niz (kajti naslednji še neprebrani vhodni znak gotovo leži v tistem delu tabele, ki je sklad doslej še ni dosegel in povozil njene vsebine). V spremenljivki n vodimo podatek o trenutni dolžini sklada. Na koncu je tako v tabeli niz ravno okrajšana različica prvotnega vhodnega niza; le z ničelnim znakom ga moramo še zaključiti, pa nastane iz njega veljaven C/C++ovski niz.

```
void Okrajsaj(char *niz)
{
  int n = 0; char c, *p = niz;
  while (*p) {
    c = *p++;
    if (n > 0 && JeMalaCrka(niz[n − 1]) && c == DajVeliko(niz[n − 1])) n−−;
    else niz[n++] = c; }
  niz[n] = 0;
}
```

Oglejmo si še rešitev v pythonu. Tu smo kot sklad uporabili kar običajen pythonov seznam (*list*) in ga na koncu z metodo join staknili v niz:

```
def Okrajsaj(niz):
  sklad = []
  for c in niz:
    if sklad and JeMalaCrka(sklad[−1]) and c == DajVeliko(sklad[−1]): sklad.pop()
    else: sklad.append(c)
  return "".join(sklad)
```

4. Ta5nik

Naloge se lahko lotimo na več načinov. Preprosta in za naše namene čisto dovolj dobra rešitev je, da se v zanki premikamo naprej po vhodnem nizu s in na vsakem mestu preverimo, ali se vsebina niza s od trenutnega položaja naprej ujema s kakšnim od vzorcev (nizov nič, ena in tako naprej). Če opazimo ujemanje, povečamo števec zamenjav in v izhodni niz (v spodnji rešitvi kaže spremenljivka izhod na začetek izhodnega niza, konec pa na njegov konec) zapišemo števko, s katero moramo zamenjati pravkar odkriti vzorec (npr. nič zamenjamo z 0 itd.). Če pa pregledamo vse vzorce, ne da bi pri katerem ugotovili, da se pojavlja na trenutnem položaju v nizu s, lahko trenutni znak niza s skopiramo v izhodni niz in se po nizu s premaknemo za eno mesto naprej. Ta postopek ponavljamo, dokler ne pridemo do konca niza s. Na koncu le še preverimo, če smo izvedli vsaj dve zamenjavi, in v tem primeru izpišemo dobljeni izhodni niz.

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

const char *vzorci[10] = { "nič", "ena", "dve", "tri", "štiri",
                           "pet", "šest", "sedem", "osem", "devet" };

void Ta5nik(const char *s)
{
  char *izhod = (char *) malloc(1 + strlen(s)), *konec; const char *S, *P;
```

```
  int stZamenjav = 0, d; konec = izhod;
  while (*s)
  {
    for (d = 0; d <= 9; d++) {
      /* Primerjajmo niz vzorci[d] z vhodnim nizom od trenutnega položaja naprej. */
      P = vzorci[d]; S = s; while (*P && *P == *S) P++, S++;
      /* Če smo prišli do konca vzorca, ne da bi opazili neujemanje,
         izvedimo zamenjavo. */
      if (! *P) { *konec++ = '0' + d; stZamenjav++; s = S; break; }}
    /* Če pri nobenem vzorcu nismo izvedli zamenjave, skopirajmo trenutni znak
       brez sprememb v izhodni niz. */
    if (d > 9) *konec++ = *s++;
  }
  /* Če smo izvedli več kot eno zamenjavo, izhodni niz izpišimo. */
  if (stZamenjav > 1) { *konec = 0; printf("%s\n", izhod); }
  free(izhod);
}
```

V gornji rešitvi smo to, ali se vzorec vzorci[d] pojavlja v trenutnem položaju vhodnega niza, preverjali z notranjo zanko **while**, ki je primerjala znake vzorca in znake niza `s`, dokler ne pride do konca vzorca ali pa zazna neujemanja. Namesto z zanko bi lahko to rešili tudi s funkcijo strncmp iz standardne knjižnice:

```
    if (0 == strncmp(s, vzorci[d], strlen(vzorci[d]))) {
      *konec++ = '0' + d; stZamenjav++; s += strlen(vzorci[d]); break; }}
```

V nekaterih programskih jezikih imamo pri roki kakšno standardno funkcijo za zamenjavo vseh pojavitev nekega podniza v danem nizu. V tem primeru nam lahko pride na misel, da bi to funkcijo preprosto klicali desetkrat: najprej bi zamenjali vse pojavitve podniza `nič` z nizom `0`, nato vse pojavitve podniza `ena` z nizom `1` in tako naprej. Težava te rešitve je, da bi na primer v nizu `petrijevka` najprej opravili zamenjavo `tri` → `3` in tako dobili `pe3jevka`, medtem ko naloga zahteva, da moramo v primeru prekrivanja vzorcev opraviti najbolj levo možno zamenjavo (`5rijevka`). Na srečo za konkretnih deset vzorcev, s katerimi delamo pri tej nalogi (nizi `nič`, `ena`, ..., `devet`) velja, da lahko pride do prekrivanja vzorcev — torej do primerov, ko se konec (sufiks) enega vzorca ujema z začetkom (prefiksom) drugega vzorca — le v naslednjih štirih primerih: `dvena`, `petri`, `šestri` in `devetri`. Vidimo torej, da bomo do najbolj leve zamenjave gotovo prišli, če izvedemo `ena` → `1` za `dve` → `2` in če izvedemo `tri` → `3` za `pet` → `5`, `šest` → `6` in `devet` → `9`. Tako pridemo na primer do naslednje rešitve v pythonu:[12]

```
def Ta5nik(s):
  vzorci = ("nič", "ena", "dve", "tri", "štiri",
            "pet", "šest", "sedem", "osem", "devet")
  stZamenjav = 0
  for d in (0, 2, 4, 5, 6, 7, 8, 9, 1, 3):
```

[12] Slabost tega pristopa je, da ga ne moremo enostavno posplošiti na poljuben nabor vzorcev, saj pri nekaterih naborih vzorcev sploh ni nobenega fiksnega vrstnega reda zamenjav, ki bi nam zagotavljal, da bo vedno izvedel najbolj levo možno zamenjavo. Na primer, če bi morali izvajati zamenjavi `cdc` → `0` in `dcd` → `1`, potem bi pri nekaterih vhodnih nizih dobili pravilni rezultat tako, da bi najprej izvedli vse možne zamenjave `cdc` → `0` in nato vse `dcd` → `1`, pri nekaterih vhodnih nizih bi morali narediti ravno obratno (naprej vse `dcd` → `1` in nato vse `cdc` → `0`), pri nekaterih vhodnih nizih pa celo nič od tega dvojega ne bi dalo pravilnega rezultata.

```
    stZamenjav += s.count(vzorci[d])
    s = s.replace(vzorci[d], str(d))
if stZamenjav >= 2: print s
```

Nekaj težav nam je povzročila še zahteva iz naloge, da moramo okrajšani niz izpisati le, če smo izvedli več kot eno zamenjavo. Metoda `replace` nam ne pove, koliko zamenjav je izvedla, zato pred zamenjavo pokličemo še `s.count(vzorci[d])`, ki nam prešteje, kolikokrat se v `s` pojavlja tisti vzorec.

5. Koalicije

Naloga zahteva, da mora biti od vsakega para sovražnih strank natanko ena v koaliciji; to nam močno omeji možno sestavo koalicij. Če sta stranki u in v sovražni in vzamemo u v koalicijo, potem vemo, da v ne smemo vzeti v koalicijo; in ker zdaj v ni v koaliciji, moramo v koalicijo vzeti vse tiste stranke, ki so sovražne stranki v; in tako naprej. Vsakič torej, ko za neko stranko odločimo, ali bo v koaliciji ali ne, vemo, da iz tega tudi za njene sovražnice enolično sledi, ali bodo morale biti v koaliciji ali zunaj nje. Pri tem si je koristno pomagati s seznamom (vrsto), v katerega dodajamo stranke, za katere smo že določili, ali bodo v koaliciji ali ne, nismo pa še pregledali njihovih sovražnic. V glavni zanki bomo jemali stranke iz vrste in pri vsaki od njih določili stanje njenih sovražnic. Ta postopek ponavljamo, dokler se nam vrsta strank, ki še čakajo na obdelavo, ne izprazni.

Med delom si bomo pomagali še s tabelo a, v kateri za vsako stranko u vodimo podatek o tem, ali je v koaliciji ($a_u = 1$) ali ne ($a_u = -1$) ali pa da zanjo še ne vemo, če bo v koaliciji ali ne ($a_u = 0$). Postopek je torej takšen:

for $u := 1$ **to** n **do** $a_u := 0$;
$Q :=$ prazna vrsta;
$a_1 := 1$; dodaj 1 v Q; (* *Recimo, da bo stranka 1 v koaliciji.* *)
while Q ni prazna:
  naj bo u poljubna stranka iz Q; pobriši u iz Q;
  za vsako v, ki je sovražna stranki u:
    if $a_v = 0$ **then**
      $a_v = -a_u$; dodaj v v vrsto Q;
    else if $a_v = a_u$ **then**
      dani vhodni primer je nerešljiv: iz dosedanjih omejitev za stranki
      u in v sledi, da bi morali biti obe v koaliciji ali pa obe zunaj nje,
      hkrati pa sta si sovražni, torej ne bomo mogli izpolniti zahteve, da je v
      koaliciji natanko ena od njiju;

Vidimo, da smo postopek začeli tako, da smo stranko 1 vzeli v koalicijo. Če bi namesto tega začeli s tem, da stranko 1 izključimo iz koalicije (postavimo $a_1 := -1$), bi se preostanek postopka odvrtel popolnoma enako, le da bi bili predznaki vseh elementov v tabeli a obrnjeni in v koalicijo bi dobili ravno tiste stranke, ki so bile prej (pri $a_1 = +1$) zunaj nje. Postopka nam torej pravzaprav ni treba izvajati še enkrat, ampak je dovolj že, če na koncu sešejemo število poslancev v koaliciji in število tistih zunaj nje ter vrnemo večjo od obeh vrednosti.

Dosedanji postopek še ni čisto pravilna rešitev naše naloge. Lahko se namreč zgodi, da ob koncu izvajanja glavne zanke za nekatere stranke še ni določil, ali so v

koaliciji ali ne (z drugimi besedami, nekateri elementi tabele a so mogoče še vedno enaki 0). Če bi vzeli recimo primer iz besedila naloge in začeli pri stranki 1 (tako kot to stori naš gornji postopek), bi v nadaljevanju obdelali še stranke 2, 3 in 6, ne pa tudi strank 4 in 5, kajti tidve prek relacije sovražnosti nista niti posredno niti neposredno povezani s stranko 1. Relacija sovražnosti nam torej množico strank lahko razbije na več delov, ki so vsak znotraj sebe sicer povezani, niso pa povezani drug z drugim. Zgoraj opisani postopek je torej obdelal šele enega od teh delov, v nadaljevanju pa moramo z njim obdelati še ostale dele in na koncu rezultate sešteti.

```
for u := 1 to n do a_u := 0;
M := 0;        (* Velikost največje koalicije. *)
for w := 1 to n do
  if a_w ≠ 0 then
    continue;    (* Točko w smo nekoč že obdelali. *)
  Q := prazna vrsta; dodaj w v vrsto Q;
  m_1 = n_w; m_{-1} := 0;    (* Št. poslancev v koaliciji in zunaj nje. *)
  while Q ni prazna:
    naj bo u poljubna stranka iz Q; pobriši u iz Q;
    za vsako v, ki je sovražna stranki u:
      if a_v = 0 then
        a_v = -a_u; dodaj v v vrsto Q; m_{a_v} := m_{a_v} + n_v;
      else if a_v = a_u then
        dani vhodni primer je nerešljiv;
  M := M + max{m_1, m_{-1}};
return M;
```

Naloge so sestavili: kino — Nino Bašić; brisanje parov, dvigalo, koalicije, silhuete — Primož Gabrijelčič; ribič, moderna umetnost, požar — Tomaž Hočevar, koalicije — Boris Horvat; lemingi — Nace Hudobivnik; binarni sef — Jurij Kodre; 3-d šah, dekodiranje nizov — Mitja Lasič; kolera — Mark Martinec; ta5nik — Polona Novak; sumljiva imenovanja, natrpan urnik — Jure Slak; iglični tiskalnik — Boštjan Slivnik; vandali, številčenje — Mitja Trampuš; kompleksnost števil — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2011

1. Primerjava IPjev

Naloga pravi, da je 128-bitna IP-številka predstavljena tako, da je razdeljena na osem 16-bitnih števil, vsako od teh pa je predstavljeno z največ štirimi šestnajstiškimi številkami. Koristno bi bilo torej predelati niz nazaj v tabelo osmih 16-bitnih celih števil; ko bomo imeli obe IP-številki v taki predstavitvi, ju ne bo težko primerjati med seboj.

Sestavimo torej podprogram, ki pretvori niz s v tabelo osmih celih števil ip. Naš podprogram se sprehaja po znakih niza s in sproti popravlja vrednosti v tabeli ip: ko preberemo novo številko, moramo trenutni element tabele pomnožiti s 16 in mu prišteti vrednost nove številke. Ko preberemo dvopičje, pa se moramo premakniti na naslednji element tabele (števec i nam pove, s katerim elementom se trenutno ukvarjamo).

Paziti moramo še na možnost, da je v nizu nekaj skupin številk opuščenih in zamenjanih z dvojnim dvopičjem (::). Naša spodnja rešitev se najprej pretvarja, kot da je tudi dvojno dvopičje le še ena čisto navadna meja med dvema skupinama številk, vendar pa si tudi zapomni, pri kateri skupini številk je do tega prišlo (spremenljivka vrzel). Ko pridemo do konca niza s, nam spremenljivka i pove, katera je zadnja celica v tabeli ip, za katero smo prebrali kakšno številko; vemo pa, da ima zadnja celica v tabeli indeks 7, tako da moramo vriniti $7 - i$ ničel na mesto, kjer je bilo v vhodnem nizu dvojno dvopičje (torej na indeks vrzel v tabeli ip), staro vsebino tabele pa pri tem premikamo naprej po tabeli.

```
void PretvoriIP(const char *s, int ip[8])
{
  int i, vrzel = 8, zamik; char c;
  for (i = 0; i < 8; i++) ip[i] = 0;
  for (i = (*s == ':' ? -1 : 0); *s; )
  {
    c = *s++;
    if (c == ':') { /* Pri dvopičju se premaknemo naprej po tabeli ip. */
      i++;
      /* Če imamo dvojno dvopičje, si zapomnimo, da je tu vrzel. */
      if (*s == ':') vrzel = i, s++; }
    else { /* Smo pri naslednji številki; ustrezno popravimo ip[i]. */
      ip[i] *= 16;
      if ('0' <= c && c <= '9') ip[i] += c - '0';
      else if ('A' <= c && c <= 'F') ip[i] += c - 'A' + 10;
      else if ('a' <= c && c <= 'f') ip[i] += c - 'a' + 10; }
  }
  /* Vrinimo ustrezno število ničel na indeks 'vrzel'. */
  if (i < 8) {
    zamik = 7 - i;
    while (i >= vrzel) { ip[i + zamik] = ip[i]; i--; }
    for (i = vrzel; i < vrzel + zamik; i++) ip[i] = 0; }
}
```

Nekaj dela je še z raznimi posebnimi primeri. Na primer, niz se sicer ne more začeti na eno samo dvopičje (ker to označuje mejo med dvema skupinama številk), lahko pa

se začne na dvojno dvopičje. Običajno tudi dvojno dvopičje obravnavamo kot mejo med dvema skupinama števk (in zato takrat povečamo i za 1); če pa se pojavlja na začetku niza, tega v bistvu ne bi smeli narediti. Zato v tem primeru začnemo glavno zanko pri $i = -1$ namesto $i = 0$; takoj v prvi iteraciji potem opazimo dvojno dvopičje in povečamo i na 0, tako da se potem v nadaljevanju zanke vpisujejo števila pravilno od začetka tabele.

Še en poseben primer (za katerega je sicer že vprašljivo, če je sintaktično pravilen), je ta, da niz vsebuje 8 skupin števk, potem pa še (nepotrebno) dvojno dvopičje. V tem primeru bi nam zanke za vrivanje ničel pri vrzeli le pokvarile vsebino tabele, zato pred tem vrivanjem preverimo, če je $i < 8$.

Zdaj znamo pretvoriti niz v tabelo števil; primerjava dveh IP-številk je potem zelo preprosta:

```
#include <stdbool.h>
bool PrimerjajIPja(const char *s, const char *t)
{
  int ip1[8], ip2[8], i;
  PretvoriIP(s, ip1); PretvoriIP(t, ip2);
  for (i = 0; i < 8; i++)
    if (ip1[i] != ip2[i]) return false;
  return true;
}
```

Nalogo lahko rešimo tudi tako, da niz z IP-številko „normaliziramo“ v neskrajšano obliko, torej takšno, ki ima prisotnih vseh 8 skupin števk in v vsaki natanko štiri števke. Poleg tega tudi spremenimo velike črke v male. Po teh spremembah smo lahko prepričani, da če se naša dva niza z IP-številkama še vedno razlikujeta, je to zato, ker vsebujeta različni IP-številki (in ne različnih predstavitev iste številke). Oglejmo si primer takšne rešitve v pythonu:

```
import re
def NormalizirajIP(s):
  s = (":" + s.lower().replace("::", "#")).replace(":#", "#")
  s = s.replace(":", ":000").replace("#", "#000").rstrip('0')
  s = re.sub(r"([:#])[0-9a-f]{0,3}([0-9a-f]4)", r"\1\2", s)
  return s.replace("#", ":0000" * ((41 - len(s)) // 5) + ":").strip(':')

def PrimerjajIPja(s, t): return NormalizirajIP(s) == NormalizirajIP(t)
```

Naša rešitev za začetek spremeni velike črke v male, morebitno dvojno dvopičje (`::`) spremeni v `#` (kar bo prišlo prav pri kasnejših zamenjavah). Poleg tega postavi na začetek niza še eno dvopičje, razen če se je niz začel na `#` (oz. bivše dvojno dvopičje `::`). Po teh spremembah je naš niz sestavljen iz največ 8 skupin, pri čemer je vsaka skupina sestavljena iz znaka `:` ali `#`, ki mu sledijo od 1 do 4 šestnajstiške števke. Za zadnjo skupino je mogoče prisoten še znak `#`.

V drugi vrstici vrinemo za vsakim `:` in `#` še tri ničle (razen za morebitnim `#` na koncu niza), s čimer zagotovimo, da bodo v vsaki skupini vsaj štiri števke. V tretji vrstici nato od vsake skupine števk obdržimo le zadnje štiri (če smo torej v drugi vrstici po nepotrebnem vrinili kakšno ničlo preveč, zdaj takšne odvečne ničle pobrišemo).

V našem nizu je zdaj torej vsaka skupina dolga natanko pet znakov (najprej `:` ali `#` in nato še štiri šestnajstiške številke). Če je prisotnih vseh osem skupin, je niz dolg 40 znakov (ali 41, če je na koncu še `#`, ki je sicer v tem primeru odveč). Če je npr. prisotnih le sedem skupin, je niz dolg 35 znakov (ali 36, če je na koncu še `#`); ipd. Če torej pogledamo razliko med 41 in dolžino niza ter jo delimo s 5, nam količnik pove, koliko skupin številk manjka in jih moramo zdaj vriniti pri znaku `#`. Slednjega torej zdaj zamenjamo s primerno dolgim podnizom oblike `:0000:0000:...:0000:`. Po tem moramo le še pobrisati odvečni dvopičji na začetku in koncu niza.

2. Dvigalo

Potnik i pride do dvigala ob času t_i; v najboljšem primeru porabi za pot do pritličja $s \cdot n_i$ časa (če ga dvigalo že čaka in ga takoj odpelje naravnost v pritličje). Potnika i torej v pritličje ne moremo spraviti prej kot ob času $t_i + s \cdot n_i$, to pa tudi pomeni, da zadnjega potnika ne moremo spraviti v pritličje prej kot ob času $T := \max_i(t_i + s \cdot n_i)$.

Označimo z $N := \max_i n_i$ najvišje nadstropje, v katerem imamo kakšnega potnika. Recimo zdaj, da bi dvigalo ob času $T - 2N \cdot s$ poslali iz pritličja navzgor, ob času $T - N \cdot s$ bi doseglo N-to nadstropje in se nato odpeljalo dol v pritličje, ob tem pa spotoma pobiralo potnike. Pritličje bi doseglo ob času T. Če nam je uspelo ob tem pobrati vse potnike, smo dobili optimalno rešitev, saj smo se v prejšnjem odstavku prepričali, da prej kot ob času T gotovo ne moremo spraviti vseh potnikov v pritličje.

Ali lahko naše dvigalo res pobere vse potnike? Na poti navzdol pride v nadstropje n_k ob času $T - s \cdot n_k$; ker je T maksimum vrednosti $t_i + s \cdot n_i$ po vseh i, je gotovo $T \geq t_k + s \cdot n_k$; zato je $T - s \cdot n_k \geq t_k$, torej naše dvigalo na poti navzdol pride v nadstropje n_k dovolj pozno, da lahko pobere potnika k (ki pride v tem nadstropju čakat dvigalo ob času t_k). Ker lahko opravimo tak razmislek za vsak k, vidimo, da lahko naše dvigalo na poti navzdol res pobere vse potnike.

Tako smo se torej prepričali, da je najzgodnejši čas, ob katerem lahko vratar odide domov, ravno T. Izračunamo ga lahko z zanko po vseh potnikih:

$T := -\infty$;
for $i := 1$ **to** z:
  if $t_i + s \cdot n_i > T$ **then** $T := t_i + s \cdot n_i$;
return T;

3. Številski sestavi

Naš program bere podatke v zanki; najprej preberemo število b in če ima to vrednost 0, potem vemo, da smo na koncu vhodnih podatkov in se lahko ustavimo. Sicer pa preskočimo morebitne presledke pred številom n in nato preberemo n po znakih (ki so zdaj pravzaprav števke n-ja v b-iškem zapisu).

Ko dobimo novo števko, moramo dosedanjo vrednost n-ja pomnožiti z b in prišteti novo vrednost števke. Na primer: recimo, da smo pri $b = 10$ in da beremo število 567. Po tistem, ko smo prebrali prvi dve števki, ima n vrednost 56, naslednja števka pa je 7; nova vrednost n-ja mora biti torej 567, kar dobimo s formulo $56 \cdot 10 + 7$. To deluje tudi pri drugih sestavih, le namesto z 10 moramo množiti z b.

Ko pretvarjamo števko `c` iz znaka (**char**) v številsko vrednost, je koristno ločiti števke `0`, ..., `9` od števk `a`, ..., `z`. Pri prvih moramo le odšteti številsko kodo znaka `'0'`, pa bomo dobili število od 0 do 9. Pri drugih pa moramo odšteti `'a'` in prišteti 10, tako da nam iz `'a'` nastane 10, iz `'z'` nastane 35 in podobno.

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

int main()
{
  int b, c, n, vsota = 0;
  while (1 == scanf("%d", &b) && b > 0)
  {
    /* Preskočimo presledke med osnovo in številom. */
    do { c = fgetc(stsin); } while (! isalnum(c));
    /* Preberimo število. */
    n = 0;
    while (isalnum(c)) {
      n *= b;
      if ('0' <= c && c <= '9') n += c - '0';
      else n += c - 'a' + 10;
      c = fgetc(stdin); }
    vsota += n;
  }
  printf("%d\n", vsota); return 0;
}
```

V nekaterih programskih jezikih je pretvarjanje med številskimi sestavi del standardne knjižnice, kar nam nalogo precej olajša. Bolj za šalo kot zares zapišimo na primer enovrstično rešitev v pythonu:

```
print sum(int(n, int(b)) for [b, n] in map(lambda s: s.split(), __import__('sys').stdin))
```

Vsako vrstico standardnega vhoda (`stdin`) torej razbijemo pri presledku (`split`) na dva niza, `b` in `n`; prvega pretvorimo v celo število in ga uporabimo pri pretvorbi `n`-ja kot drugi parameter, s katerim povemo, iz katerega sestava bi radi pretvorili `n`. Tako dobljena števila seštejemo (`sum`) in vsoto izpišemo (**print**).

4. Puškomitraljez

Ker je možnih le 26 različnih črk, lahko podatke o tem, katere smo v trenutni besedi že videli, hranimo kot bite v celoštevilski spremenljivki (v spodnji številki je to `crke`); bit 0 nam torej pove, ali smo v trenutni besedi že videli črko a, bit 1 nam pove, ali smo že videli b in tako naprej. Pri vsaki črki pogledamo, ali je pripadajoči bit že prižgan; če je, potem vemo, da gledamo zdaj že vsaj drugo pojavitev iste črke, torej beseda ne ustreza zahtevam naloge (to si zapomnimo tako, da spremenljivko `ok` postavimo na **false**). Spotoma tudi štejemo črke v besedi, da bomo na koncu poznali njeno dolžino.

Ko pridemo do konca besede (kar prepoznamo po tem, da naslednji prebrani znak ni črka, ampak nekaj drugega — lahko je znak za konec vrstice ali pa `EOF`), pogledamo, ali je ustrezala zahtevam naloge in ali je daljša od najdaljše take besede doslej; če je, si njeno dolžino zapomnimo (v spremenljivki `naj`, ki jo na koncu izvajanja tudi izpišemo).

Na koncu besede se moramo tudi pripraviti na obdelavo naslednje besede: dolžino postavimo spet na 0, pobrišemo bitno karto crke in postavimo zastavico ok nazaj na **true** (saj beseda velja za ustrezno, dokler se med pregledovanjem njenih črk ne prepričamo o nasprotnem).

```
int main()
{
  int naj = 0, dolzina = 0, crke = 0, c; bool ok = true;
  do {
    c = getc(stdin);
    if ('a' <= c && c <= 'z') {
      dolzina++;
      if (crke & (1 << (c - 'a'))) ok = false;
      else crke |= 1 << (c - 'a'); }
    else { /* smo pri presledku med besedama */
      if (ok && dolzina > naj) naj = dolzina;
      dolzina = 0; crke = 0; ok = true; }
  } while (c != EOF);
  printf("%d\n", naj); return 0;
}
```

Kot zanimivost omenimo, da so najdaljše slovenske besede s samimi različnimi črkami, kar smo jih uspeli najti,[13] dolge po 14 črk. Poleg besede *puškomitraljez* iz naslova naše naloge so take še *nerazumljivost*, *prisluškovanje*, *republikanstvo*, *spodbujevalnik*, *združljivosten* in še več drugih. Nekatere bi lahko seveda še malo podaljšali, če bi jih postavili v kak drug sklon (npr. *prisluškovanjem*). V angleščini smo našli tudi eno s 15 črkami, namreč *dermatoglyphics* (veda o prstnih odtisih).

5. Disemvowelling

Nalogo lahko rešimo na več načinov. Lahko si na primer pripravimo nov niz t in vanj nato v dveh prehodih čez vhodni niz s skopiramo najprej vse ne-samoglasnike, nato pa še vse samoglasnike. Na koncu niz t izpišemo.

```
void PobrisiSamoglasnike(char *s)
{
  bool jeSamoglasnik;
  int i, j, dolzina = strlen(s);
  char *t = (char *) malloc(dolzina + 1);
  /* Skopirajmo v t vse ne-samoglasnike iz s-ja. */
  for (i = 0; i < dolzina; i++)
    if (! (s[i] == 'a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'o' || s[i] == 'u')) t[j++] = s[i];
  /* Skopirajmo v t vse samoglasnike iz s-ja. */
  for (i = 0; i < dolzina; i++)
    if (s[i] == 'a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'o' || s[i] == 'u') t[j++] = s[i];
  t[n] = 0; printf("%s\n", t); free(t);
}
```

[13] V zbirki slovenskih besed na strežniku Inštituta za slovenski jezik pri ZRC SAZU, `http://bos.zrc-sazu.si/besede.html`.

Dovolj pa je tudi en sam prehod čez s; pri tem izpisujemo ne-samoglasniške znake, samoglasnike pa le preštejemo. Naloga namreč pravi, da je vseeno, v kakšnem vrstnem redu izpišemo samoglasnike, zato je dovolj že, da vemo, koliko pojavitev katerega samoglasnika moramo izpisati. Izpišemo jih lahko na primer kar v abecednem vrstnem redu.

```
void PobrisiSamoglasnike2(char *s)
{
  const char *sam = "aeiou";
  int n[5], i, j;
  /* Postavimo števce samoglasnikov na 0. */
  for (i = 0; i < 5; i++) n[i] = 0;
  /* Zapeljimo se po nizu s. */
  for ( ; *s; s++) {
    /* Poglejmo, ali je trenutni znak s-ja samoglasnik. */
    for (i = 0; i < 5 && sam[i] != *s; i++) ;
    /* Če je samoglasnik, povečamo pripadajoči števec, sicer pa ga izpišemo. */
    if (i < 5) n[i]++; else fputc(*s, stdout); }
  /* Na koncu izpišemo še samoglasnike. */
  for (i = 0; i < 5; i++) for (j = 0; j < n[i]; j++) fputc(sam[i], stdout);
  fputc('\n', stdout);
}
```

Razmislimo zdaj še o težji različici naloge, pri kateri bi radi iz besedila brez samoglasnikov rekonstruirali prvotno besedilo s samoglasniki. Naloga pravi, da imamo na voljo slovar besed (s samoglasniki) in njihove pogostosti v jeziku, v katerem je bilo napisano naše besedilo. Besedilo lahko poskusimo rekonstruirati tako, da za vsako besedo vhodnega besedila (torej brez samoglasnikov), recimo w, poiščemo v slovarju najpogostejšo tako besedo, iz katere po brisanju samoglasnikov nastane ravno w. Da bomo to počeli učinkovito, je koristno slovar predelati v razpršeno tabelo: za vsako besedo x našega slovarja (s pogostostjo f_x) pogledamo, kaj nastane iz nje po brisanju samoglasnikov; recimo tej besedi $d(x)$; in v razpršeno tabelo dodajmo ključ $d(x)$, kot spremljevalno vrednost pa x in f_x. Če se več različnih x preslika v isti $d(x)$, obdržimo med njimi v razpršeni tabeli le najpogostejšega (torej tistega z največjo f_x).

Ta rešitev je seveda nekoliko naivna, saj iz neke besede vhodnega besedila (brez samoglasnikov) vedno rekonstruira isto besedo: na primer, *dv* lahko nastane iz *davi*, *diva*, *dva*, *udav* in jasno je, da bo včasih pravilna ena od teh besed, včasih druga itd., naš postopek pa bi vedno vračal eno in isto (namreč tisto, ki je v slovarju označena kot najpogostejša).

Rešitev lahko poskusimo izboljšati, če upoštevamo tudi pogostost parov besed. Naloga pravi, da imamo na voljo veliko zbirko besedil; recimo, da se v njej beseda w pojavi $f(w)$-krat, par besed wx (torej w, ki mu sledi x) pa $f(wx)$-krat; razmerje $f(wx)/f(w)$ lahko potem gledamo kot verjetnost, da bo naslednja beseda x, če je bila prejšnja beseda w; to verjetnost bomo označili s $P(x|w)$. Podobno lahko ocenimo tudi verjetnost besede same po sebi; to je $P(w) = f(w)/D$, če je D dolžina (merjena s številom besed) celotne zbirke besedil.

Recimo zdaj, da je naše vhodno zaporedje dolgo n besed (brez samoglasnikov): $w_1, w_2, \ldots, w_n$. Beseda w_i je morala nastati z brisanjem samoglasnikov iz neke be-

sede x_i; v okviru te omejitve pa bi radi poiskali takšno zaporedje $x_1, x_2, \ldots, x_n$, ki bo najverjetnejše, torej ki ima največji produkt $P(x_1)P(x_2|x_1)P(x_3|x_2)\ldots P(x_n|x_{n-1})$. Lažje kot maksimizirati produkt je maksimizirati njegov logaritem (rezultat pa je enak, kajti kjer je produkt največji, je tudi njegov logaritem največji), torej vsoto $\log P(x_1) + \sum_{i=2}^{n} \log P(x_i|x_{i-1})$.

Ta problem lahko rešimo z dinamičnim programiranjem: naj bo $g(k, x_k)$ najmanjša vsota oblike $\sum_{i=k+1}^{n} \log P(x_i|x_{i-1})$, torej po vseh možnih izborih nadaljnjih besed $x_{k+1}, \ldots, x_n$. Opazimo lahko, da je $g(k, x_k) = \min_{x_{k+1}}(\log P(x_{k+1}|x_k) + g(k+1, x_{k+1}))$, pri čemer gre minimum po vseh takih x_{k+1}, ki ob brisanju samoglasnikov dajo niz w_{k+1}. Vrednosti $g(k, x_k)$ lahko računamo sistematično po padajočem k (in pri vsakem k po vseh tistih x_k, ki dajo ob brisanju samoglasnikov niz w_k). Na koncu moramo le še izračunati $\min_{x_1}(\log P(x_1) + g(1, x_1))$ in to je rezultat, ki ga iščemo. Če si ob vsakem računanju minimuma zapomnimo še, pri katerem x_k smo ga dosegli, bomo s temi podatki lahko na koncu tudi rekonstruirali konkretno zaporedje $x_1, \ldots, x_n$, ki pripelje do največje vsote.[14]

Rešitev lahko na razne načine še izboljšamo. (1) Namesto parov bi lahko gledali daljše fraze in ocenjevali npr. verjetnost besede glede na dve prejšnji, ne le na eno prejšnjo. (2) Če se nek par wx v zbirki besedil ne pojavlja, bomo dobili $f(wx) = 0$ in zato $P(x|w) = 0$, kar v bistvu pomeni, da razglasimo nek par za nemogoč že samo zato, ker se ni pojavil v naši zbirki besedil; bolje bi bilo pri našem računanju verjetnosti prišteti vsaki pogostosti $f(wx)$ neko majhno pozitivno konstanto, tako da se izognemo ničelnim verjetnostim. (3) Paziti bi morali še na možnost, da se besede iz samih samoglasnikov pri brisanju samoglasnikov popolnoma izgubijo in ne pustijo nobenih sledov v besedilu, ki smo ga dobili kot vhodni podatek. Torej bi morali dovoliti, da ima rekonstruirano zaporedje več besed kot prvotno, pri čemer bi prišle v poštev za dodatne vrinjene besede le tiste, ki so iz samih samoglasnikov. (4) Namesto na nivoju besed bi lahko podoben razmislek naredili na nivoju črk (in torej ocenjevali verjetnost naslednje črke v odvisnosti od prejšnjih nekaj).

6. Kemijske formule

Nalogo lahko rešimo z rekurzijo. Naš podprogram bere formulo od leve proti desni in postopoma v nekem seznamu ali razpršeni tabeli računa število atomov vsakega elementa. Če pri branju formule naletimo na simbol nekega elementa, preberemo še morebitno število za njim in nato ustrezno povečamo števec atomov tega elementa v našem seznamu ali razpršeni tabeli. Če pa pri branju formule naletimo na oklepaj, izvedemo rekurzivni klic našega podprograma; ta nam vrne seznam, v katerem za vsak atom znotraj oklepajev piše, kolikokrat se tam pojavi. Nato moramo le še prebrati število za zaklepajem, pomnožiti z njim število pojavitev vsakega atoma v oklepajih in nato te pojavitve prišteti našemu seznamu.

Oglejmo si primer takšne rešitve v pythonu. Naš rekurzivni podprogram `PrestejAtome` dobi poleg niza `s` kot parameter še trenutni položaj `i`, do katerega smo prišli pri branju niza pred tem klicem; kot rezultat pa vrne razpršeno tabelo s številom pojavitev vsakega elementa ter položaj, do katerega je prišel v nizu `s` ob vrnitvi iz klica.

[14] Postopek, ki smo ga opisali tukaj, je pravzaprav poseben primer znanega Viterbijevega algoritma za rekonstrukcijo notranjih stanj skritega markovskega modela; za več o tem gl. npr. Wikipedijo *s. v.* Viterbi algorithm in tam citirano literaturo.

```
def PrestejAtome(s, i):
    h = {}
    while i < len(s):
        # Če smo dosegli zaklepaj, se moramo vrniti iz rekurzivnega klica.
        if s[i] == ')': i += 1; break
        if s[i] == '(':
            # Del formule v oklepajih obdelajmo z rekurzivnim klicem.
            h2, i = PrestejAtome(s, i + 1)
            # Indeks i zdaj kaže na prvi znak za zaklepajem.
        else:
            # Preberimo simbol elementa.
            j = i; i += 1
            while i < len(s) and s[i].islower(): i += 1
            h2 = { s[j:i]: 1 }
        # Preberimo še število za zaklepajem oz. simbolom elementa.
        j = i
        while i < len(s) and s[i].isdigit(): i += 1
        # Če števila ni, velja, kot da bi bilo število 1.
        n = int(s[j:i]) if i > j else 1
        # Ustrezno povečajmo število pojavitev elementov v našem slovarju h.
        for simbol in h2:
            h[simbol] = h.get(simbol, 0) + n * h2[simbol]
    # Vrnimo slovar in končni položaj.
    return h, i

def IzpisiAtome(s):
    h, i = PrestejAtome(s, 0)
    for simbol in sorted(h): print("%s %d" % (simbol, h[simbol]))
```

Rekurzijo smo pravzaprav potrebovali zato, ker med tem, ko obdelujemo tisti del formule, ki je zapisan med oklepaji, še ne vemo, s kakšnim faktorjem bomo morali na koncu pomnožiti atome, ki nastopajo v njem — to bomo vedeli šele takrat, ko bomo prebrali število za zaklepajem. Temu se lahko zelo elegantno izognemo, če beremo formulo od desne proti levi namesto od leve proti desni. Tako bomo vedno najprej dobili število pojavitev nečesa, šele nato pa tisto nekaj (to pa je lahko bodisi simbol nekega elementa bodisi del formule v oklepajih).

Znotraj oklepajev se seveda lahko pojavlja več simbolov (in celo vgnezdeni oklepaji) in faktor množenja, ki smo ga prebrali za zaklepajem, velja za vse te simbole, zato si ga moramo zapomniti v neki spremenljivki: na primer, v $(CN)_6$ imamo ne le 6 atomov N, ampak tudi 6 atomov C.

Pri gnezdenju oklepajev bo morala ta spremenljivka vsebovati zmnožek faktorjev, ki jih prispevajo vsi trenutno odprti oklepaji: na primer, ko se v $(Fe(CN)_6)_2$ ukvarjamo s C in N, moramo imeti pri roki podatek, da se tadva atoma pojavljata po $6 \cdot 2 = 12$-krat. Po drugi strani pa, ko pridemo do oklepaja pred C, moramo iti s faktorja 12 nazaj na faktor 2, saj se Fe pojavlja le 2-krat. Torej je koristno, če si poleg trenutnega faktorja zapomnimo še prejšnjega in tako naprej. To bomo najlažje naredili, če bomo faktorje hranili na skladu; pri zaklepaju dodamo nov faktor na vrh sklada, pri oklepaju pa faktor z vrha sklada pobrišemo.

```
def PrestejAtome2(s):
    h = {}; sklad = [1]
    i = len(s) - 1
    while i >= 0:
```

```
    # Pri oklepaju le pobrišemo število z vrha sklada.
    if s[i] == '(': i -= 1; sklad.pop(); continue

    # Sicer na tem mestu pričakujemo število; preberimo ga in ga dodajmo na sklad.
    n = 1 # če števila ni, si mislimo 1
    if s[i].isdigit():
      j = i + 1
      while i > 0 and s[i - 1].isdigit(): i -= 1
      n = int(s[i:j]); i -= 1
    sklad.append(n * sklad[-1])

    # Če imamo zaklepaj, ni treba narediti ničesar drugega.
    if s[i] == ')': i -= 1; continue

    # Sicer pa je tu simbol nekega elementa.
    j = i + 1
    while i > 0 and s[i].islower(): i -= 1 # poiščimo veliko črko, s katero se začne ta simbol
    simbol = s[i:j]; i -= 1
    h[simbol] = h.get(simbol, 0) + sklad.pop() # dodajmo ga v slovar h

  return h
```

Bolj za šalo kot zares pa si oglejmo še naslednjo dekadentno rešitev. Delovanje rekurzivne rešitve (funkcija PrestejAtome zgoraj) lahko simuliramo tudi z iteracijo: pri oklepaju dodamo na sklad nov slovar (ki je ekvivalenten tistemu, ki ga ima rekurzivna rešitev kot lokalno spremenljivko h v rekurzivnem klicu); pri simbolu le povečamo števec pojavitev tega elementa v slovarju na vrhu sklada; pri zaklepaju pa vzamemo slovar z vrha sklada in njegove atome preselimo v slovar pod njim, pri čemer jih pomnožimo s faktorjem, ki sledi zaklepaju. Te tri operacije v spodnji rešitvi izvajajo lokalne funkcije simbol, oklepaj in zaklepaj. Našo kemijsko formulo lahko z nekaj dobro merjenimi regularnimi izrazi predelamo v niz veljavne pythonovske izvorne kode, ki na primeren način kliče te tri funkcije; na primer, iz $\mathrm{Fe(CN)_6)_2}$ bi v spodnji rešitvi nastal niz ; simbol('Fe', 1); oklepaj(); simbol('C', 1); simbol('N', 1); zaklepaj(6); zaklepaj(2). Tega na koncu izvedemo s pythonovo funkcijo exec.

```
def PrestejAtome3(s):
  def simbol(s, n): sklad[-1][s] = sklad[-1].get(s, 0) + n; return 0
  def oklepaj(): sklad.append({})
  def zaklepaj(n): sum(simbol(s, c * n) for (s, c) in sklad.pop().items())
  import re
  s = re.sub(r"([A-Z][a-z]*|\))($|(?=[^0-9]))", r"\1$", s).replace("$", "1")
  s = s.replace("(", "<").replace(")", ">")
  s = re.sub(r"([A-Z][a-z]*)([0-9]*)", r"; simbol('\1', \2)", s)
  s = re.sub(r">([0-9]*)", r"; zaklepaj(\1)", s).replace("<", "; oklepaj()")
  sklad = [{}]; exec("1" + s); return sklad[0]
```

Funkciji exec se je sicer v splošnem bolje izogibati, ker je možen vir varnostnih lukenj (zloben uporabnik nam lahko v izvajanje podtakne niz, ki ima kakšne za nas neugodne stranske učinke).

7. Branje luknjanega traku

Vsak znak je predstavljen s petimi biti; naloga pravi, da se lahko naši podatki začnejo nekje na sredi neke take peterice bitov. Torej je mogoče, da prvih nekaj bitov (največ štirje biti) pripada neki peterici, ki je berljiva le delno in zanjo torej

ne moremo ugotoviti, kateri znak je prvotno predstavljala. Zato moramo te bite preskočiti in začeti z dekodiranjem šele za njimi. Ker ne vemo, koliko bitov moramo preskočiti, bomo v zanki preizkusili vse možnosti (ena od možnosti je seveda tudi ta, da ne preskočimo ničesar, ker se naše zaporedje bitov začne ravno na meji med dvema znakoma). Spodnji program ima v ta namen zanko po p od 0 do 4.

Pri vsakem p se potem v gnezdeni zanki (po i) sprehajamo po preostanku vhodnega zaporedja bitov; vsak bit dodamo v spremenljivko x, v kateri se s tem postopoma nabira koda naslednjega znaka. Ko se nam v x nabere pet bitov, znak dekodiramo in izpišemo, x pa postavimo na 0, da bomo pripravljeni na dekodiranje naslednjega znaka.

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

extern char Znak(int x);

int main()
{
  char s[MaxN + 1]; int p, i, n, x;

  /* Preberimo vhodno zaporedje bitov. */
  gets(s); n = strlen(s);

  /* Prvih nekaj bitov (največ 4) lahko preskočimo. */
  for (p = 0; p < 5; p++) {

    /* Dekodirajmo preostanek zaporedja, od p-tega bita naprej. */
    for (i = 0, x = 0; p + i < n; i++) {
      /* Dodajmo trenutni bit v spremenljivko x. */
      x = (x << 1) | (s[p + i] == '1' ? 1 : 0);
      /* Ko se nam nabere 5 bitov, lahko x dekodiramo in izpišemo. */
      if (i % 5 == 4) putc(Znak(x), stdout), x = 0; }

    /* Izpišimo še znak za konec vrstice. */
    putc('\n', stdout); }

  return 0;
}
```

8. Pari besed

Ker naloga pravi, naj naš postopek deluje tudi v primerih, ko je vhodno besedilo zelo dolgo, je bolje, če ne preberemo celega naenkrat v pomnilnik, ampak ga beremo po delih oz. kar po besedah. Poleg trenutne besede si zapomnimo še prejšnjo, tako da obe skupaj tvorita par, po kakršnih sprašuje naša naloga. Pare bomo dodajali v razpršeno tabelo (*hash table*),[15] pri čemer kot spremljevalni podatek ob vsakem paru hranimo število pojavitev tega para v doslej prebranem besedilu. Ko par prvič dodamo v razpršeno tabelo, postavimo njegov števec pojavitev na 1, ob vsaki nadaljnji pojavitvi tega para pa povečamo njegov števec za 1. Ko pridemo do konca besedila, se moramo le še sprehoditi po vseh parih v naši razpršeni tabeli in izpisati tiste, pri katerih je števec pojavitev večji ali enak 2.

[15] Namesto razpršene tabele bi lahko uporabili tudi kakšno drugo podatkovno strukturo, npr. kar navadno tabelo ali seznam, vendar bi bil naš postopek potem zelo neučinkovit, ker bi porabil veliko časa za iskanje in/ali za dodajanje parov. Pri razpršeni tabeli je mogoče obe operaciji, iskanje in dodajanje, izvesti v $O(1)$ časa.

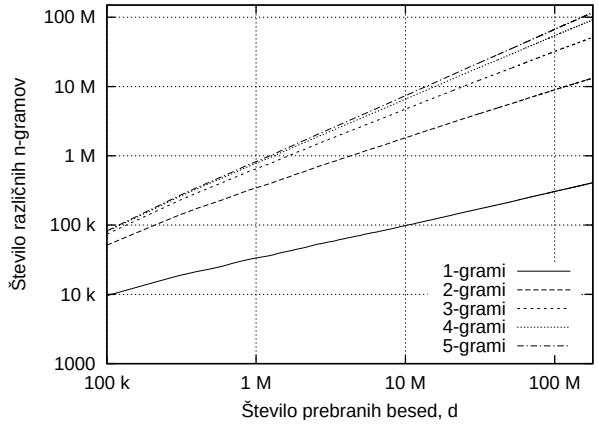
Če bi hoteli to rešitev napisati v C-ju, bi imeli nekaj več dela, ker nimamo pri roki razpršene tabele. Zato rešimo nalogo raje v C++11, kjer je razpršena tabela del standardne knjižnice (razred unordered_map):

```
#include <iostream>
#include <fstream>
#include <string>
#include <unordered_map>
using namespace std;

int main()
{
  ifstream f("besedilo.txt");
  unordered_map<string, int> pari;
  bool prva = true; string beseda, prejsnja;

  while (true)
  {
    // Zapomnimo si prejšnjo besedo.
    prejsnja = beseda;

    // Preberimo naslednjo besedo.
    f >> beseda; if (f.fail()) break;
    if (prva) { prva = false; continue; }

    // Sestavimo par iz prejšnje in trenutne besede.
    string par = prejsnja + " " + beseda;

    // Poglejmo, ali je že prisoten v razpršeni tabeli.
    auto it = pari.find(par);

    // Če je že v tabeli, mu povečajmo število pojavitev, sicer pa ga dodajmo.
    if (it != pari.end()) it->second++;
    else pari.insert(unordered_map<string, int>::value_type(par, 1));
  }

  // Izpišimo vse pare z dovolj velikim številom pojavitev.
  const int k = 2;
  for (auto it = pari.begin(); it != pari.end(); ++it)
    if (it->second >= k) cout << it->first << endl;
}
```

Za poskus smo pognali ta postopek na zbirki približno 800 000 Reutersovih člankov s skupaj približno 180 milijoni besed. V zbirki se je pojavilo približno 13,2 milijona različnih parov besed, od tega jih je imelo 6,0 milijonov po vsaj dve pojavitvi.

Razmislimo še o splošnejši različici naloge, pri kateri dobimo n in k, zanimajo pa nas vsi n-grami z vsaj k pojavitvami (takim n-gramom bomo rekli, da so *pogosti*). Naše gornje rešitve ne bi bilo težko posplošiti na poljubna n in k; število pojavitev, k, imamo zgoraj kot konstanto z vrednostjo 2, zdaj pa bi ga morali dobiti kot parameter. Namesto samo ene prejšnje besede pa bi si morali zapomniti zadnjih $n-1$ besed pred trenutno, da bi lahko iz njih sestavili trenutni n-gram; takšne n-grame bi potem zlagali v razpršeno tabelo in šteli njihove pojavitve na enak način, kot smo zgoraj počeli s pari.

Slabost te rešitve je, da je pri večjih n in k vseh različnih n-gramov zelo veliko, le majhen delež pa jih ima vsaj k pojavitev; zato naša rešitev porabi veliko pomnilnika za shranjevanje n-gramov, ki nas na koncu v resnici ne bodo zanimali (ker niso dovolj pogosti). Na primer, na isti zbirki besedil kot zgoraj je kar 117 milijonov različnih



Graf kaže, kako je pri našem poskusu naraščalo število različnih n-gramov, $v_n(d)$, v odvisnosti od števila prebranih vhodnih besed (d).

5-gramov, od tega jih ima 23 milijonov vsaj dve pojavitvi, samo 838 tisoč pa jih ima vsaj 10 pojavitev. Pri $n = 5$, $k = 10$ bi torej naša rešitev hranila v razpršeni tabeli kar 117 milijonov n-gramov, čeprav je med njimi manj kot en odstotek dovolj pogostih, da jih bomo morali na koncu izpisati. Naslednja tabela za $n = 1, \ldots, 5$ pove, koliko je različnih n-gramov in koliko izmed njih se pojavi vsaj dvakrat ali vsaj desetkrat (v oklepajih so deleži glede na vse n-grame):

		Št. različnih n-gramov z vsaj k pojavitvami		
n	c_n	$k = 1$	$k = 2$	$k = 10$
1	0,49	410 k	269 k (66 %)	116 k (28 %)
2	0,69	13,2 M	6,07 M (45 %)	1,32 M (10 %)
3	0,83	51,4 M	17,1 M (33 %)	1,98 M (4 %)
4	0,92	91,8 M	22,9 M (25 %)	1,41 M (1,5 %)
5	0,96	117 M	23,2 M (20 %)	838 k (0,7 %)

To opažanje velja tudi bolj na splošno. Recimo, da smo že prebrali d besed našega vhodnega besedila (zgoraj smo videli, da gre v naši zbirki d do 180 milijonov) in pri tem opazili $v_n(d)$ različnih n-gramov. V praksi se izkaže, da to število različnih n-gramov narašča približno kot $O(d^{c_n})$ za nek konstanten eksponent c_n (glej zgornji graf). Pri posameznih besedah, torej 1-gramih, je eksponent $c_1 \approx 1/2$ (to opažanje je znano tudi kot *Heapsov zakon*), pri večjih n pa je eksponent c_n večji in se hitro približa 1. To pomeni, da sicer število različnih besed v korpusu narašča približno kot $O(\sqrt{d})$, število daljših n-gramov (recimo od $n = 5$ naprej) pa že bolj kot $O(d)$. Naša dosedanja rešitev bi zato pri takšnih n porabila veliko pomnilnika, saj bi morala vseh teh $O(d)$ različnih n-gramov hraniti v razpršeni tabeli.

Do rešitve, ki porabi manj pomnilnika, lahko pridemo z naslednjim opažanjem: v n-gramu (skupini n zaporednih besed) $w_1 w_2 \ldots w_n$ se skrivata dva $(n-1)$-grama, namreč $w_1 w_2 \ldots w_{n-1}$ in $w_2 w_3 \ldots w_n$. Kjerkoli se pojavi $w_1 w_2 \ldots w_n$, se torej pojavita tudi tadva $(n-1)$-grama, zato je njuno število pojavitev na koncu vsaj tolikšno kot število pojavitev n-grama $w_1 w_2 \ldots w_n$. Če bi torej poznali število pojavitev vsakega $(n-1)$-grama, bi lahko, ko bi pri branju besedila prišli do n-grama $w_1 w_2 \ldots w_n$, najprej preverili, ali imata oba $(n-1)$-grama, $w_1 w_2 \ldots w_{n-1}$ in $w_2 w_3 \ldots w_n$, vsak

po vsaj k pojavitev; če jih nimata, potem vemo, da jih tudi $w_1w_2\ldots w_n$ nima, zato nam ga sploh ni treba dodajati v razpršeno tabelo. Tistim n-gramom, ki ta preizkus prestanejo (in jih torej dodamo v razpršeno tabelo), rečemo *kandidati* za pogoste n-grame; za nekatere med njimi se sicer na koncu še vedno lahko izkaže, da v resnici niso pogosti (torej imajo manj kot k pojavitev), vendar jih bo veliko manj kot pri naši prvotni rešitvi.

Pri tem postopku torej potrebujemo podatke o tem, kateri so dovolj pogosti $(n-1)$-grami, preden lahko začnemo iskati pogoste n-grame. Da poiščemo pogoste $(n-1)$-grame, pa seveda razmišljamo podobno in zato še prej potrebujemo pogoste $(n-2)$-grame. Tako bo torej naš postopek pravzaprav naredil n prehodov čez vhodno besedilo: pri prvem bo poiskal pogoste besede (torej tiste z vsaj k pojavitvami), v drugem prehodu pogoste pare, nato pogoste trojice besed in tako naprej. V vsakem prehodu si pomagamo s pogostimi $(n-1)$-grami iz prejšnjega prehoda, podatke o pogostosti še krajših fraz pa lahko sproti pozabljamo, saj jih v nadaljevanju ne bomo več potrebovali.

Spodnji program ima v ta namen dve razpršeni tabeli: ko obdeluje m-grame, ima v tabeli stari vse pogoste $(m-1)$-grame (in mogoče še nekatere, ki niso pogosti), v tabelo novi pa dodaja tiste m-grame, ki so prestali preizkus z $(m-1)$-grami (in pri katerih torej obstaja možnost, da se bodo na koncu izkazali za dovolj pogoste). Po vsakem prehodu čez podatke lahko staro tabelo pobrišemo in obe tabeli zamenjata vlogi.

```
void PogostiNGrami2(int n, int k)
{
  unordered_map<string, int> nGrami[2];
  vector<string> besede; // zadnjih nekaj prebranih besed

  for (int m = 1; m <= n; m++)
  {
    unordered_map<string, int> &stari = nGrami[(m - 1) % 2], &novi = nGrami[m % 2];
    novi.clear(); besede.push_back("");

    // V tabeli „stari“ so podatki o (m − 1)-gramih in njihovih pogostostih.
    // Tisti (m − 1)-grami, ki jih v tabeli „stari“ ni, imajo gotovo manj kot k pojavitev.
    // Zdaj bomo v tabeli „novi“ pripravili vse pogoste m-grame (in mogoče še nekatere,
    // ki se ne bodo izkazali za pogoste).
    ifstream f("besedilo.txt");
    for (int stBesede = 0; ; stBesede++)
    {
      f >> besede[stBesede % m]; if (f.fail()) break;
      // Ali smo že prebrali vsaj m besed?
      if (stBesede + 1 < m) continue;

      // V m-gramu iz zadnjih m besed se skrivata dva (m − 1)-grama, levi in desni;
      // njun presek pa je srednji (m − 2)-gram. Sestavimo te podnize.
      string levi, desni, srednji;
      unordered_map<string, int>::iterator it;
      if (m > 1) {
        for (int i = 1; i < m − 1; i++)
          srednji = besede[(stBesede − i) % m] + (i == 1 ? "" : " ") + srednji;
        levi = besede[(stBesede − (m − 1)) % m]; if (m > 2) levi += " " + srednji;
        desni = besede[stBesede % m]; if (m > 2) desni = srednji + " " + desni;

        // Ali sta oba (m − 1)-grama dovolj pogosta?
```

```
            it = stari.find(levi); if (it == stari.end() || it->second < k) continue;
            it = stari.find(desni); if (it == stari.end() || it->second < k) continue; }

          // Izračunajmo trenutni m-gram in ga dodajmo v razpršeno tabelo.
          string nGram = levi + (m > 1 ? " " : "") + besede[stBesede % m];
          it = novi.find(nGram);
          if (it != novi.end()) it->second++;
          else novi.insert(unordered_map<string, int>::value_type(nGram, 1));
        }

        // Če smo prišli do n-gramov, pogoste n-grame izpišimo.
        if (m == n) for (auto it = novi.begin(); it != novi.end(); ++it)
          if (it->second >= k) cout << it->first << endl;
      }
    }
```

Tudi to rešitev smo preizkusili na isti zbirki člankov kot prejšnjo. Rezultate kaže naslednja tabela; vidimo lahko, da je na primer pri $n = 5$, $k = 10$ prejšnja rešitev v razpršeni tabeli hranila 117 milijonov 5-gramov, naša nova rešitev pa le 3,17 milijona 5-gramov (od katerih se jih 838 tisoč izkaže za pogoste). Pri $k = 2$ je pogostih n-gramov več, zato je prihranek pomnilnika manjši, vendar še vedno precejšen.

Število različnih n-gramov

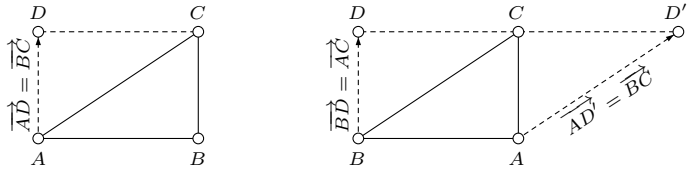
	$k = 1$	$k = 2$		$k = 10$	
n		kandidati	pogosti	kandidati	pogosti
1	410 k	410 k	269 k (66 %)	410 k	116 k (28 %)
2	13,2 M	13,0 M	6,07 M (47 %)	12,1 M	1,32 M (11 %)
3	51,4 M	38,6 M	17,1 M (44 %)	22,3 M	1,98 M (9 %)
4	91,8 M	41,9 M	22,9 M (55 %)	11,3 M	1,41 M (12 %)
5	117 M	31,9 M	23,2 M (73 %)	3,17 M	838 k (26 %)

9. Pravokotnik

Označimo oglišča našega pravokotnika po vrsti z A, B, C in D. Recimo, da so podane točke A, B in C, mi pa bi radi izračunali D. Pomagamo si lahko z dejstvom, da je premik iz A v D prav tak kot premik iz B v C (glej levo sliko spodaj), torej imamo

$$\overrightarrow{0D} = \overrightarrow{0A} + \overrightarrow{AD} = \overrightarrow{0A} + \overrightarrow{BC} = \overrightarrow{0A} + \overrightarrow{0C} - \overrightarrow{0B}.$$

Do enake rešitve bi seveda prišli tudi, če bi začeli v točki C in si pomagali z dejstvom, da je premik iz C v D enak premiku iz B v A (potem bi dobili $\overrightarrow{0D} = \overrightarrow{0C} + \overrightarrow{CD} = \overrightarrow{0C} + \overrightarrow{BA} = \overrightarrow{0C} + \overrightarrow{0A} - \overrightarrow{0B}$).



Neugodno pa je, da nam naloga nikjer ne zagotavlja, da si točke A, B in C sledijo prav v tem vrstnem redu, torej da A in C ležita diagonalno nasproti drug drugega, B pa je med njima. Lahko se na primer zgodi, da ležita diagonalno nasproti druga

druge točki B in C, točka A pa je med njima (glej desno sliko). V tem primeru bi nam zgornja formula rekonstruirala nekakšen paralelogram, ne pa pravokotnika (glej točko D' na desni sliki). Pravi rezultat dobimo tako, da začnemo v točki B in naredimo enak premik, kakršen bi nas iz A pripeljal v C:

$$\overrightarrow{0D} = \overrightarrow{0B} + \overrightarrow{BD} = \overrightarrow{0B} + \overrightarrow{AC} = \overrightarrow{0B} + \overrightarrow{0C} - \overrightarrow{0A}.$$

Obstaja še tretja možnost (ki je na sliki ni): da si ležita diagonalno nasproti točki A in B, točka C pa je med njima. V tem primeru skonstruiramo točko D tako, da začnemo v A in izvedemo enak premik, kakršen bi nas iz B pripeljal v C:

$$\overrightarrow{0D} = \overrightarrow{0A} + \overrightarrow{AD} = \overrightarrow{0A} + \overrightarrow{BC} = \overrightarrow{0A} + \overrightarrow{0C} - \overrightarrow{0B}.$$

Če primerjamo te tri formule, vidimo, da so si zelo podobne; v vseh nastopajo členi $\overrightarrow{0A}$, $\overrightarrow{0B}$, $\overrightarrow{0C}$, pri čemer seštejemo tista dva, ki predstavljata diagonalno nasprotni točki, nato pa od njiju odštejemo tretjega.

Kako ugotovimo, kateri dve od točk A, B in C si ležita diagonalno nasproti? Opazimo lahko, da ne glede na to, kako si izberemo tri oglišča pravokotnika, ta tri oglišča gotovo tvorijo pravokotni trikotnik; hipotenuza tega pravokotnega trikotnika pa je hkrati tudi diagonala našega pravokotnika. Lahko torej izračunamo dolžine vseh stranic trikotnika in poiščemo najdaljšo med njimi; tista je hipotenuza in njeni krajišči sta tisti, ki ju bomo morali v formuli za izračun D-ja sešteti. Preostalo oglišče pa je tisto, pri katerem ima trikotnik pravi kot, in tisto oglišče bomo morali v formuli za izračun D-ja odšteti.

```
void Pravokotnik(double Ax, double Ay, double Bx, double By, double Cx, double Cy,
                 double *Dx, double *Dy)
{
  /* Izračunajmo dolžine stranic trikotnika ABC. */
  double dAB = (Ax − Bx) * (Ax − Bx) + (Ay − By) * (Ay − By);
  double dAC = (Ax − Cx) * (Ax − Cx) + (Ay − Cy) * (Ay − Cy);
  double dBC = (Bx − Cx) * (Bx − Cx) + (By − Cy) * (By − Cy);

  /* Za vsako točko določimo, ali jo bomo morali v formuli za D
     prištevati ali odštevati. (Če leži točka pri pravem kotu, nasproti
     hipotenuze, jo odštevamo, sicer pa jo prištevamo.) */
  int sA = 1, sB = 1, sC = 1;
  if (dBC >= dAB && dBC >= dAC) sA = −1;
  else if (dAC >= dAB && dAC >= dBC) sB = −1;
  else /* if (dAB >= dAC && dAB >= dBC) */ sC = −1;

  /* Zdaj lahko izračunamo koordinate točke D. */
  *Dx = sA * Ax + sB * Bx + sC * Cx;
  *Dy = sA * Ay + sB * By + sC * Cy;
}
```

Nalogo bi lahko rešili tudi drugače; na primer, pravi kót v trikotniku lahko iščemo s skalarnim produktom (če je pri B pravi kot, potem je $\overrightarrow{BA} \cdot \overrightarrow{BC} = 0$ ipd.). Ko enkrat vemo, kje je pravi kot, tudi vemo, da moramo tisto točko v formuli za $\overrightarrow{0D}$ odšteti, ostali dve pa prišteti.

10. Pismo iz zapora

(a) Naš podprogram gre v zanki po znakih vhodnega pisma in jih kopira v izhodni niz. Pri tem v spremenljivki k šteje, koliko znakov je že preskočil; ko ta števec doseže

(ali preseže) a, vemo, da moramo ob prvi priložnosti v izhodni niz vpisati naslednji znak skritega sporočila. Pri tem moramo le preveriti, da še nismo prišli do konca sporočila, da je trenutni znak pisma črka in da je ta črka različna od trenutne črke sporočila. Ko v izhodni niz izpišemo znak sporočila (namesto znaka iz vhodnega pisma), postavimo k spet na 0, s čimer zagotovimo, da naslednje črke sporočila ne bomo izpisali prej kot čez a znakov.

Če pridemo do konca pisma, ne da bi prišli tudi do konca sporočila, potem vemo, da sporočila v to pismo ni bilo mogoče skriti (ker je pismo prekratko oz. sporočilo predolgo).

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

void Sifriraj(const char *pismo, const char *sporocilo, int a)
{
  int i, k, n = strlen(pismo);
  char *izhod = (char *) malloc(n + 1);

  /* Sifrirajmo sporočilo. */
  for (i = 0, k = 0; i <= n; i++)
    if (k >= a && *sporocilo && isalpha(pismo[i]) && pismo[i] != *sporocilo)
      izhod[i] = *sporocilo++, k = 0;
    else
      izhod[i] = pismo[i], k++;

  /* Izpišimo rezultat. */
  if (*sporocilo) printf("Sporocila ni mogoce skriti :-(\n");
  else printf("%s\n", izhod);

  free(izhod); /* Pospravimo za sabo. */
}
```

(b) Preden vpišemo naslednji znak skritega sporočila v izhodni niz, moramo preveriti, ali je istoležni znak prvotnega pisma velika črka ali mala; odvisno od tega potem tudi znak skritega sporočila spremenimo v veliko oz. malo črko. Začetni del prvega stavka **if** v gornjem podprogramu lahko torej spremenimo v:

```
if (k >= a && *sporocilo && isalpha(pismo[i]) &&
    toupper(pismo[i]) != toupper(*sporocilo))
{
  izhod[i] = isupper(pismo[i]) ? toupper(*sporocilo) : tolower(*sporocilo);
  sporocilo++; k = 0;
}
```

(c) V zanki se bomo istočasno premikali po obeh vhodnih nizih. Ko preskočimo že vsaj a znakov (to štejemo s spremenljivko k, podobno kot zgoraj pri šifriranju), počakamo na prvi tak primer, kjer ima prvotno pismo črko in je istoležni znak izhodnega niza drugačen; takrat vemo, da smo na mestu, kjer je šifrirni postopek povozil znak prvotnega niza z naslednjim znakom skritega sporočila, zato moramo ta znak izpisati.

```
void Desifriraj(const char *pismo, const char *izhod, int a)
{
  int k = 0;
```

```
  while (*pismo)
  {
    if (k >= a && isalpha(*pismo) && *pismo != *izhod)
      fputc(*izhod, stdout), k = 0;
    else k++;
    pismo++; izhod++;
  }
  fputc('\n', stdout);
}
```

(*d*) Postopek za šifriranje, kot je opisan v tej nalogi, žal ne omogoča enoličnega dešifriranja, če ne poznamo prvotnega pisma. Ena težava je, da ne moremo zanesljivo ugotoviti, kje se skrito sporočilo konča; druga težava pa je, da če ne poznamo prvotnega pisma, tudi ne vemo, kje se je med šifriranjem zgodilo, da je bila črka pisma enaka črki skrivnega sporočila (in je zato šifrirni postopek preskočil v vhodnem pismu še več kot a znakov). Na primer:

Pismo pred šifriranjem	Skrito sporočilo	a	Niz po šifriranju
`xxyxxy`	`yy`	1	`xyyyxy`
`xxyxxx`	`yyy`	1	`xyyyxy`
`xyxyyy`	`yx`	1	`xyyyxy`

Če bi torej dobili $a = 1$ in `xyyyxy` kot niz po šifriranju, ne bi mogli ugotoviti, katerega od nizov `yy`, `yyy`, `yx` (ali `y`, ki ga sicer v gornji tabeli ni) je imel naš zapornik v mislih kot skrito sporočilo.

Rešitev postane enolična, če si lahko privoščimo nekaj dodatnih predpostavk, na primer to, da se pri šifriranju nikoli ni zgodilo, da bi bil trenutni znak prvotnega pisma enak trenutnemu znaku skritega sporočila (in bi zato moral šifrirni postopek preskočiti še en znak več kot sicer), in to, da je skrito sporočilo maksimalno dolgo. Potem lahko dešifriramo tako, da gremo v zanki po znakih vhodnega niza in jih preskakujemo; ko jih preskočimo a, izpišemo prvo črko, na katero naletimo. To ponavljamo, dokler ne pridemo do konca niza. Podobno kot pri šifriranju imamo tudi tu spremenljivko `k`, s katero štejemo, koliko znakov smo preskočili.

```
void Desifriraj2(int a, const char *s)
{
  int k = 0;
  while (*s)
    if (k >= a && isalpha(*s))
      fputc(*s++, stdout), k = 0;
    else k++, s++;
  fputc('\n', stdout);
}
```

(*e*) Nalogo lahko rešimo z rekurzijo. Za začetek preskočimo prvih a znakov šifriranega niza in še morebitne ne-črkovne znake, dokler ne dosežemo kakšne črke; tej črki recimo c.

Možno je, da je ta črka prišla v naš niz iz skritega sporočila; torej je bila v prvotnem pismu tu neka druga črka, recimo x, in je šifrirni postopek čeznjo napisal črko c iz skritega sporočila.

Druga možnost pa je, da je ta črka prišla v naš niz še iz prvotnega pisma, ker jo je šifrirni postopek pri šifriranju preskočil. To je mogoče le v primeru, če je enaka naslednji črki skritega sporočila. Torej je imelo prvotno pismo tukaj enega ali več c-jev in potem prej ali slej neko črko x, ki je različna od c, tako da je lahko šifrirni postopek čeznjo zapisal c iz skritega sporočila.

prvotno pismo	$\ldots x \ldots$	$\ldots yx \ldots$	$\ldots zyx \ldots$
niz po šifriranju	$\ldots c \ldots$	$\ldots yc \ldots$	$\ldots zyc \ldots$
pogoj, da je to mogoče	$x \neq c$	$y = c$ in $x \neq c$	$z = c$ in $y = c$ in $x \neq c$
torej imamo po šifriranju v resnici	$\ldots c \ldots$	$\ldots cc \ldots$	$\ldots ccc \ldots$

Pogledati moramo torej strnjeno skupino c-jev na trenutnem mestu našega niza, ki je nastal po šifriranju; za vsakega od njih je mogoče, da je prav on tisti, ki je v niz prišel iz skritega sporočila namesto iz prvotnega pisma. Za vsakega od njih moramo torej izvesti rekurzivni klic, ki bo nadaljeval z dešifriranjem pri naslednjem znaku za tistim c-jem.

Ker tudi ne vemo, kje se skrito sporočilo konča, ga moramo izpisati vsakič, ko vanj dodamo kakšno črko, saj je mogoče, da se konča prav pri njej. Tako dobimo naslednjo rekurzivno rešitev; parameter izhod ne kaže na začetek niza, ki je nastal s šifriranjem, ampak na tisto mesto v njem, kjer mora trenutni rekurzivni klic nadaljevati z dešifriranjem. Parameter sporocilo kaže na tabelo, v kateri postopoma sestavljamo skrito sporočilo, ki ga luščimo iz niza izhod; indeks si pove, kam v tej tabeli moramo shraniti naslednji znak skritega sporočila.

```
void Rekurzija(const char *izhod, char *sporocilo, int si, int a)
{
  int k; char c;

  /* Preskočimo a znakov sporočila in se premaknimo do naslednje črke. */
  k = 0; while (k < a && izhod[k]) k++;
  if (k < a) return;
  while (izhod[k] && ! isalpha(izhod[k])) k++;
  if (! izhod[k]) return;

  /* Trenutna črka izhoda je vsekakor tudi trenutna črka sporočila.
     Če imamo v vhodu tu več pojavitev te črke, pa je mogoče, da so bile
     nekatere prisotne že v prvotnem pismu in smo jih takrat ob šifriranju preskočili. */
  c = izhod[k++]; sporocilo[si++] = c;
  sporocilo[si] = 0; printf("%s\n", sporocilo);
  do { Rekurzija(izhod + k, sporocilo, si, a); k++; } while (izhod[k] == c);
}
```

Glavni podprogram za dešifriranje mora le alocirati primerno velik blok pomnilnika za shranjevanje skritega sporočila, nato pa začne z rekurzijo:

```
void DesifrirajVse(const char *izhod, int a)
{
  int n = strlen(izhod);
  char *sporocilo = (char *) malloc(n / (a + 1) + 1);
  Rekurzija(izhod, sporocilo, 0, a);
  free(sporocilo);
}
```

11. Knjige

(*a*) Razpored si lahko pripravimo v tabeli, v katero po vrsti vpisujemo knjige od 1 do n. Postopek razporejanja knjig lahko simuliramo čisto sistematično tako, kot je opisan v besedilu nalogi. Pomagamo si lahko s spremenljivko kam, ki nam pove, na katero stran bomo dali naslednjo knjigo; poleg tega pa imejmo še dva indeksa, levo in desno, ki nam povesta, na katero mesto v tabeli moramo vpisati naslednjo levo in naslednjo desno knjigo. Po tistem, ko vpišemo novo knjigo v tabelo, zamenjamo smer kam in popravimo ustrezni indeks (levo se povečuje za 1, desno pa zmanjšuje za 1).

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

void IzpisiA(int n)
{
  enum { Levo, Desno } kam;
  int *tabela, levo, desno, knjiga, i;

  /* Alocirajmo tabelo. */
  tabela = (int *) malloc(n * sizeof(int));

  /* Po vrsti vpišimo knjige v tabelo. */
  levo = 0; desno = n - 1;
  kam = Desno; /* Prva knjiga gre na desno. */
  for (knjiga = 1; knjiga <= n; knjiga++)
    if (kam == Levo) tabela[levo++] = knjiga, kam = Desno;
    else tabela[desno--] = knjiga, kam = Levo;

  /* Izpišimo dobljeni razpored knjig. */
  for (i = 0; i < n; i++)
    printf("%d%s", tabela[i], (i == n - 1) ? "\n" : " ");

  /* Pospravimo za sabo. */
  free(tabela);
}
```

(*b*) Nalogo lahko rešimo tudi brez tabele. Iz načina, kako razporejamo knjige po polici (prvo knjigo postavimo na desno in nato vsako knjigo postavimo z nasprotnega konca kot prejšnjo), sledi, da je naš razpored sestavljen tako, da so v njem najprej vsa soda števila v naraščajočem vrstnem redu, nato pa še vsa liha števila v padajočem vrstnem redu.

Sodih števil ni težko izpisovati; začnemo pri 2 in v zanki povečujemo števec po 2, dokler ne preseže n. Desni del razporeda pa bomo morali izpisovati v padajočem vrstnem redu, zato moramo razmisliti, pri katerem številu začeti. Če je n sod, smo ga že izpisali in moramo zdaj desni del razporeda začeti pri $n-1$; če pa je n lih, ga še nismo izpisali in moramo desni del razporeda začeti pri n.

```
void IzpisiB(int n)
{
  /* Izpišimo sode knjige v naraščajočem vrstnem redu. */
  int knjiga = 2;
  while (knjiga <= n) {
    printf("%d ", knjiga);
    knjiga += 2; }

  /* Pri katerem številu moramo začeti desni, lihi del razporeda? */
```

```
  if (n % 2 == 0) knjiga = n − 1;
  else knjiga = n;
  /* Izpišimo lihe knjige v padajočem vrstnem redu. */
  while (knjiga >= 1) {
    printf("%d%s", knjiga, (knjiga == 1) ? "\n" : " ");
    knjiga −= 2; }
}
```

(*c*) Oglejmo si, kako bi potekalo prestavljanje knjig na nekaj konkretnih primerih.

Začnimo na primer z $n = 12$:	2 4 6 8 10 12 11 9 7 5 3 1
Cilka si sposodi $x = 9$ in jo vrne na konec:	2 4 6 8 10 12 11 7 5 3 1 9
Knjiga 1 ni na pravem mestu in jo prestavimo:	2 4 6 8 10 12 11 7 5 3 9 1
Knjiga 2 je na pravem mestu; 3 pa ni in jo prestavimo:	2 4 6 8 10 12 11 7 5 9 3 1
Knjiga 4 je na pravem mestu; 5 pa ni in jo prestavimo:	2 4 6 8 10 12 11 7 9 5 3 1
Knjiga 6 je na pravem mestu; 7 pa ni in jo prestavimo:	2 4 6 8 10 12 11 9 7 5 3 1

Zdaj so vse knjige na pravem mestu; skupaj smo torej prestavili 4 knjige. Ta razmislek lahko posplošimo takole: če je x lih, bi morale biti knjige 1, 3, ..., $x - 2$ desno od knjige x; vendar pa so po tistem, ko je Cilka vrnila knjigo x na konec police, te knjige levo od x. Vsako od njih moramo prestaviti, kar je skupaj $(x-1)/2$ prestavljanj. Knjig s sodimi številkami pa ne prestavljamo, ker so že na začetku vse na pravih mestih in jih tudi kasnejša prestavljanja lihih knjig nič ne premikajo.

Razmislimo še o sodem x, recimo $x = 10$:

Cilka si sposodi $x = 10$ in jo vrne na konec:	2 4 6 8 12 11 9 7 5 3 1 10
Knjiga 1 ni na pravem mestu in jo prestavimo:	2 4 6 8 12 11 9 7 5 3 10 1
Knjiga 2 je na pravem mestu; 3 pa ni in jo prestavimo:	2 4 6 8 12 11 9 7 5 10 3 1
Knjiga 4 je na pravem mestu; 5 pa ni in jo prestavimo:	2 4 6 8 12 11 9 7 10 5 3 1
Knjiga 6 je na pravem mestu; 7 pa ni in jo prestavimo:	2 4 6 8 12 11 9 10 7 5 3 1
Knjiga 8 je na pravem mestu; 9 pa ni in jo prestavimo:	2 4 6 8 12 11 10 9 7 5 3 1
Knjiga 10 ni na pravem mestu in jo prestavimo:	2 4 6 8 10 12 11 9 7 5 3 1

Zdaj so vse knjige na pravem mestu; skupaj smo torej prestavili 6 knjig. Vidimo torej, da če je x sod, bi morale biti knjige 1, 3, ..., $x - 1$ desno od knjige x; vendar pa so po tistem, ko je Cilka vrnila knjigo x na konec police, te knjige levo od x. Vsako od njih moramo prestaviti, kar je skupaj $x/2$ prestavljanj. S tem pridemo pri prestavljanju že do knjige x, nato prestavimo tudi njo in po tistem se prestavljanje konča, ker so vse knjige na svojem mestu. Tako imamo skupaj $x/2 + 1$ prestavljanj.

Poseben primer nastopi, če je x sod in velja $x = n$; takrat knjige x ni treba prestaviti, kajti ko pridemo pri prestavljanju do x, smo pred tem že postavili vse druge knjige na pravo mesto, zato je neizogibno, da je tudi knjiga x na pravem mestu. Tedaj je torej potrebnih le $x/2$ prestavljanj namesto $x/2 + 1$.

(*d*) Podobno kot zgoraj si oglejmo najprej konkreten primer; vzemimo $n = 10$ in $i = 5$. To pomeni, da je nova knjiga po debelini med 5 in 6; stare knjige $6, \ldots, 10$ se torej spremenijo v $7, \ldots, 11$, nova knjiga pa dobi številko 6.

Vrstni red pred prihodom nove knjige:	2 4 6 8 10 9 7 5 3 1
Po prihodu nove knjige:	6 2 4 7 9 11 10 8 5 3 1
1 je na pravem mestu; prestavimo 2:	2 6 4 7 9 11 10 8 5 3 1
3 je na pravem mestu; prestavimo 4:	2 4 6 7 9 11 10 8 5 3 1

V nadaljevanju bi prestavili še 7, 8, 9, 10, s tem pa bi bilo urejanje končano. Če ta razmislek posplošimo, vidimo naslednje: liha števila do vključno i so na pravem mestu in jih ne prestavljamo; soda števila pod i moramo prestaviti, ker je bila nova knjiga $i+1$ na začetku levo od njih namesto desno od njih; in nato moramo prestaviti vsa števila od $i+2$ do n, kajti tem se je ob dodajanju nove knjige spremenila parnost (ker so se povečala za 1), tako da so bila soda števila na desni in liha na levi namesto obratno. Skupno število prestavljanj je torej $(i-1)/2+(n-i-1)$. Poseben primer je $i=n$, ko drugi člen odpade (števil od $i+2$ do n sploh ni, izraz $n-i-1$ pa ima vrednost -1 namesto 0, zato ga tedaj ne smemo upoštevati).

Razmislimo še o sodem i, recimo $i=6$:

Po prihodu nove knjige:	7 2 4 6 9 11 10 8 5 3 1
1 je na pravem mestu; prestavimo 2:	2 7 4 6 9 11 10 8 5 3 1
3 je na pravem mestu; prestavimo 4:	2 4 7 6 9 11 10 8 5 3 1
5 je na pravem mestu; prestavimo 6:	2 4 6 7 9 11 10 8 5 3 1

V nadaljevanju bi prestavili še 7, 8, 9, 10. V splošnem torej zdaj prestavljamo soda števila do vključno i, nato pa še vsa števila od $i+1$ do n. Skupno število prestavljanj je zdaj $i/2+(n-i)$.

(*e*) Z zanko se sprehodimo po knjigah (v takem vrstnem redu, kot jih Mirko prestavlja na polici, torej od 1 do n); pri vsaki knjigi se sprehodimo po tabeli polica, da vidimo, kje na polici se trenutno nahaja (spremenljivka kjeJe); nato še izračunajmo, kje bi se morala nahajati v pravem končnem vrstnem redu (kjeMoraBiti). Če je kjeJe < kjeMoraBiti, moramo knjige na indeksih od kjeJe + 1 do kjeMoraBiti premakniti za eno mesto v levo; podobno pa, če je kjeJe > kjeMoraBiti, moramo knjige na indeksih od kjeJe − 1 do kjeMoraBiti premakniti za eno mesto v desno. Na koncu še vpišemo trenutno knjigo (tisto, ki jo prestavljamo) na indeks kjeMoraBiti.

```
int KolikoPrestavljanj(int n, int *polica)
{
  int knjiga, kjeJe, kjeMoraBiti, nPrestavljanj = 0, i;
  for (knjiga = 1; knjiga <= n; knjiga++)
  {
    /* Poglejmo, kje na polici je trenutno ta knjiga. */
    kjeJe = 0; while (polica[kjeJe] != knjiga) kjeJe++;
    /* Izračunajmo indeks, na katerem bi morala biti. */
    if (knjiga % 2 == 0) kjeMoraBiti = knjiga / 2 - 1;
    else kjeMoraBiti = n - 1 - knjiga / 2;
    if (kjeJe == kjeMoraBiti) continue;
    /* Prestavimo jo na pravo mesto; knjige med njenim trenutnim položajem
       in pravim mestom se pri tem zamaknejo. */
    nPrestavljanj++;
    while (kjeJe < kjeMoraBiti) { polica[kjeJe] = polica[kjeJe + 1]; kjeJe++; }
    while (kjeJe > kjeMoraBiti) { polica[kjeJe] = polica[kjeJe - 1]; kjeJe--; }
    polica[kjeJe] = knjiga;
  }
  return nPrestavljanj;
}
```

Z vsako knjigo imamo torej $O(n)$ dela: najprej zato, da jo poiščemo na polici, nato pa še zaradi premikanja knjig med indeksoma kjeJe in kjeMoraBiti. Ker je knjig n, je časovna zahtevnost tega postopka skupno $O(n^2)$.

Razmislimo še o učinkovitejši rešitvi. Seznam knjig na polici lahko predelamo v uravnoteženo binarno drevo. Drevo ima po eno vozlišče za vsako knjigo, tako da je številka knjige (od 1 do n) hkrati tudi številka vozlišča. Za vsako vozlišče x imamo v neki tabeli številko njegovega starša $P[x]$ in obeh otrok (levega $L[x]$ in desnega $D[x]$, če ju ima), poleg tega pa še skupno število elementov (recimo $V[x]$) v celotnem poddrevesu, ki se začenja pri tem vozlišču. V binarnem iskalnem drevesu so elementi ponavadi urejeni naraščajoče, v našem primeru pa ne bo tako, temveč bodo knjige urejene v takem vrstnem redu, v kakršnem se pojavljajo na polici.

S pomočjo tega drevesa lahko v času $O(\log n)$ izračunamo, kje na polici se nahaja neka knjiga x: prešteti moramo, koliko knjig je levo od nje. To so za začetek vse knjige iz njenega levega poddrevesa (če ga ima); nato pa se lahko sprehajamo od x gor po drevesu (vse do korena); na vsakem koraku, če je trentutno vozlišče desni otrok svojega očeta, to pomeni, da so na polici ta oče in vse knjige v njegovem levem poddrevesu levo od naše knjige.

algoritem KjeJe(x):
  $r := 0$;
  if $L[x] \neq$ NIL **then** $r := r + V[L[x]]$;
  while $P[x] \neq$ NIL:
    if $x = D[P[x]]$ **and** $L[P[x]] \neq$ NIL
    then $r := r + V[L[P[x]]] + 1$;
    $x := P[x]$;
  return r;

Ker se zanka v vsaki iteraciji premakne za en korak gor po drevesu in ker je globina drevesa le $O(\log n)$, je časovna zahtevnost tega postopka le $O(\log n)$.

Ko je treba knjigo prestaviti na polici, lahko za začetek pobrišemo vozlišče x iz drevesa. To storimo kot običajno pri binarnih drevesih: če je x brez otrok, ga le pobrišemo; če ima enega otroka, postane ta zdaj neposredno otrok x-ovega starša (namesto pobrisanega vozlišča x); če pa ima dva otroka, lahko vzamemo najbolj levo vozlišče v desnem x-ovem poddrevesu in ga premaknemo na mesto vozlišča x.

Nato moramo vozlišče x vriniti v drevo na novi položaj; recimo, da smo izračunali, da mora biti levo od knjige x še r drugih knjig (v naši prvotni, manj učinoviti rešitvi smo to količino izračunali v spremenljivki kjeMoraBiti). Zdaj lahko začnemo pri korenu drevesa in se spuščamo po njem. Ko smo v vozlišču y in je v njegovem levem poddrevesu l elementov, lahko razmišljamo takole: če bi dodali x nekam v desno poddrevo vozlišča y, bi bili levo od njega vsi elementi levega poddrevesa, pa še vozlišče y samo; to je skupaj $l + 1$ elementov. Mi pa bi radi imeli r elementov levo od x; torej, če je $r < l + 1$, potem novega vozlišča x ne smemo dodati v y-ovo desno poddrevo, pač pa v levo; drugače pa ga bomo morali dodati v desno poddrevo (kajti če bi ga dodali v levo, bi bilo desno od njega največ l elementov). Nato se premaknemo v ustreznega od y-ovih otrok in nadaljujemo po enakem postopku. Ob premiku v desno poddrevo smo pustili na svoji levi vse elemente levega poddrevesa in še vozlišče y samo, zato moramo takrat r zmanjšati za $l + 1$. Postopek se ustavi, ko vidimo, da poddrevesa, v katerega bi se morali premakniti, sploh ni: takrat lahko x postane otrok trenutnega vozlišča y. Tako dobimo naslednji postopek:

$y :=$ koren drevesa;

while true:
  (* *Na tem mestu gotovo velja* $0 \le r \le V[y]$. *)
  $l := 0$; **if** $L[y] \neq$ NIL **then** $l := V[L[y]]$;
  if $l < r$:
    if $R[y] \neq$ NIL: $y := R[y]$; $r := r - l - 1$;
    else: dodaj x kot desnega otroka vozlišča y; **break**;
  else:
    if $L[y] \neq$ NIL: $y := L[y]$
    else: dodaj x kot levega otroka vozlišča y; **break**;

Vidimo lahko, da imamo tako pri brisanju vozlišča x na starem mestu kot pri njegovem dodajanju na novo mesto le $O(1)$ dela za vsak nivo drevesa, tako da je časovna zahtevnost takšnega premika knjige le $O(\log n)$.

Čeprav smo začeli z uravnoteženim drevesom, nam lahko zaradi dodajanj in brisanj sčasoma postane neuravnoteženo (torej so nekateri deli drevesa veliko globlji od drugih in časovna zahtevnost naših operacij mogoče ni več $O(\log n)$). Da to preprečimo, lahko drevo sproti uravnotežujemo z enakimi tehnikami, kot se uporabljajo na primer pri AVL-drevesih ali pri rdeče-črnih drevesih (to seveda med drugim pomeni, da moramo dosedanjim tabelam dodati še eno za globino poddreves ali pa za barvo vozlišč). Ta drevesa sicer tradicionalno nimajo podatka o številu vozlišč v vsakem poddrevesu (naša tabela V), vendar se lahko hitro prepričamo, da te tabele ni težko vzdrževati in popravljati tudi ob operacijah, ki skrbijo za ohranjanje uravnoteženosti drevesa. Tako smo prišli do postopka, ki ima res z vsako knjigo največ $O(\log n)$ dela, zato je časovna zahtevnost celotnega postopka $O(n \log n)$ namesto $O(n^2)$.

(*f*) Nalogo lahko rešimo s pomočjo tabele na zelo podoben način kot pri (*a*). Razlika je predvsem v tem, da spremenljivke `kam` ne smemo zamenjati po vsaki knjigi, ampak le po vsaki drugi. To bi lahko izvedli tako, da bi imeli še eno spremenljivko, ki bi nam štela, kdaj je treba naslednjič spremeniti smer; po vsaki spremembi smeri bi to spremenljivko postavili na 2, po vsaki izpisani knjigi pa bi jo zmanjšali na 1 in ko bi padla na 0, bi smer spremenili. Še lažje pa je, če opazimo, da do spremembe pride po vsaki knjigi z liho številko:

```
void IzpisiF(int n)
{
  enum { Levo, Desno } kam;
  int *tabela, levo, desno, knjiga, i;

  /* Alocirajmo tabelo. */
  tabela = (int *) malloc(n * sizeof(int));

  /* Po vrsti vpišimo knjige v tabelo. */
  levo = 0; desno = n - 1;
  kam = Desno;
  for (knjiga = 1; knjiga <= n; knjiga++) {
    if (kam == Levo) tabela[levo++] = knjiga;
    else tabela[desno--] = knjiga;
    if (knjiga % 2 == 1) kam = (kam == Levo) ? Desno : Levo; }

  /* Izpišimo dobljeni razpored knjig. */
  for (i = 0; i < n; i++)
    printf("%d%s", tabela[i], (i == n - 1) ? "\n" : " ");

  /* Pospravimo za sabo. */
```

```
  free(tabela);
}
```

Šlo bi tudi brez spremenljivke kam. Ker zamenjamo stran po vsaki drugi knjigi, pridemo po štirih knjigah nazaj na prvotno stran. Tako lahko hitro opazimo, da na desno stran police pridejo knjige s številkami oblike $4k$ in $4k+1$, na levo stran pa knjige s številkami oblike $4k+2$ in $4k+3$. Program mora torej le pogledati ostanek po deljenju spremenljivke knjiga s 4, pa bo vedel, na katero stran dati to knjigo:

```
  for (knjiga = 1; knjiga <= n; knjiga++)
    if (knjiga % 4 <= 1) tabela[desno--] = knjiga;
    else tabela[levo++] = knjiga;
```

Ta razmislek o ostankih po deljenju s 4 pride zelo prav tudi, če hočemo nalogo rešiti brez tabele. Naraščajoči del tabele lahko izpišemo tako, da gremo v zanki po vseh knjigah, vendar izpisujemo le tiste, ki dajo po deljenju s 4 ostanek 2 ali 3. Nato gremo še enkrat po vseh knjigah, vendar v padajočem vrstnem redu, in izpisujemo tiste, ki dajo po deljenju s 4 ostanek 0 ali 1:

```
void IzpisiF2(int n)
{
  int knjiga;
  /* Izpišimo naraščajoči del tabele. */
  for (knjiga = 1; knjiga <= n; knjiga++)
    if (knjiga % 4 > 1)
      printf("%d ", knjiga);
  /* Izpišimo padajoči del tabele. */
  for (knjiga = n; knjiga >= 1; knjiga--)
    if (knjiga % 4 <= 1)
      printf("%d%s", knjiga, (knjiga == 1) ? "\n" : " ");
}
```

12. Transfuzije

(a) Naš podprogram mora našteti tiste tipe krvi, ki so kompatibilne s prejemnikovo in ki jih imamo na zalogi. Lahko bi šli torej v zanki po vseh tipih krvi in pri vsakem preverjali, ali je kompatibilen s prejemnikovo in ali ga imamo na zalogi. Zanimivo vprašanje pri tej nalogi je predvsem to, kako preverjati, ali sta dva tipa kompatibilna ali ne. S pomočjo pravil v besedilu naloge lahko za vsak par tipov krvi preverimo kompatibilnost in rezultate predstavimo s takšno tabelo:

Prejemnik	Darovalec							
	0−	A−	B−	AB−	0+	A+	B+	AB+
0−	×							
A−	×	×						
B−	×		×					
AB−	×	×	×	×				
0+	×				×			
A+	×	×			×	×		
B+	×		×		×		×	
AB+	×	×	×	×	×	×	×	×

Križci označujejo kombinacije, pri katerih je transfuzija možna. Takšno tabelo lahko vključimo tudi v naš program:

```
const char *imena[8] = { "0-", "A-", "B-", "AB-", "0+", "A+", "B+", "AB+" };
const int mozna[8][8] = {
  { 1, 0, 0, 0, 0, 0, 0, 0 },
  { 1, 1, 0, 0, 0, 0, 0, 0 },
  { 1, 0, 1, 0, 0, 0, 0, 0 },
  { 1, 1, 1, 1, 0, 0, 0, 0 },
  { 1, 0, 0, 0, 1, 0, 0, 0 },
  { 1, 1, 0, 0, 1, 1, 0, 0 },
  { 1, 0, 1, 0, 1, 0, 1, 0 },
  { 1, 1, 1, 1, 1, 1, 1, 1 } };

void MozneTransfuzije(int tipPrejemnika, int zaloga[8])
{
  int tipDarovalca;
  for (tipDarovalca = 0; tipDarovalca < 8; tipDarovalca++)
    if (mozna[tipPrejemnika][tipDarovalca] && zaloga[tipDarovalca] > 0)
      printf("%s\n", imena[tipPrejemnika]);
}
```

Tipe smo torej v našem programu označili z indeksi od 0 do 7 v takem vrstnem redu, v kakršnem jih vidimo tudi v zgornji tabeli. Za lepši izpis imamo še tabelo `imena`, v kateri je za vsak tip niz z imenom tega tipa.

Z malo dodatnega razmisleka pa lahko nalogo rešimo zelo elegantno tudi brez tabele. Opazimo lahko, da ima naša tabela zelo sistematično zgradbo,[16] in se vprašamo, ali se ne bi dalo njenih elementov računati z nekakšno formulo, tako da tabele sploh ne bi potrebovali.

Števila od 0 do 7, s katerimi smo predstavili naših osem tipov krvi, nam pravzaprav zelo sistematično opisujejo vsak tip krvi. Tip u smo predstavili s trobitnim celim številom $u_2u_1u_0$ (torej tako, da je $u = 4u_2 + 2u_1 + u_0$), pri čemer nam bit u_0 pove, ali je v imenu tipa prisoten A, bit u_1 nam pove, ali je v imenu tipa prisoten B, bit u_2 pa nam pove, ali je v imenu tipa prisoten +.

Recimo zdaj, da imamo prejemnika $u = u_2u_1u_0$ in darovalca $v = v_2v_1v_0$. Kako bi po pravilih iz besedila naloge preverili, ali sta kompatibilna? Če je prejemnik u Rh D-negativen, darovalec v pa je Rh D-pozitiven, potem sta nekompatibilna; to je torej takrat, ko je bit v_2 prižgan, u_2 pa ugasnjen: $v_2 \wedge \bar{u}_2$. Podobno, če prejemnik nima tipa A (ker je bodisi 0 bodisi B, ne pa A ali AB), darovalec pa ga ima (ker je bodisi A ali AB, ne pa 0 ali B), potem sta nekompatibilna; to je torej takrat, ko je v_0 prižgan, u_0 pa ugasnjen: $v_0 \wedge \bar{u}_0$. Enak razmislek velja tudi, če A in B zamenjamo, kar nam dá še pogoj $v_1 \wedge \bar{u}_1$.

Tako smo dobili tri (zelo podobne) pogoje za nekompatibilnost prejemnika in darovalca. Kompatibilna sta le v primeru, če ni izpolnjen nobeden od teh treh

[16] Zgornja desna četrtina tabele je prazna, ostale tri četrtine pa so si med seboj enake. Če pogledamo vsako od treh nepraznih četrtin, zanjo velja enako: tudi *njena* zgornja desna četrtina je prazna, ostale tri pa so si enake. In tudi pri teh četrtinah četrtin, torej podtabelah velikosti 2×2, lahko opazimo, da je zgornja desna četrtina prazna, v ostalih treh pa so enice. V naši tabeli se tako pravzaprav skriva zgornji del enega od najbolj znanih fraktalov — trikotnika Sierpińskega.

pogojev. Zapišimo torej pogoj za kompatibilnost:

$$\begin{aligned} & \neg\big((v_0 \wedge \bar{u}_0) \vee (v_1 \wedge \bar{u}_1) \vee (v_2 \wedge \bar{u}_2)\big) \\ = \quad & \neg(v_0 \wedge \bar{u}_0) \wedge \neg(v_1 \wedge \bar{u}_1) \wedge \neg(v_2 \wedge \bar{u}_2) \\ = \quad & (\bar{v}_0 \vee u_0) \wedge (\bar{v}_1 \vee u_1) \wedge (\bar{v}_2 \vee u_2). \end{aligned}$$

Na vsakem od naših treh bitov mora torej veljati, da ima na tistem mestu bodisi u enico ali pa v ničlo (ali pa celo oboje). Če torej v v obrnemo vse tri bite (iz enic naredimo ničle in obratno; to lahko naredimo tako, da izračunamo v xor 7) in ga z binarnim operatorjem *ali* združimo z u, morajo biti v rezultatu vsi trije biti prižgani (torej mora imeti rezultat vrednost 7). Tako se torej lahko v gornji rešitvi odpovemo tabeli mozna in namesto pogoja

```
mozna[tipPrejemnika][tipDarovalca]
```

zapišemo:[17]

```
(tipPrejemnika | (tipDarovalca ^ 7)) == 7
```

Še en zelo zanimiv, čeprav ne tako praktičen način za preverjanje kompatibilnosti pa je naslednji. Videli smo, da naša tabela tvori prvih osem vrstic trikotnika Sierpińskega; in znano je (o tem se lahko prepričamo z indukcijo), da lahko ta trikotnik med drugim dobimo tako, da vzamemo Pascalov trikotnik (ki ga tvorijo binomski koeficienti) in od vsakega števila v njem obdržimo le ostanek po deljenju z 2. Tako torej vidimo, da lahko pacient tipa u dobi transfuzijo krvi tipa v natanko tedaj, ko je $\binom{u}{v} \bmod 2 = 1$.[18]

(*b*) Ker imamo pri tej podnalogi opravka le z negativnimi tipi krvi, znaka − v tem delu rešitve ne bomo pisali.

Za začetek lahko vso kri tipa 0 razdelimo pacientom tipa 0, dokler ne zmanjka bodisi krvi bodisi pacientov. Kako vemo, da nas to ne bo pripeljalo do rešitve, ki bi bila slabša od najboljše možne? Pa recimo, da bi se dalo dobiti boljšo rešitev tako, da razdelimo pacientom tipa 0 nekaj manj krvi tipa 0, recimo d enot manj. Teh d enot krvi tipa 0 lahko porabimo za kakšne druge paciente, ki v naši prvotni rešitvi mogoče niso dobili krvi; tako se lahko rešitev za največ d izboljša. Toda po drugi strani imamo tudi d pacientov tipa 0, ki po novem ne dobijo krvi tipa 0, v prvotni rešitvi pa so jo; in ker ti pacienti sploh ne morejo sprejeti krvi nobenega drugega tipa, bodo po novem ostali brez transfuzije; tako se rešitev za d poslabša. Tako smo torej izgubili d pacientov in jih pridobili kvečjemu d, tako da po novem gotovo nismo nič na boljšem kot prej.

S podobnim razmislekom lahko v nadaljevanju utemeljimo, da je smiselno vso kri tipov 0 in A razdeliti pacientom tipa A, dokler bodisi ne zmanjka krvi bodisi takih pacientov. Nato je smiselno vso kri tipov 0 in B razdeliti pacientom tipa B, dokler spet ne zmanjka bodisi krvi bodisi pacientov. Nazadnje lahko vso preostalo kri razdelimo pacientom tipa AB.

[17] Pozor, zunanji par oklepajev je pomemben, saj v C/C++ operator == veže močneje kot | in ^. Po drugi strani pa bi se notranjemu paru oklepajev lahko tudi odpovedali, saj ^ veže močneje kot |.

[18] Za več o povezavi med Pascalovim trikotnikom in trikotnikom Sierpińskega glej tudi rešitev naloge 2011.X.32 na str. 146.

Če uporabimo enako številčenje tipov krvi od 0 do 7 kot pri (a), je naša rešitev zdaj pravzaprav zelo preprosta:

```
void Razdeli(int zaloga[4], int potrebe[4])
{
  int u, v, prenos, skupaj = 0;
  for (u = 0; u < 4; u++) for (v = 0; v < 4; v++) {
    if (! mozna[u][v]) continue;
    prenos = (zaloga[v] < potrebe[u]) ? zaloga[v] : potrebe[u];
    if (prenos <= 0) continue;
    printf("%d pacientov tipa %s dobi kri tipa %s\n", prenos, imena[u], imena[v]);
    zaloga[v] -= prenos; potrebe[u] += prenos; skupaj += prenos; }
  printf("Skupaj %d transfuzij.\n", skupaj);
}
```

Ali bi lahko to rešitev uporabili za vseh osem tipov krvi, torej če negativnim tipom dodamo še pozitivne? Na prvi pogled se mogoče zdi, da bi to šlo; ker lahko pacienti z negativnimi krvnimi skupinami dobijo le kri negativnih skupin, pacienti s pozitivnimi skupinami pa kri tako pozitivnih kot negativnih skupin, bi lahko poskusili kri najprej razdeliti pacientom z negativnimi skupinami po enakem postopku kot zgoraj, kasneje pa po podobnem postopku še pozitivnim. Ker imajo v našem številčenju negativni tipi številke od 0 do 3, pozitivni pa od 4 do 7, bi se spremenil le začetek zgornjega podprograma:

```
void Razdeli(int zaloga[8], int potrebe[8])
{
  int u, v, prenos, skupaj = 0;
  for (u = 0; u < 8; u++) for (v = 0; v < 8; v++) {
```

Izkaže pa se, da razdelitve, ki jih najde ta postopek, niso vedno najboljše možne. Primer: recimo, da imamo na zalogi po eno transfuzijo tipa 0−, A− in B−; in da imamo po enega pacienta tipa AB−, 0+ in B+. Zgornji postopek bi (pri $u = 3$, $v = 0$) dal pacientu AB− kri tipa 0−, nato bi pri $u = 4$ ugotovil, da za pacienta 0+ nima primerne krvi, in nazadnje bi (pri $u = 6$, $v = 1$) dal pacientu B+ kri tipa A−. Boljšo rešitev dobimo, če damo pacientu AB− kri tipa A−, pacientu 0+ kri tipa 0− in pacientu B+ kri tipa B−.

Naš postopek bi lahko poskusili popraviti tako, da bi pare (u, v) (pri čemer je u tip krvi prejemnika, v pa tip krvi darovalca) pregledal v kakšnem drugačnem vrstem redu. Doslej smo jih pregledovali naraščajoče po u in pri vsakem u še naraščajoče po v; lahko bi zanko po v predelali tako, da bi šla po padajočih v, saj bi tako prej porabila bolj specifične tipe krvi (tiste, ki so primerne za manj različnih vrst prejemnikov). Vendar tudi po tej predelavi na primeru iz prejšnjega odstavka ne dobimo optimalne rešitve. Tudi v splošnem se izkaže, da za vsak vrstni red parov (u, v) obstaja kakšen tak vhodni primer (torej tabeli zaloga in potrebe), za katerega bi pri tem vrstnem redu dobili suboptimalno rešitev.

(c) V tej različici naloge se skriva problem maksimalnega pretoka v grafu. Pri tem problemu imamo usmerjen graf, v katerem ima vsaka povezava $u \to v$ tudi znano *kapaciteto*, $c(u, v)$. V grafu si izberimo dve točki, ki jima bomo rekli *izvor* s in *ponor* t. Potem je *pretok* od s do t vsaka funkcija f, ki priredi vsaki povezavi $u \to v$ neko

količino toka $f(u,v)$ in pri tem upošteva naslednji dve omejitvi: (1) tok ne preseže kapacitete povezav: $0 \leq f(u,v) \leq c(u,v)$; (2) v vsaki točki razen izvora s in ponora t se pretok niti ne izgublja niti ne nastaja, zato je vsota tokov na vhodnih povezavah enaka vsoti tokov na izhodnih povezavah. Maksimalni pretok je tisti, pri katerem je skupna količina toka iz točke s največja možna.

Za našo nalogo s transfuzijami bomo sestavili graf takole. Označimo zalogo krvi tipa i z z_i; število pacientov, ki potrebujejo kri tipa i, pa naj bo p_i. Naš graf bo imel $n+2$ točki: poleg izvora s in ponora t imejmo še po eno točko za vsak tip krvi (točki, ki predstavlja tip i, recimo x_i). Za vsak tip krvi i imejmo povezavo $s \to x_i$ s kapaciteto z_i in povezavo $x_i \to t$ s kapaciteto p_i. Poleg tega še za vsak par (i,j), če lahko pacient s krvjo tipa i prejme kri tipa j, v naš graf dodajmo povezavo $x_j \to x_i$ z neomejeno kapaciteto.

Vidimo lahko, da nam vsak pretok f od s do t v tem grafu zdaj pravzaprav predstavlja načrt razdelitve krvi med paciente; če lahko i prejme kri tipa j, nam tok $f(x_j, x_i)$ pove, koliko pacientov tipa i naj dobi kri tipa j; tok $f(s, x_i)$ nam pove skupno število pacientov, ki prejmejo kri tipa i; in tok $f(x_i, t)$ nam pove skupno število pacientov tipa i, ki so dobili transfuzijo. Če torej poiščemo največji možni pretok od s do t, bomo s tem dobili tudi največje možno število transfuzij.

Preprost postopek za iskanje največjega pretoka je na primer Ford-Fulkersonov algoritem. Na začetku postavimo tok po vseh povezavah na 0; nato pa na vsakem koraku poiščemo poljubno táko pot od s do t, po kateri bi se dalo pretok še povečati (ne da bi pri tem presegli kapaciteto kakšne povezave na poti); pretok po povezavah na tej poti zdaj povečamo (na vseh enako) in ta postopek ponavljamo, dokler se še da najti kakšno primerno pot. Pomembno pri tem pa je, da sme naša pot uporabljati povezave tudi v obratni smeri; v takem primeru dodajanje toka na pot pomeni, da pretok po tisti povezavi za toliko zmanjšamo (pri čemer seveda ne sme pasti pod 0).

13. KvadMars

(*a*) Ko se premikamo naprej po labirintu, moramo po vsakem premiku ugotoviti, v katero smer se pot nadaljuje: lahko naprej v isti smeri kot doslej, lahko pa v smeri 90 stopinj levo ali desno. Še ena možnost pa je, da so v vseh teh smereh neprehodna polja; takrat vemo, da smo prišli na cilj in bomo morali začeti razmišljati o tem, kako se vrniti na začetno polje. Zapišimo za začetek dosedanji postopek (brez vračanja na začetni položaj):

```
while (true) {
  if (Tipalo() != 0) { Naprej(1); continue; }
  ObrniLevo();
  if (Tipalo() != 0) { Naprej(1); continue; }
  ObrniDesno(); ObrniDesno();
  if (Tipalo() != 0) { Naprej(1); continue; }
  break; }
```

Potencialna izboljšava je, da se premikamo po več polj naenkrat, če nam tipalo pravi, da je ovira še daleč:

```
int t;
⋮
if ((t = Tipalo()) != 0) { Naprej(t < 0 ? 10 : t); continue; }
```

Spomnimo se, da naloga pravi, da če je ovira več kot 10 polj daleč, nam Tipalo vrne -1, tako da se takrat ne smemo premakniti za več kot 10 polj, saj ne vemo, kako daleč je ovira v resnici.

Kako pa naj se robot vrne na začetni položaj? Ena možnost je, da vpeljemo sklad, na katerega odlagamo vse premike, ki smo jih robotu doslej ukazali. Ko se moramo vrniti v začetni položaj, beremo premike s sklada in izvajamo z robotom ravno nasprotne premike: kjer smo prej izvedli premik desno, izvedemo zdaj premik levo in obratno; kjer smo prej izvedli korak naprej, izvedemo zdaj (enako dolg) korak nazaj. Sklad bi lahko v praksi implementirali s seznamom (*linked list*) ali pa s tabelo (ki pa bi jo morali dinamično povečevati, saj vnaprej ne moremo vedeti, koliko premikov bomo izvedli).

Slabost rešitve s skladom je, da lahko potencialno porabi veliko pomnilnika. Ker podnaloga (d) želi, da se temu izognemo, lahko razmišljamo takole: ker na poti ni nobenih razvejitev ali česa podobnega, je dovolj že, če si zapomnimo, koliko korakov naprej smo naredili. Prvotni postopek dopolnimo takole:

```
int d = 0;
⋮
if (Tipalo() != 0) { Naprej(1); d++; continue; }
```

Ob koncu postopka vemo, da je robot na ciljnem polju in obrnjen za 90 stopinj desno glede na smer, v katero je bil obrnjen, ko je prišel v to polje. Če ga torej obrnemo še za 90 stopinj v desno, bomo pripravljeni za pot nazaj.

```
ObrniDesno();
while (d > 0) {
  if (Tipalo() != 0) { Naprej(1); d--; continue; }
  ObrniLevo();
  if (Tipalo() != 0) { Naprej(1); d--; continue; }
  ObrniDesno(); ObrniDesno();
  if (Tipalo() != 0) { Naprej(1); d--; continue; }
  break; }
```

Ta postopek na vsakem koraku najprej preizkusi smer naravnost, nato levo (glede na dosedanjo smer) in nato desno. Če ta vrstni red spremenimo v levo, naravnost, desno, lahko postane postopek še malo elegantnejši (in niti ne potrebuje obrata v desno pred zanko):

```
while (d > 0) {
  ObrniLevo();
  while (Tipalo() == 0) ObrniDesno();
  Naprej(1); d--; }
```

Načeloma bi lahko podobno naredili že pri prvi zanki while, torej tisti, s katero se robot premika od začetnega položaja do cilja. (Tam bi se morala zunanja zanka ustaviti, če bi kdaj v notranji zanki naredila tri zaporedne obrate desno.) Težava nastopi le na samem začetku: če je robot v začetnem položaju obrnjen tako, da se labirint nadaljuje tako v smeri naravnost kot v smeri levo, bi ga ta različica rešitve obrnila levo in ga s tem pripravila do tega, da bi raziskoval labirint v drugi smeri kot pa v tisti, kamor je bil prvotno obrnjen.

(*b*) Pri raziskovanju labirinta si lahko pomagamo z zagotovilom, da prosta polja ne tvorijo niti ciklov niti velikih prostih dvoran (velikosti 2×2 ali še večjih). Obrnimo se tako, da bo zid na naši desni, in se odtlej premikajmo ob zidu tako, da bo ta ves čas na naši desni. Po vsakem koraku naprej preverimo, če je pred nami zid, in če je, poskusimo najprej z obratom v desno in šele nato z obratom v levo (ali celo za 180 stopinj). Tako bomo sčasoma obiskali vsa dosegljiva prosta polja in prišli nazaj v začetno polje. To, da je naš obhod končan, prepoznamo po tem, da smo se ne le znašli v istem polju kot na začetku, pač pa smo tudi obrnjeni v isto smer. (Večino polj namreč pri tem obhodu obiščemo dvakrat, ker pač polje meji na dva zidova in imamo enkrat na svoji desni enega od teh dveh zidov, enkrat pa drugega.)

Da bomo lahko ugotovili, kdaj je robot prišel nazaj na začetni položaj, moramo med premikanjem in obračanjem robota v nekaj spremenljivkah vzdrževati njegove koordinate in smer. Pravih koordinat in smeri sicer ne moremo poznati, saj ne poznamo začetnih koordinat in smeri, vendar to za naš namen niti ni pomembno; dovolj je, če hranimo robotove koordinate in smer relativno glede na začetne koordinate in smer. Spodnja rešitev ima zato globalne spremenljivke xr, yr in dr ter podprograme Naprej2, Nazaj2, ObrniLevo2 in ObrniDesno2, ki popravijo te spremenljivke in nato pokličejo prave funkcije za nadzor robota.

Paziti moramo še na možnost, da je začetni položaj robota izbran tako, da ob njem ni nobenega zidu (ker robot stoji na križišču štirih poti). V tem primeru se lahko premaknemo v poljubno smer in smo lahko prepričani, da novi položaj meji na zid na vsaj eni strani (v nasprotnem primeru bi imeli kvadrat 2×2 prostih polj). Iz tega novega položaja torej zdaj poženemo dosedanji postopek, nato pa se le še vrnemo nazaj na začetni položaj.

```
const int DX[4] = { 1, 0, −1, 0 }, DY[4] = { 0, 1, 0, −1 };
int xr, yr, dr;

void Naprej2(int d) { xr += d * DX[dr]; yr += d * DY[dr]; Naprej(d); }
void Nazaj2(int d) { xr −= d * DX[dr]; yr −= d * DY[dr]; Nazaj(d); }
void ObrniLevo2() { dr = (dr + 1) % 4; ObrniLevo(); }
void ObrniDesno2() { dr = (dr + 3) % 4; ObrniDesno(); }

void RobotB()
{
  int x0, y0, d0, n;
  xr = 0; yr = 0; dr = 0;
  /* Poiščimo kakšen zid ob trenutnem položaju. */
  for (n = 0; n < 4 && Tipalo() != 0; n++) ObrniLevo2();
  /* Če ob trenutnem položaju ni zidu, naredimo korak naprej. */
  if (n == 4) { Naprej2(1);
    /* Ob novem položaju zid gotovo obstaja. */
    while (Tipalo() != 0) ObrniLevo2(); }
  /* Zdaj je pred nami zid. Obrnimo se levo, da bo zid na naši desni,
     zapomnimo si trenutni položaj in smer ter začnimo obhod ob zidu. */
  ObrniLevo2(); x0 = xr; y0 = yr; d0 = dr;
  do {
    /* Če na naši desni ni zidu, se obrnimo v desno in naredimo korak naprej; če pa na
       naši desni je zid, poskusimo iti naravnost; če niti to ne gre, poskusimo levo itd. */
    ObrniDesno2();
    for (n = 0; n < 4 && Tipalo() == 0; n++) ObrniLevo2();
```

```
    if (n == 4) break; /* Očitno je trenutno polje povsem zazidano. */
    Naprej2(1);
  } while (! (xr == x0 && yr == y0 && dr == d0));
  /* Vrnimo se na pravi začetni položaj, torej (0, 0). */
  if (! (x0 == 0 && y0 == 0)) {
    /* Najprej se moramo obrniti v primerno smer. */
    while (! (xr + DX[dr] == 0 && yr + DY[dr] == 0)) ObrniLevo2();
    Naprej2(1); }
}
```

Ta rešitev porabi le konstantno mnogo pomnilnika, tako da je primerna tudi za podnalogo (d).

(c) Če ima robot dovolj pomnilnika, si lahko v neki tabeli ali kakšni podobni podatkovni strukturi gradi sliko tistih delov labirinta, ki jih je doslej že raziskal. Naloga postane tako precej podobna problemu barvanja s poplavljanjem (*flood fill*). Lahko si jo tudi predstavljamo kot problem pregledovanja grafa, pri čemer graf tvorijo vsa dosegljiva polja našega labirinta (povezave pa so prisotne tam, kjer imata dve polji skupno stranico).

Labirint lahko na primer pregledujemo z rekurzijo: ko se nahajamo v nekem polju, pregledamo vsa štiri sosednja polja in če so prosta, zanje izvedemo rekurzivne klice. Pri tem si moramo nekako označiti, katera polja smo že obiskali, da se nam ne bo robot zaciklal (preden izvedemo nov rekurzivni klic, najprej preverimo, če nismo tistega polja že obiskali, in če smo ga, klica ne izvedemo). Če bi vnaprej vedeli, kako velika je mreža in kje v njej se nahajamo, bi lahko to počeli z običajno dvodimenzionalno tabelo; ker tega ne vemo, bi morali tabelo občasno dinamično povečevati in premikati podatke po njej. Zato si raje pomagajmo z razpršeno tabelo (*hash table*); ker ta v C-ju ni del standardne knjižnice, bomo spodnjo rešitev raje napisali v C++11, kjer lahko uporabimo razred unordered_set. Par koordinat bomo predstavili s strukturo tipa pair<**int**, **int**>; ker zanje C++ova standardna knjižnica ne zna računati razprševalnih kod, bomo morali napisati tudi svojo razprševalno funkcijo, za kar v spodnji rešitvi skrbi razred MyPairHash.[19]

```
#include <utility> // za pair<...>
#include <unordered_set>

using namespace std;

struct MyPairHash {
  inline size_t operator ()(const pair<int, int>& p) const {
    size_t a = (size_t) ((p.first <= 0) ? -2 * p.first : 2 * p.first - 1);
    size_t b = (size_t) ((p.second <= 0) ? -2 * p.second : 2 * p.second - 1);
    return (a + b) * (a * b * 1) / 2 + a; }
};

unordered_set<pair<int, int>, MyPairHash> obiskano;

void Rekurzija()
{
```

[19] Naša razprševalna funkcija deluje takole: najprej vsako koordinato pretvorimo v nenegativno celo število in to tako, da iz pozitivnih koordinat nastanejo liha števila ($t \mapsto 2t - 1$), iz ostalih pa soda ($t \mapsto -2t$). Nato par nenegativnih števil (x, y) preslikamo v $(x + y)(x + y + 1)/2 + x$. Z nekaj razmisleka se lahko prepričamo, da smo tako dobili bijektivno funkcijo iz $\mathbb{Z} \times \mathbb{Z}$ v $\mathbb{Z}_0^+$.

```
  int n;
  // Označimo polje kot obiskano.
  obiskano.insert(pair<int, int>(xr, yr));
  // Preglejmo sosednja štiri polja.
  for (n = 0; n < 4; n++) {
    // Ali smo to sosednje polje že obiskali?
    if (obiskano.find(pair<int, int>(xr + DX[dr], yr + DY[dr])) == obiskano.end())
      // Če ga nismo in če je prosto, ga obiščimo zdaj (z rekurzivnim klicem).
      if (Tipalo() != 0) { Naprej2(1); Rekurzija(); Nazaj2(1); }
    ObrniLevo2(); }
}

void RobotC()
{
  xr = 0; yr = 0; dr = 0;
  Rekurzija();
}
```

Slabost te rešitve je, da je poraba pomnilnika sorazmerna s številom prostih polj, ki jih robot lahko doseže. Ker naloga (d) sprašuje, kako lahko problem rešimo z manj pomnilnika, razmislimo še o kakšni varčnejši rešitvi.

Pri nalogi (b) smo videli, kako lahko robot sledi zidu (meji med prostimi in neprehodnimi polji) tako, da je ta zid ves čas na njegovi desni in da se robot sčasoma vrne v polje, na katerem je obhod okoli zidu začel. To lahko počnemo tudi zdaj pri (c), je pa koristno ločiti med dvema vrstama zidov: zunanjimi (torej zidovi na zunanjem robu dosegljivega območja) in notranjimi (do teh pridemo, če znotraj dosegljivega območja ležijo kakšni „otoki“ iz nedosegljivih polj). Zunanji zid lahko od notranjega ločimo med drugim tako, da preštejemo, koliko je bilo (pri obhodu, kjer je bil zid na naši desni) v vogalih levih ovinkov (recimo L) in koliko desnih (recimo D); pri zunanjem zidu velja $L - D = 4$, pri notranjem pa $D - L = 4$.

Recimo, da smo se z robotom sprehodili ob nekem zidu in si med prostimi polji (ob zidu), ki smo jih pri tem obiskali, zapomnili najvišje polje, torej tisto z največjo y-koordinato. Če je bil zid zunanji, je severni sosed tega polja gotovo neprehodno polje, če pa je bil zid notranji, je severni sosed tega polja gotovo prosto polje.

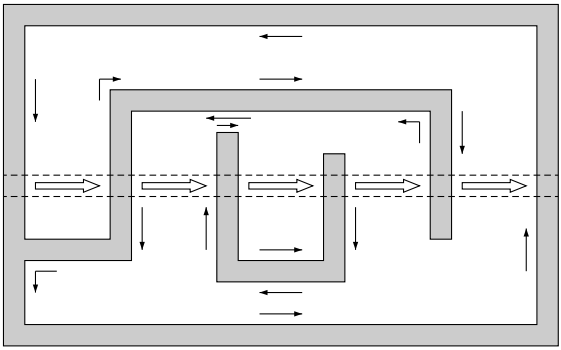
S tem razmislekom ni težko poiskati na primer najvišjega dosegljivega polja nasploh. Robota postavimo v smer navzgor (v smeri naraščajočih y-koordinat) in se premikajmo naprej, dokler nam tipalo ne pove, da je tik pred nami zid. Ta zid obhodimo, pri tem si zapomnimo polje z najvišjo y-koordinato (recimo mu P) in ob koncu obhoda nadaljujmo ob zidu, dokler ne pridemo spet v polje P. Če je P-jev zgornji sosed neprehoden, smo ob zunanjem zidu in je P najvišje dosegljivo polje sploh; če pa je P-jev zgornji sosed prost, smo bili ob notranjem zidu in lahko robot nadaljuje s premikanjem navzgor, dokler ne naleti na nek nov zid; ta postopek ponavljamo, dokler ne pridemo do zunanjega zidu.

Zdaj smo ob zunanjem zidu in poznamo y-koordinato najvišjega dosegljivega polja, recimo y_M. Izvedimo naslednji postopek:

```
1   t := y_M;
2   while true:
3     naredi obhod ob zunanjem zidu in si med polji z y-koordinato t zapomni
```

tisto, ki ima najmanjšo x-koordinato; recimo mu (x, t);
4 **if** takega polja sploh ni **then break**;
5 nadaljuj z obhodom po zunanjem zidu, dokler ne prideš spet v (x, t);
6 obišči od leve proti desni vsa dosegljiva prosta polja, ki imajo $y = t$
(ob koncu tega postopka je robot spet ob zunanjem zidu);
7 $t := t - 1$;

V vsaki iteraciji zunanje zanke torej obiščemo vsa prosta polja v neki vrstici naše mreže, ustavimo pa se, ko pridemo do dna mreže (do najmanjše y-koordinate, ki je še dosegljiva ob zunanjem zidu).



Primer labirinta z enim zunanjim in enim notranjim zidom. Tanke puščice kažejo smer gibanja robota pri obhodu zidov, če je zid ves čas na robotovi desni strani. Debele puščice kažejo premike, s katerimi bi robot obiskal vsa dosegljiva polja v eni vrstici.

Razmislimo še o tem, kako v koraku 6 obiskati vsa dosegljiva polja v vrstici $y = t$; postopek je zelo podoben tistemu, s katerim smo prej iskali najvišje polje. Na začetku se robot nahaja v najbolj levem dosegljivem polju te vrstice. Obrnimo se proti desni in se premikajmo v to smer, dokler ne pridemo do zidu. Recimo, da smo zdaj v polju (x_1, t). Naredimo obhod ob tem zidu (pri čemer ga imamo ves čas na naši desni) in si pri tem zapomnimo najbolj levo med vsemi polji, ki imajo $y = t$ in $x > x_1$; recimo temu polju (x_2, t). Nadaljujmo obhod ob zidu, dokler ne pridemo spet v (x_2, t); če je desni sosed tega polja neprehoden, smo končali (in tisti zid je zunanji), sicer pa se lahko spet premikamo v desno, dokler ne pridemo spet do zidu in tako naprej. Postopek se ustavi, ko opazimo, da ob trenutnem zidu na $y = t$ ni nobenega polja, ki bi bilo še bolj desno od našega trenutnega. (Primer takšnega prehoda vrstice je na gornji sliki prikazan z debelimi puščicami.)

(e) Premik naprej lahko izvedemo s pomočjo ostalih ukazov tako, da se obrnemo za 180 stopinj, izvedemo premik nazaj in se nato obrnemo nazaj v prvotno smer. Vse klice funkcije Naprej lahko torej zamenjamo s klici takšne funkcije:

```
void ZasilnoNaprej(int d)
{
  ObrniLevo(); ObrniLevo();
  Nazaj(d);
  ObrniLevo(); ObrniLevo();
}
```

14. Torta

Izberimo si dva reza, recimo i in j; kako bi preverili, ali se sekata? Če je $u_i < u_j$ in $v_i < v_j$, potem leži rez i v celoti levo od reza j, torej se ne sekata. Podobno je, če

je $u_i > u_j$ in $v_i > v_j$ (tedaj leži rez i v celoti desno od reza j). Če je $u_i = u_j$ ali pa $v_i = v_j$, se reza stikata na začetku ali na koncu, torej se tudi ne moreta sekati. Do sekanja pride torej le v primeru, če je $u_i < u_j$ in $v_i > v_j$ ali pa obratno (torej $u_i > u_j$ in $v_i < v_j$). Takrat pa je sekanje neizogibno: ker se en rez začne levo od drugega, konča pa desno od njega, se nekje vmes zagotovo sekata.

S tem razmislekom lahko že zapišemo preprosto rešitev naše naloge: pregledamo lahko moramo vse pare rezov in za vsakega preverimo, ali se reza sekata ali ne.

```
s := 0;
for i := 2 to n do for j := 1 to i − 1 do
  if (u_i < u_j and v_i > v_j) or (u_i > u_j and v_i < v_j)
  then s := s + 1;
```

Na koncu tega postopka imamo v s število presečišč. Pogoj, s katerim preverjamo, ali se reza sekata, bi lahko elegantno zapisali tudi kot $(u_i - u_j)(v_i - v_j) < 0$, saj nam predznak razlike $u_i - u_j$ pove, ali se i začne levo ali desno od j, podobno pa nam predznak $v_i - v_j$ pove, ali se i konča levo ali desno od j; če je njun zmnožek negativen, se predznaka razlikujeta, kar pomeni, da se i in j sekata.

Ta postopek ima časovno zahtevnost $O(n^2)$. Razmislimo o tem, kako ga lahko še izboljšamo. Reze lahko uredimo naraščajoče po začetni koordinati, u_i, in jih oštevilčimo v tem vrstnem redu. V nadaljevanju torej predpostavimo, da je $u_1 \leq u_2 \leq \ldots \leq u_n$. Gornji postopek bi zdaj pri vsakem i šel po vseh tistih rezih, ki se začnejo levo od našega (ker je $j < i$, bi bil $u_j < u_i$), in med njimi bi pravzaprav štel tiste, ki se končajo desno od našega (torej imajo $v_j > v_i$). Predstavljamo si lahko, da imamo končne koordinate vseh teh rezov v nekem seznamu M:

```
s := 0; M := prazen seznam;
for i := 2 to n:
  s := s + (število elementov M-ja, ki so večji od v_i);
  dodaj v_i v M;
```

Vprašanje je, kako implementirati M, da bomo lahko učinkovito prešteli elemente, ki so večji od v_i. Če bi bil M neurejen seznam, bi za preštevanje teh elementov potrebovali zanko po vseh elementih seznama, tako da ne bi bili nič na boljšem kot pri prvotni rešitvi — pri vsakem i bi imeli še vedno $O(n)$ dela. Če bi bil M urejen seznam, bi lahko z bisekcijo poiskali prvi element, večji od v_i, kar bi nam vzelo le $O(\log n)$ časa, vendar bi za dodajanje novega elementa v M porabili $O(n)$ časa, tako da spet nismo ničesar pridobili.

Boljša rešitev je, da M organiziramo v drevo, na primer rdeče-črno drevo ali AVL-drevo; tedaj lahko obe operaciji, dodajanje novega elementa in štetje elementov, ki so večji od v_i, izvedemo v $O(\log n)$ časa. Še ena možnost je, da koordinate v_i pred izvajanjem našega postopka predelamo v cela števila od 1 do n tako, da se njihov medsebojni vrstni red nič ne spremeni;[20] potem lahko za M uporabimo Fenwickovo drevo. Pri vseh teh vrstah dreves bo časovna zahtevnost celotnega postopka $O(n \log n)$, vključno z urejanjem rezov po u_i na začetku.

[20] Torej tisti v_i, ki je med vsemi najmanjši, dobi vrednost 1; tisti, ki je bil drugi najmanjši, dobi vrednost 2 in tako naprej. To lahko naredimo v $O(n \log n)$ časa z urejanjem parov (v_i, i).

15. Trgovina

Spodnja rešitev uporablja tri tabele: za tip izdelka t imamo v stanje[t] trenutno stanje zaloge tega tipa izdelkov, v minimalno[t] in polnilno[t] pa minimalno in polnilno število kosov izdelkov tega tipa. Ker naloga ne določa podrobno, od kod naj te podatke preberemo in v kakšni obliki so zapisani, smo predpostavili kar, da jih dobimo na standardnem vhodu: v prvi vrstici je število tipov izdelkov, nato pa je za vsak tip po ena vrstica, ki vsebuje začetno stanje zaloge ter minimalno in polnilno število za ta tip izdelka. Nato prebiramo podatke o nakupih. Po vsakem nakupu zmanjšamo stanje zaloge tistega tipa izdelkov za število kupljenih izdelkov; nato še preverimo, če je zaloga zdaj pod minimalnim številom, in če je, jo povečamo za polnilno število.

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

int main()
{
  int nTipov, tip, nakup, *stanje, *minimalno, *polnilno;

  /* Preberimo podatke o številu tipov izdelkov, začetnem stanju zaloge,
     minimalnem in polnilnem številu kosov. */
  scanf("%d", &nTipov);
  stanje = (int *) malloc(sizeof(int) * nTipov);
  minimalno = (int *) malloc(sizeof(int) * nTipov);
  polnilno = (int *) malloc(sizeof(int) * nTipov);
  for (tip = 0; tip < nTipov; tip++)
    scanf("%d %d %d", &stanje[tip], &minimalno[tip], &polnilno[tip]);

  /* Prebirajmo podatke o nakupih, vse do konca vhoda. */
  while (2 == scanf("%d %d", &tip, &nakup))
  {
    /* Zaloga tega izdelka se zaradi nakupa zmanjša. */
    stanje[tip] -= nakup;

    /* Če zaloga ni več nad minimalno, se poveča za polnilno število. */
    if (stanje[tip] <= minimalno[tip])
      stanje[tip] += polnilno[tip];
  }

  /* Izpišimo končno stanje zaloge. */
  for (tip = 0; tip < nTipov; tip++)
    printf("%d\n", stanje[tip]);

  /* Pospravimo za sabo. */
  free(stanje); free(minimalno); free(polnilno); return 0;
}
```

Rešitev bi se dalo še izboljšati, da bi opozarjala na morebitne napake v vhodnih podatkih; na primer, če je nakup večji od trenutnega stanja zaloge tistega izdelka. Do neugodnih primerov lahko pride tudi, če je polnilno število manjše od minimalnega; tedaj se lahko zgodi, da zaloga ostane pod minimalno tudi po tistem, ko smo jo povečali za polnilno število. Za take primere bi bilo koristno spremeniti stavek **if** v zanko **while** (stanje[tip] <= minimalno[tip]), vendar to pravzaprav že ni več v skladu z besedilom naloge.

16. T9

Za vsako besedo našega slovarja določimo zaporedje pritiskov na tipke, s katerimi

bi to besedo natipkali. Na primer, recimo, da imamo besedo w, dolgo n znakov: $w = w_1 w_2 \ldots w_n$. Za vsak znak poglejmo, kateri tipki pripada; recimo, da znak w_i pripada tipki t_i. Naloga pravi, da hočemo vsako besedo natipkati z največ k pritiski na tipke; sem najbrž štejemo tudi pritisk na `#` na koncu besede. Pred tem lahko torej izvedemo le $k-1$ pritiskov na tipke; besedo w bi torej natipkali kot $t_1 t_2 \ldots t_m$#, pri čemer je $m = \min\{n, k-1\}$. Zaporedje $t_1 t_2 \ldots t_m$ označimo kot $t(w)$.

Pri tej predelavi besede v (morebiti krajše) zaporedje tipk se seveda lahko zgodi, da iz različnih besed nastane isto zaporedje tipk. Naloga pravi, da moramo slovar oklestiti tako, da se bo na koncu dalo vsako besedo zapisati enolično, torej moramo od take skupine besed zavreči vse razen ene. Obdržimo seveda tisto, ki je najpogostejša, saj naloga pravi, naj maksimiziramo vsoto ocen pogostosti.

Vprašanje je še, kako čim ceneje odkriti primere, ko se več različnih besed preslika v isto zaporedje tipk. Besede, ki jih bomo obdržali v slovarju, lahko zlagamo v razpršeno tabelo (*hash table*) ali pa v drevo. Pri tem kot ključ uporabimo zaporedje tipk t, kot spremljevalne podatke pa besedo w in njeno pogostost uporabe. Če ob neki novi besedi ugotovimo, da se njena $t(w)$ že pojavlja kot ključ v naši razpršeni tabeli ali drevesu, moramo le še pogledati, ali je nova beseda pogostejša od tiste, ki je bila doslej zapisana ob tem ključu; če je nova beseda pogostejša, si namesto dosedanje zapomnimo njo. Na koncu vsebuje naša tabela (ali drevo) ravno tiste besede, ki jih moramo v vhodnem slovarju obdržati.

```
S := prazna razpršena tabela ali drevo;
za vsako besedo w iz vhodnega slovarja (s pogostostjo f_w):
  izračunaj pripadajoče zaporedje tipk t := t(w);
  če se t še ne pojavlja kot ključ v S:
    dodaj v S ključ t s spremljevalnimi podatki (w, f_w);
  sicer:
    naj bo w' beseda, ki se v S pojavlja kot spremljevalni
      podatek pri ključu t, in naj bo f_w' njena pogostost;
    če je f_w > f_w':
      popravi v S spremljevalne podatke pri ključu t na (w, f_w);
za vsak ključ t v S:
  izpiši besedo w, ki je v S kot spremljevalni podatek pri ključu t;
```

Količina dela, ki ga imamo s posamezno besedo w, je linearno sorazmerna z dolžino besede: to velja tako za računanje $t(w)$ kot za poizvedbo v S (in morebitno dodajanje v S ali nekoč kasneje brisanje iz S). Časovna zahtevnost celotnega postopka je torej v najslabšem primeru $O(nb)$.

17. Spirala

Ker obravnavamo pri tej nalogi vse znake kot enako široke, si lahko predstavljamo, da pišemo spiralo na karirasto mrežo, pri čemer v vsako celico mreže zapišemo po en znak. V mrežo vpeljimo še koordinatni sistem, pri čemer postavimo koordinatno izhodišče v tisto celico, v kateri začnemo izpisovati spiralo. Na spodnji sliki so celice oštevilčene v takem vrstnem redu, v kakršnem jih spirala doseže (0 je celica, v katero zapišemo prvi znak spirale):

3	25	26	27	28	29	30
2	24	9	10	11	12	31
1	23	8	1	2	13	32
0	22	7	0	3	14	33
−1	21	6	5	4	15	34
$y = -2$	20	19	18	17	16	35
	$x = -2$	−1	0	1	2	3

Opazimo lahko, da za vsak n velja naslednje: prvih n^2 celic, ki jih spirala doseže (torej tiste s številkami od 0 do $n^2 - 1$), tvori kvadrat oblike $n \times n$. Če je n lih, recimo $n = 2k + 1$, pokriva ta kvadrat v vsaki smeri (x in y) koordinate od $-k$ do $+k$; če pa je n sod, recimo $n = 2k$, pokriva ta kvadrat v vsaki smeri koordinate od $-k + 1$ do $+k$. Ti kvadrati so na zgornji sliki tudi narisani.

Če pogledamo polja od n^2 do $(n+1)^2 - 1$, vidimo, da se kvadrat $n \times n$ poveča v kvadrat $(n + 1) \times (n + 1)$ tako, da najprej pridobi spodnjo (če je n sod) ali zgornjo (če je n lih) stranico; to sestavlja n polj in z njo se nam kvadrat $n \times n$ poveča v pravokotnik $n \times (n + 1)$; v nadaljevanju pa ta pravokotnik pridobi na levi (če je n sod) oz. desni (če je n lih) še eno stranico z $n+1$ polji in se tako razraste v kvadrat $(n + 1) \times (n + 1)$.

S pomočjo teh opažanj ni težko ugotoviti, kateri del ravnine pokriva spirala za nek konkretni vhodni niz s. Recimo, da je ta niz dolg N znakov; poiščimo $n = \lfloor \sqrt{N} \rfloor$; tako je torej n tisto celo število, za katerega velja $n^2 \leq N < (n + 1)^2$. Zdaj vemo, da spirala v celoti pokrije kvadrat $n \times n$; ostanejo še morebitna polja s številkami od n^2 do $N - 1$. Če je $n^2 = N$, takih polj ni in spirala ima ravno obliko kvadrata $n \times n$. Če je $n^2 < N \leq n(n+1)$, imamo poleg kvadrata $n \times n$ še del vrstice tik nad ali pod njim; če pa je $n(n + 1) < N \leq (n + 1)^2$, imamo tisto vrstico v celoti, poleg tega pa še del vrstice levo ali desno od kvadrata $n \times n$.

Zdaj torej poznamo najmanjši pravokotnik, v katerem leži naša spirala (je pa res, da tega pravokotnika ne pokriva nujno v celoti — tak primer vidimo tudi v besedilu naloge, kjer je v zadnji vrstici pravokotnika samo en znak spirale, drugod pa so presledki). Izpišemo ga lahko tako, da se v zanki zapeljemo po vseh poljih pravokotnika, za vsako od njih ugotovimo, kateri znak v spirali se nahaja v njem, in ga izpišemo:

```
#include <stdio.h>

void Spirala(char *s)
{
  int N, n, k, xOd, xDo, yOd, yDo, x, y, i;

  /* Določimo dolžino niza, N, in njen koren, n. */
  N = 0; while (s[N]) N++;
  n = 0; while ((n + 1) * (n + 1) <= N) n++;

  /* Spirala torej pokriva vsaj kvadrat velikosti n * n.
     Na katerem razponu koordinat leži ta kvadrat? */
  k = n / 2; xOd = (n % 2 == 0) ? 1 - k : -k; xDo = k;
  yOd = xOd; yDo = xDo;

  /* Ali kakšen del spirale štrli čez rob tega kvadrata? */
```

```
  if (N > n * n) { if (n % 2 == 0) yOd--; else yDo++; }
  if (N > n * (n + 1)) { if (n % 2 == 0) xOd--; else xDo++; }
  /* Zdaj vemo, v katerem delu mreže leži spirala. */
  for (y = yDo; y >= yOd; y--) {
    for (x = xOd; x <= xDo; x++) {
      i = KoordVIndeks(x, y);
      putchar((i >= N) ? ' ' : s[i]); }
    putchar('\n'); }
}
```

Razmisliti moramo še o funkciji KoordVIndeks(x, y), ki za dano celico (x, y) izračuna, kateri znak spirale se nahaja na tej celici. Z drugimi besedami, ta funkcija mora prešteti, koliko celic obišče spirala, preden doseže celico (x, y).

Oglejmo si še enkrat gornjo sliko in primerjajmo kvadrata $n \times n$ in $(n+1) \times (n+1)$. Razlika med njima je lik v obliki črke L. Naj bo $a(x, y)$ tista izmed koordinat x in y, ki ima večjo absolutno vrednost; če pa imata obe enako absolutno vrednost, vzemimo za a manjšo od njiju (če sta torej koordinati različno predznačeni, vendar enaki po absolutni vrednosti, bomo za a vzeli tisto od njiju, ki je negativna). Opazimo lahko, da imajo vse celice na posameznem L-ju enako vrednost a. Če je $a < 0$, potem smo na tistem L-ju, ki ločuje kvadrat $2|a| \times 2|a|$ od kvadrata $(2|a| + 1) \times (2|a| + 1)$; drugače pa smo na tistem L-ju, ki ločuje kvadrat $(2|a| - 1) \times (2|a| - 1)$ od kvadrata $2|a| \times 2|a|$.

V vsakem primeru torej vemo, da spirala, preden doseže našo celico (x, y), v celoti obišče tisti manjši kvadrat (in zanj tudi vemo, koliko celic pokriva). Zdaj moramo prešteti le še to, koliko celic obišče spirala na trenutnem L-ju, preden doseže našo celico. Na primer, če je $a > 0$, spirala najprej obišče tiste celice L-ja, ki imajo $y = a$; obiskuje jih po naraščajočem x, od $x = -a + 1$ do $x = a$; torej pred celico (x, a) pride na vrsto $x - (-a+1)$ celic L-ja. Nato pa ta L obišče še celice, ki imajo $x = a$, in to po padajočem y, od $y = a - 1$ do $y = -a + 1$. Če ima torej naša celica koordinate oblike (a, y) (za $y < a$), pridejo na L-ju pred njo najprej vse celice z $y = a$ (takih celic je $2a$), nato pa mogoče še nekaj celic z $x = a$ (namreč tiste, ki imajo višji y od naše, takih pa je $(a - 1) - y$). S podobnim razmislekom lahko obdelamo tudi primer, ko je $a < 0$. Zapišimo to rešitev še v obliki podprograma:

```
#include <stdlib.h>

int KoordVIndeks(int x, int y)
{
  int a = (abs(x) == abs(y)) ? (x < y ? x : y) : (abs(x) > abs(y)) ? x : y;
  int n = 0;
  if (a > 0) {
    n = (2 * a - 1) * (2 * a - 1);
    if (y == a) n += x + a - 1;
    else n += a + a + (a - y) - 1; }
  else {
    n = (2 * a) * (2 * a);
    if (y == a) n += -a - x;
    else n += (-a) + (-a) + y - a; }
  return n;
}
```

Oglejmo si še malo drugačno rešitev naloge, ki je preprostejša, vendar porabi več pomnilnika. Naj bo spet N dolžina našega vhodnega niza s; vzemimo zdaj za n najmanjše liho število, ki je $\geq \sqrt{N}$. Recimo, da je $n = 2k + 1$; pripravimo si tabelo velikosti $n \times n$ elementov, ki nam bo predstavljala tisti del ravnine, na katerem so koordinate celic od $-k$ do $+k$. V naši tabeli nam celico s koordinatama (x, y) predstavlja element na indeksu $(y + k)n + (x + k)$.

Ker je $n^2 \geq N$, vemo, da tisti del spirale, ki ga pokriva naš niz (dolžine N znakov), v celoti leži znotraj kvadrata $n \times n$, ki ga predstavlja naša tabela. Tabelo lahko za začetek zapolnimo s presledki, nato pa se sprehodimo po spirali od začetka (v celici $(0, 0)$) naprej in sproti vpisujemo znake niza v primerne celice tabele. Na koncu moramo vsebino tabele le še izpisati; pri tem pazimo na to, da je tabela $n \times n$ mogoče malo prevelika za našo spiralo, zato si sproti zapomnimo največjo in najmanjšo x- in y-koordinato, ki smo jo pri zapisovanju spirale res dosegli, tako da bomo na koncu izpisali le tisti del tabele, na katerem se naša spirala res nahaja.

Slediti spirali po vrsti od celice $(0, 0)$ naprej je preprosto. Lahko si jo predstavljamo kot sestavljeno iz več skupin celic: najprej imamo dve skupini s po 1 celico, nato dve skupini dolžine 2, nato dve skupini dolžine 3, dve skupini dolžine 4 in tako naprej. Po vsaki skupini se moramo obrniti za 90 stopinj v smeri urinega kazalca.

3	25	26	27	28	29	30
2	24	9	10	11	12	31
1	23	8	1	2	13	32
0	22	7	0	3	14	33
−1	21	6	5	4	15	34
−2	20	19	18	17	16	35
$y = -3$						36
	$x = -2$	−1	0	1	2	3

Zapišimo to rešitev še s podprogramom:

```
void Spirala2(char *s)
{
  const int dx[4] = { 0, 1, 0, − 1};
  const int dy[4] = { 1, 0, −1, 0 };
  int N, n, k, K, xOd, xDo, yOd, yDo, x, y, i, smer, dolz, j;
  char *a;

  /* Določimo dolžino niza, N, in njen koren, n. */
  N = 0; while (s[N]) N++;
  n = 0; while (n * n < N) n++;

  /* Spirala torej pokriva vsaj kvadrat velikosti n * n. Za vsak slučaj
     ga zaokrožimo do lihe stranice, torej K * K za K = 2 * k + 1. */
  k = (n + 1) / 2; K = 2 * k + 1;
  a = (char *) malloc(K * K); memset(a, ' ', K * K);

  /* Sprehodimo se po spirali od začetne celice naprej in vpisujmo
     pripadajoče znake niza s v tabelo a. */
  x = 0; y = 0; xOd = x; yOd = y; xDo = x; yDo = y;
  i = 0; a[(y + k) * K + (x + k)] = s[i++];
```

```
  smer = 0; dolz = 1;
  while (i < N)
  {
    /* Nadaljujmo s skupino 'dolz' celic v smeri 'smer'. */
    for (j = 0; j < dolz && i < N; j++) {
      /* Premaknimo se v novo celico. */
      x += dx[smer]; y += dy[smer];
      /* Vpišimo trenutni znak niza v to celico. */
      a[(y + k) * K + (x + k)] = s[i++];
      /* Če je treba, popravimo meje območja, ki ga spirala pokriva. */
      if (x < xOd) xOd = x; else if (x > xDo) xDo = x;
      if (y < yOd) yOd = y; else if (y > yDo) yDo = y; }
    /* Po vsaki drugi skupini se dolžina poveča za 1. */
    if (smer == 1 || smer == 3) dolz++;
    /* Po vsaki skupini se obrnemo za 90 stopinj v desno. */
    smer = (smer + 1) % 4;
  }
  /* Zdaj vemo tudi, v katerem delu mreže leži spirala. Izpišimo ga. */
  for (y = yDo; y >= yOd; y--) {
    for (x = xOd; x <= xDo; x++) putchar(a[(y + k) * K + (x + k)]);
    putchar('\n'); }
  free(a);
}
```

18. Histogram

Naj bo naš histogram predstavljen z zaporedjem $a_1, \ldots, a_n$. Recimo, da se premikamo po histogramu z drsečim oknom širine w od leve proti desni. Najvišja možna višina pravokotnika pri trenutnem položaju okna je ravno minimum vseh a_i, ki so trenutno v oknu. Koristno si je torej zapomniti, kakšen je ta minimum, pa tudi to, pri katerem stolpcu nastopi; tako bomo zlahka videli, kdaj ob premiku okna ta minimum pade (na levi strani) iz okna (takrat moramo poiskati novi minimum). Ob vsakem premiku okna moramo seveda tudi preveriti, če ni stolpec, ki je ob tem premiku na novo prišel v okno (na desni strani) manjši od dosedanjega minimuma.

Slabost te rešitve je, da ko pade dosedanji minimum iz okna, se moramo sprehoditi po celem oknu, da najdemo novi minimum. Če so vhodni podatki dovolj neugodni, se lahko to zgodi pri vsakem premiku okna in časovna zahtevnost takega postopka bi bila $O(nw)$. Iskanje novega minimuma bi lahko pospešili tako, da bi stolpce, ki so trenutno v oknu, hranili v kopici, tako da bi bil najnižji med njimi vedno pri roki v korenu kopice; takšna rešitev bi imela časovno zahtevnost $O(n \log w)$.

Še lepša izboljšava pa je naslednja. Poleg najnižjega stolpca v oknu (recimo x_1) si zapomnimo še najnižjega od tistih stolpcev, ki so desno od njega. Ta (recimo mu x_2) je namreč tisti, ki postane najnižji v oknu sploh, ko dosedanji x_1 pade iz okna. Ob tem pa se znajdemo pred podobnim problemom kot prej: ko dosedanji x_1 pade iz okna in dosedanji x_2 postane novi x_1, moramo poiskati novega x_2. Da ne bo treba še enkrat po celem oknu, je torej koristno, če imamo pri roki še podatek o tem, kateri je najnižji stolpec desno od x_2 — recimo mu x_3. Podobno vidimo, da bomo potrebovali še x_4 in tako naprej. Imeli bomo torej celo zaporedje stolpcev, pri

čemer je x_1 najnižji stolpec v oknu, od tam naprej pa je x_{i+1} najnižji tak stolpec, ki je v oknu in leži desno od x_i.

Razmislimo še o tem, kako to zaporedje popraviti, ko v okno na desni strani vstopi nov stolpec. Če je novi stolpec (recimo mu u) nižji od x_1, postane u novi x_1; če ni nižji od x_1, je pa nižji od x_2, postane u novi x_2 in tako naprej. Ker je u trenutno najbolj desni stolpec okna, moramo v zaporedju x pobrisati vse člene za mestom, kamor smo vpisali u. Zelo prikladno je torej, če je zaporedje x predstavljeno z seznamom (pri čemer je x_1 na začetku seznama): dokler na koncu seznama še opažamo stolpce, ki so višji od u, jih brišemo, nato pa na konec seznama dodamo u. Brisanje x_1, ko le-ta pade na levi strani ven iz okna, pa je tudi enostavno, saj je x_1 na začetku seznama.

Zapišimo dobljeni postopek še s psevdokodo; v spremenljivki r hranimo najboljši rezultat doslej (po vseh doslej pregledanih položajih okna).

```
x := prazen seznam; r := −∞;
for i := 1 to n:
  u := a_i;
  if x ni prazen in je njegov začetni element enak i − w
    then pobriši začetni element seznama x;
  while x ni prazen:
    naj bo v zadnji element seznama x;
    if a_v > u then pobriši zadnji element seznama x else break;
  dodaj i na konec seznama x;
  if i ≥ w:
    naj bo v začetni element seznama x; r := max{v, r};
```

Kakšna je časovna zahtevnost tega postopka? Notranja zanka v vsaki iteraciji bodisi pobriše en element iz seznama bodisi izvede **break** in se konča. Slednje se lahko zgodi le enkrat pri vsakem i, vseh brisanj skupaj pa je lahko le toliko kot vseh dodajanj skupaj, to pa je n, saj vsak stolpec dodamo v seznam natanko enkrat. Zato izvede notranja zanka vsega skupaj le $O(n)$ iteracij in tudi časovna zahtevnost celotnega postopka je le $O(n)$.[21]

19. Pikavost

Ko ugotavljamo, ali je nek človek v danem družinskem drevesu pikast ali ne, je koristno, če takrat že vemo, kako je s pikavostjo njegovih staršev. Za to, da ugotovimo slednje, pa moramo pred tem poznati pikavost *njunih* staršev in tako nazaj po drevesu. Koristno je torej, če drevo pregledujemo sistematično od zgodnejših generacij proti kasnejšim. Postopek lahko zapišemo z zanko:

```
1  dokler nismo obdelali vseh ljudi v drevesu:
2    poišči takšno osebo u, za katero še nismo določili pikavosti,
       smo jo pa že določili za njena starša (ali pa starša nista znana);
3    določi pikavost osebe u;
```

[21] Mimogrede, ta naloga je pravzaprav poseben primer naloge s križanko, ki smo jo imeli na enem od prejšnjih tekmovanj (2010.3.4, str. 24 v biltenu 2010). Tudi naš pravkar opisani algoritem s časovno zahtevnostjo $O(n)$ je v tesnem sorodstvu z Vandevoordejevim algoritmom za problem križanke (gl. str. 69 v biltenu 2010).

Korak 3 je preprost: če za u v drevesu nimamo podanih staršev, potem je pikast le, če so ga prežarčili vesoljci (ta podatek pa najdemo v vhodni datoteki); sicer pa je mogoče tudi, da je pikavost podedoval po starših v skladu s pravili naloge.

Vprašanje je, kako ta postopek izvesti dovolj učinkovito. V koraku 2 bi lahko imeli še eno gnezdeno zanko, ki bi šla z u po vseh osebah, dokler ne bi naletela na tako, ki ustreza pogojem. Takšna zanka bi imela torej v najslabšem primeru $O(n)$ dela (če je $n = M + Z$ skupno število vseh ljudi v drevesu); in ker ima tudi zunanja zanka $O(n)$ iteracij (v vsaki iteraciji obdelamo po eno osebo), bi bila časovna zahtevnost celotnega postopka kar $O(n^2)$.

Razmišljamo lahko takole: neka oseba u postane primerna za določanje pikavosti (torej bi jo lahko na primer obdelali že kar v naslednji iteraciji glavne zanke) takoj, ko določimo pikavost obeh njenih staršev; če pa njena starša nista znana, moremo določiti njeno pikavost kadarkoli, lahko že na začetku postopka. Koristno bi bilo torej imeti neko vrsto ali seznam oseb, za katere vemo, da so že nared za določanje pikavosti. Za vsako osebo u vzdržujmo števec $s[u]$, ki pove, koliko njenim staršem še nismo določili pikavosti; ko nekomu določimo pikavost, zmanjšamo ta števec pri njegovih otrocih; ko pade nekemu otroku števec na 0, ga dodamo v seznam:

L := prazna vrsta;
za vsako osebo u:
  if poznamo u-jeve starše **then** $s[u] := 2$
  else $s[u] := 0$; dodaj u v vrsto L;
while L ni prazna:
  vzemi osebo u z začetka vrste in določi njeno pikavost;
  za vsakega u-jevega otroka v:
    $s[v] := s[v] - 1$;
    if $s[v] = 0$ **then** dodaj v v vrsto L;

Časovna zahtevnost te rešitve je zdaj le še $O(n)$, saj se notranja zanka (po v) izvede največ dvakrat za vsakega v (ker ima posamezni v največ dva starša).

Z vidika naše naloge je pri tem postopku rahlo nepraktično to, da moramo iti v notranji zanki po vseh otrocih trenutne osebe u. V vhodnih podatkih nimamo seznamov otrok, pač pa sezname staršev. Ena možnost bi torej bila, da pri branju vhodne datoteke predelamo podatke tako, da za vsakega človeka sestavimo seznam njegovih otrok. Druga možnost pa je, da z gornjim postopkom pregledamo drevo v nasprotni smeri, od kasnejših generacij k zgodnejšim; zdaj namesto seznamov otrok potrebujemo sezname staršev, kar je točno to, kar smo dobili v vhodni datoteki; namesto števila nepregledanih staršev $s[u]$ pa potrebujemo število nepregledanih otrok (spodnja rešitev ima za to tabelo nOtrok). Pri takšnem pregledu drevesa od spodaj navzgor seveda zdaj ne moremo sproti določati pikavosti, lahko pa si zapomnimo, v kakšnem vrstnem redu smo osebe pregledali (v spodnji rešitvi se ta vrstni red nabere v tabeli vrsta). Na koncu imamo vrstni red, v katerem se vsaka oseba pojavlja pred svojimi starši; če zdaj ta vrstni red pogledamo v obratni smeri, se bo v njem vsaka oseba pojavljala pred svojimi otroki, zato je takšen obrnjeni vrstni red primeren za določanje pikavosti.

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

```
#define MaxOseb 100

int main()
{
  int M, Z, t, p, i, j, k; char S[2];
  int starsi[MaxOseb][2], nOtrok[MaxOseb];
  int vrsta[MaxOseb], glava, rep;
  bool pikast[MaxOseb];

  /* Preberimo vhodne podatke. */
  fscanf(stdin, "%d %d %d %d", &M, &Z, &t, &p);
  for (i = 0; i < M + Z; i++)
    starsi[i][0] = −1, starsi[i][1] = −1, nOtrok[i] = 0, pikast[i] = false;
  while (t−− > 0) {
    fscanf(stdin, "%s %d %d %d", S, &i, &j, &k);
    i−−; j−−; k−−;
    if (S[0] == 'Z') i += M;
    starsi[i][0] = j; starsi[i][1] = k + M;
    nOtrok[j]++; nOtrok[k + M]++; }
  while (p−− > 0) {
    fscanf(stdin, "%s %d", S, &i);
    i−−; if (S[0] == 'Z') i += M;
    pikast[i] = true; }

  /* Topološko uredimo graf od potomcev proti prednikom. V vrsta[0..glava − 1] so
     že obdelane točke, v vrsta[glava..rep − 1] pa so tiste, ki jih še nismo
     obdelali, smo pa že obdelali njihove potomce. */
  glava = 0; rep = 0;
  for (i = 0; i < M + Z; i++) if (nOtrok[i] == 0) vrsta[rep++] = i;
  while (glava < rep)
    /* Vzemimo naslednjo osebo, i, iz vrste in poglejmo njena starša. */
    for (i = vrsta[glava++], j = 0; j < 2; j++) {
      k = starsi[i][j]; if (k < 0) continue;
      if (−−nOtrok[k] == 0) vrsta[rep++] = k; }

  /* Preglejmo graf v smeri od prednikov k potomcem in določimo pikavost. */
  for (i = rep − 1; i >= 0; i−−) {
    k = vrsta[i]; if (pikast[k]) continue;
    /* Otroci pikaste matere so pikasti. */
    j = starsi[k][1]; if (j >= 0 && pikast[j]) { pikast[k] = true; continue; }
    /* Hčere pikastega očeta so pikaste. */
    j = starsi[k][0]; if (j >= 0 && pikast[j] && k >= M) { pikast[k] = true; continue; }}

  /* Izpišimo rezultate. */
  for (i = 0; i < M + Z; i++) if (pikast[i])
    printf("%c %d\n", (i >= M) ? 'Z' : 'M', ((i >= M) ? i − M : i) + 1);
  return 0;
}
```

Mimogrede, problem, s katerim smo se ukvarjali v tej nalogi, je v teoriji grafov znan kot *topološko urejanje*. V usmerjenem grafu je topološki vrstni red tak vrstni red točk, pri katerem se začetno krajišče vsake povezave pojavlja v vrstnem redu prej kot končno krajišče te povezave. V našem primeru dobimo iz družinskega drevesa graf tako, da vpeljemo po eno točko za vsakega človeka, povezave $u \to v$ pa v tistih primerih, kjer je u eden od staršev osebe v.

20. Neskončne stopnice

Naša polja in dovoljeni premiki med njimi tvorijo usmerjen graf, v katerem gresta iz vsake točke po dve povezavi. Naloga pravzaprav zahteva, da v tem grafu poiščemo Hamiltonov cikel. Ker je problem Hamiltonovih ciklov v splošnem NP-težak in zanj ne poznamo učinkovitih algoritmov, lahko poskusimo primeren obhod našega grafa poiskati kar z rekurzijo. Na vsakem koraku imamo dve možni nadaljevanji poti, ki ju lahko raziščemo z dvema rekurzivnima klicema. Paziti pa moramo, da pri takem premiku ne pridemo v polje, ki smo ga nekoč prej na naši poti že obiskali. Spodnji program si v ta namen pomaga z globalno spremenljivko obiskana, v kateri označuje, katere točke je že obiskal. Trenutno pot hranimo v globalni spremenljivki pot; tako nam kot parameter pri rekurziji ni treba prenašati drugega kot naš trenutni položaj. Možne premike iz vsakega polja si lahko izračunamo vnaprej (spodnji program uporablja zato tabelo premiki), lahko pa bi jih računali tudi sproti.

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

#define MaxN 20

int n, polja[MaxN], premiki[MaxN][2], pot[MaxN + 1];
bool obiskana[MaxN];

/* Vpiše „polje“ v „pot[i]“ in skuša nadaljevati pot.
   Če uspe pot dokončati, vrne true, sicer pa false. */
bool NajdiPot(int i, int polje)
{
  int novo, j;
  pot[i] = polje; obiskana[polje] = true;
  for (j = 0; j < 2; j++) /* Preizkusimo oba možna premika iz trenutnega polja. */
  {
    novo = premiki[polje][j];
    if (i == n - 1 && novo == pot[0])
      return true; /* Pot lahko uspešno končamo s korakom v začetno polje, pot[0]. */
    else if (! obiskana[novo]) /* Poskusimo nadaljevati pot s korakom v „novo“. */
      if (NajdiPot(i + 1, novo))
        return true; /* Rekurzivni klic je uspešno našel preostanek poti. */
  }
  /* Označimo polje spet za neobiskano, preden se vrnemo iz rekurzivnega klica. */
  pot[i] = -1; obiskana[polje] = false;
  return false;
}

int main()
{
  int i, polje, st;

  /* Preberimo vhodne podatke. Pri tem pazimo na dejstvo, da v naših
     tabelah štejemo indekse polj v pozitivni smeri, v vhodni datoteki
     pa so polja navedena v negativni smeri (torej v smeri urinega kazalca). */
  fscanf(stdin, "%d", &n);
  for (i = 0; i < n; i++)
    fscanf(stdin, "%d", &polja[(n - i) % n]);

  /* Označimo polja za neobiskana in izračunajmo možne premike. */
  for (i = 0; i < n; i++) {
    obiskana[i] = false;
```

```
    st = polja[i];
    premiki[i][0] = (i + st - 1) % n;
    premiki[i][1] = (i - (st % n) - 1 + n) % n; }
  /* Poiščimo pot in jo izpišimo. */
  if (NajdiPot(0, 0)) {
    pot[n] = pot[0];
    for (i = 0; i < n; i++) {
      polje = pot[i]; st = polja[polje];
      if (pot[i + 1] == premiki[polje][0]) printf("+%d", st - 1);
      else printf("-%d", st + 1);
      printf(i == n - 1 ? "\n" : " "); }}
  return 0;
}
```

21. Najdaljši palindrom

Recimo, da smo dobili niz s, ki je dolg n znakov: $s = s_1 s_2 \ldots s_n$. V njem bi želeli najti najdaljši palindromni podniz, torej tak podniz $s_i s_{i+1} \ldots s_{j-1} s_j$, za katerega velja $s_i s_{i+1} \ldots s_{j-1} s_j = s_j s_{j-1} \ldots s_{i+1} s_i$. Zelo preprosta, vendar neučinkovita rešitev je, da gremo z dvema gnezdenima zankama po vseh i in j in pri vsakem paru (i, j) preverimo, ali je podniz $s_i s_{i+1} \ldots s_{j-1} s_j$ palindrom ali ne; če je res palindrom in če je daljši od najdaljšega doslej znanega (to dolžino hrani spremenljivka $d^\star$), si ga zapomnimo.

```
d* := 0;
for i := 1 to n do for j := i to n:
  l := i; r := j;
  while l < r:
    if s_l ≠ s_r then break;
    l := l + 1; r := r - 1;
  if l < r then continue; (* podniz s_i s_{i+1} ... s_j ni palindrom *)
  if j - i + 1 > d* then d* := j - i + 1;
```

To, ali je podniz $s_i s_{i+1} \ldots s_j$ palindrom, lahko preverimo na primer tako, da se z dvema števcema pomikamo po njem: l gre od leve proti desni, d pa od desne proti levi. Prvi in zadnji znak se morata ujemati, drugi in predzadnji ravno tako itd.; če kdaj opazimo neujemanje, lahko zanko prekinemo in vemo, da opazovani podniz ni palindrom. Če pa se števca srečata, ne da bi opazili neujemanje, potem vemo, da imamo palindrom.

Slabost tega algoritma je, da za preverjanje, ali je nek podniz palindrom, porabimo $O(n)$ časa, saj je število iteracij notranje zanke (while) sorazmerno z dolžino opazovanega niza. Ker je podnizov $O(n^2)$, je časovna zahtevnost celotnega postopka $O(n^3)$.

Do učinkovitejše rešitve pridemo, če upoštevamo, da se v daljših palindromih nujno skrivajo krajši palindromi: podniz $s_i s_{i+1} \ldots s_{j-1} s_j$ je palindrom natanko tedaj, če je $s_i = s_j$ in je tudi podniz $s_{i+1} \ldots s_{j-1}$ palindrom. Koristno bi bilo torej pregledovati podnize od krajših proti daljšim; če za $s_i \ldots s_j$ že vemo, da ni palindrom, potem nam podnizov oblike $s_{i-c} \ldots s_{j+c}$ sploh ni treba gledati, saj tudi oni gotovo niso palindromi.

Lahko se na primer postavimo v nek znak s_i in se vprašamo, kako dolg je najdaljši palindrom s središčem v s_i. To je torej palindrom oblike $s_{i-c}s_{i-c+1}\ldots s_{i+c-1}s_{i+c}$, dolg $2c+1$ znakov. Najdaljši tak palindrom poiščemo z zanko, ki primerja znake levo in desno od s_i in se ustavi, čim opazi neujemanje:

$c := 0$;
while $i-c \geq 1$ **and** $i+c \leq n$:
  if $s_{i-c} = s_{i+c}$ **then** $c := c+1$ **else break**;
zdaj vemo, da je najdaljši palindrom s središčem v s_i dolg $2c+1$ znakov;

Ta postopek lahko ponovimo za vsak i (od 1 do n) in tako poiščemo najdaljši palindrom lihe dolžine.

Podobno razmišljamo tudi za palindrome sode dolžine. Tak palindrom ima središče „med“ dvema znakoma, recimo s_{i-1} in s_i. Palindrom je potem oblike $s_{i-c}\ldots s_{i+c-1}$ in je dolg $2c$ znakov. Tudi takšne palindrome lahko iščemo z zanko po c:

$c := 0$;
while $i-c \geq 1$ **and** $i+c-1 \leq n$:
  if $s_{i-c} = s_{i+c-1}$ **then** $c := c+1$ **else break**;
zdaj vemo, da je najdaljši palindrom s središčem med s_{i-1} in s_i dolg $2c$ znakov;

Tudi ta postopek lahko ponovimo za vsak i in tako poiščemo najdaljši palindrom sode dolžine. Na koncu vrnemo daljšega od obeh palindromov. Pri vsakem i imamo le $O(n)$ dela, da poiščemo najdaljši palindrom s središčem v s_i, in $O(n)$ dela za najdaljši palindrom s sreidščem med s_{i-1} in s_i; časovna zahtevnost celotnega postopka je tako le še $O(n^2)$.

Oglejmo si zdaj še učinkovitejšo rešitev, pri kateri bo časovna zahtevnost celotnega postopka le $O(n)$.[22] Recimo, da se omejimo na palindrome lihe dolžine.[23] Naj bo p_i najdaljši tak palindrom s središčem v znaku i in naj bo njegova dolžina $2d_i-1$; torej je $p_i = s_{i-d_i+1}\ldots s_{i+d_i-1}$. Vrednosti d_i bomo računali po naraščajočem i. Recimo, da že poznamo $d_1, \ldots, d_k$. Kaj lahko zdaj povemo o palindromih, ki imajo središče v desni polovici palindroma p_k, torej o palindromih p_{k+i} za $1 \leq i < d_k$? Ker je p_k palindrom, je okolica znaka $k+i$ (v desni polovici palindroma p_k) prav taka kot okolica znaka $k-i$ (v levi polovici palindroma p_k), le obrnjena od desne proti levi. Če torej poznamo nek palindrom s središčem v $k-i$, bomo enak palindrom opazili tudi s središčem v $k+i$, vsaj dokler se vse skupaj dogaja znotraj palindroma p_k in ne seže čez njegove robove. Tako lahko torej vrednosti d_{k-1}, d_{k-2} in tako nazaj prenašamo v d_{k+1}, d_{k+2} in tako naprej, dokler ne pridemo do situacije, ko bi takšen palindrom s središčem v d_{k+i} segel čez desni rob palindroma p_k; takrat pa se moramo ustaviti in začeti primerjati znake od s_{k+d_k} naprej, da vidimo, kako daleč od roba palindroma p_k še lahko podaljšamo palindrom p_{k+i}.

[22] Ta postopek je opisal Johan Jeuring na naslovu `http://johanjeuring.blogspot.com/2007/08/finding-palindromes.html`; glej tudi opis Freda Akalina na `http://www.akalin.cx/longest-palindrome-linear-time`.

[23] Palindrome sode dolžine lahko obravnavamo na podoben način; lahko pa tudi predelamo prvotni niz $s = s_1 s_2 \ldots s_n$ v $s' := s_1\#s_2\#\ldots\#s_n$, pri čemer vzamemo za # poljuben znak, ki se ne pojavlja v s. Hitro se lahko prepričamo, da vsakemu palindromnemu podnizu prvotnega niza s, recimo $s_i s_{i+1} \ldots s_j$, ustreza nek palindromni podniz novega niza s' (namreč $s_i\#s_{i+1}\#\ldots\#s_j$) in obratno; pri tem pa so vsi palindromni podnizi niza s' lihe dolžine.

Primer: recimo, da imamo niz $s =$ `cabacabab` in smo že izračunali dolžine palindromov do vključno $k = 5$. Pri slednjem imamo $d_5 = 4$ in $p_5 =$ `abacaba`.

s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
c	a	b	a	c	a	b	a	b

d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
1	1	3	1	4				

Postavimo zdaj $i = 1$ in razmišljajmo o palindromih s središčem v $k+i$, torej pri s_6. Če je $d_6 \leq 3$, bi tak palindrom v celoti ležal znotraj p_5, torej bi morali popolnoma enak palindrom najti tudi s središčem v $k-i$, to je pri s_4. Ker vidimo, da je $d_4 = 1$, lahko zaključimo, da je tudi $d_6 = 1$ (in ne na primer 2 ali kaj še večjega).

Nato se premaknimo na $i = 2$; zdaj torej gledamo palindrome s središčem v s_7. Tak palindrom bi ležal v celoti znotraj p_5 le, če bi bil $d_7 \leq 2$. V tem primeru bi enak palindrom našli s središčem pri $k-i$, torej pri s_3. Ker vidimo, da je $d_3 = 3$, torej večji od 2, ne smemo takoj zaključiti, da je tudi $d_7 = 3$. Raje postavimo $d_7 = 2$; s tem smo dosegli desni rob palindroma p_5; nato pa postavimo k na 7 in s primerjanjem znakov s_{k-d_k} in s_{k+d_k} poglejmo, če lahko ta palindrom še kaj podaljšamo. V našem primeru bi primerjali s_9 in s_5, ugotovili neujemanje in torej ostali pri $d_7 = 2$.

V nadaljevanju bi pri $k = 7$, $i = 1$ postavili d_8 na isto vrednost kot d_6, to je 1. Ker smo s tem dosegli desni rob palindroma p_k (to je $p_7 =$ `aba`), postavimo k na 8 in primerjajmo znaka s_{k-d_k} in s_{k+d_k}; zdaj sta to s_7 in s_9; ker se ujemata, povečajmo d_8 na 2. To moramo ponavljati, dokler ne opazimo neujemanja ali pa pridemo do roba niza s; v našem primeru smo že trčili ob desni rob niza s, zato se takoj ustavimo.

Naslednji korak bi bil, da bi pri $k = 8$, $i = 1$ računali d_9. Vidimo lahko, da sme biti ta največ 1, če naj ne seže čez rob niza p_k (trenutno je to $p_8 =$ `bab`). Zato za d_9 ne smemo uporabiti vrednosti iz $d_{k-i} = d_7 = 2$, ampak ga postavimo na 1. Ker smo dosegli desni rob niza p_k, postavimo k na 8 in poskusimo dolžino d_9 še podaljšati, vendar takoj opazimo, da je ne moremo, ker smo prišli do konca niza s. Tako je k dosegel n, torej poznamo dolžine palindromov z vsemi možnimi središči in se lahko ustavimo.

Zapišimo ta postopek še s psevdokodo:

```
d_1 := 1; k := 1;
while k < n:
  i := 1;
  while i < d_k:
    (* Naj bo m največja dolžina palindroma s središčem v k + i,
       pri kateri še ne seže čez rob palindroma p_k. *)
    m := d_k − i;
    d_{k+i} := min{d_{k−1}, m};
    if d_{k+i} = m then break;
    i := i + 1;
  (* Če je i = d_k, pomeni, da se je gornja zanka iztekla brez stavka break.
     Do tega pride lahko le pri d_k = 1. *)
  if i = d_k then
```

if $k+i > n$ **then break**
else $d_{k+i} := 1$;
(* *Postavimo središče v* $k+i$ *in poglejmo, do kod še lahko podaljšamo ta palindrom.* *)
$k := k+i$;
while $k > d_k$ **and** $k + d_k \leq n$:
if $s_{k-d_k} = s_{k+d_k}$ **then** $d_k := d_k + 1$ **else break**;

O tem, da je časovna zahtevnost tega postopka le $O(n)$, se lahko prepričamo takole. Prva notranja zanka izvede največ eno iteracijo za vsako d_{k+i}, kajti tam, kjer se notranja zanka konča, postavimo nato novo vrednost k in ko z to notranjo zanko kasneje (v naslednji iteraciji zunanje zanke) spet začnemo, bomo nadaljevali pri $k+1$. Torej izvede ta notranja zanka vsega skupaj le $O(n)$ iteracij.

Pri drugi notranji zanki lahko podobno opazimo, da se izvede največ ena iteracija za vsako vsoto $k + d_k$, kajti ko se ta notranja zanka enkrat konča, to pomeni, da bomo v naslednji iteraciji zunanje zanke obravnavali palindrom p_k, ki sega od $k - d_k$ do $k + d_k$, in bomo do druge notranje zanke naslednjič prišli šele takrat, ko bomo na nekem takem novem položaju (na katerega se postavimo z vrstico „$k := k + i$“), na katerem sega tamkajšnji palindrom vsaj do konca dosedanjega, če ne še čez njegov rob. Ko torej naslednjič pridemo do druge notranje zanke, bo le-ta nadaljevala tam, kjer je prejšnjič (pri prejšnji iteraciji zunanje zanke) končala.

Vsaka od notranjih zank torej izvede vsega skupaj (pri vseh iteracijah zunanje zanke) največ $O(n)$ iteracij (in ima z vsako iteracijo $O(1)$ dela). Tudi zunanja zanka izvede $O(n)$ iteracij in (če odmislimo tisto, kar počne v notranjih zankah) ima z vsako iteracijo $O(1)$ dela. Tako torej vidimo, da je časovna zahtevnost vseh zank skupaj (in s tem celotnega postopka) le $O(n)$.

Mimogrede, besedilo naloge omenja palindrome v *Sonetnem vencu*, vendar po naših opažanjih to ni ravno najbolj posrečena izbira. Najdaljši palindromni podniz, ki smo ga uspeli najti v njem, je „ni ene in“ v prvem sonetu: „Vse misli 'zvirajo 'z ljubezni ene/ in kjer ponoči v spanju so zastale,/ zbudé se, ko spet zarja noč prežene.“

22. Skakač na šahovnici

Šahovnico lahko predstavimo z grafom, v katerem je po ena točka za vsako polje. Med dvema točkama naj obstaja neusmerjena povezava natanko v primeru, ko lahko skakač skoči od enega polja do drugega z eno potezo.

Naj bo t_0 začetno polje našega skakača, $t_1, \ldots, t_4$ pa polja, ki jih moramo obiskati. Za vse t_i poženimo iskanje najkrajših poti od njih do vseh ostalih polj (npr. z iskanjem v širino). Zdaj torej poznamo za vsak i in za vsako polje p dolžino najkrajše poti od t_i do p — označimo jo z $d(t_i, p)$.

Naloga ne predpisuje, v kakšnem vrstnem redu mora skakač obiskati dane štiri točke t_1, t_2, t_3, t_4, zato lahko preizkusimo vseh $4! = 24$ možnih vrstnih redov in vrnemo najkrajšega od tako dobljenih sprehodov. Če nam permutacija π pove, v kakšnem vrstnem redu obiskujemo točke $t_1, \ldots, t_4$, je skupna dolžina takega sprehoda $d(t_0, t_{\pi(1)}) + d(t_{\pi(1)}, t_{\pi(2)}) + d(t_{\pi(2)}, t_{\pi(3)}) + d(t_{\pi(3)}, t_{\pi(4)})$. Vse dolžine, ki jih potrebujemo pri tem razmisleku, smo izračunali v prejšnjem odstavku, tako da je iskanje najkrajšega sprehoda zdaj preprosto in poceni.

Zapišimo to rešitev še v C-ju. Koordinate polj bomo predstavili s števili od 0 do 7. Najprej si pripravimo funkcijo IskanjeVSirino, ki z iskanjem v širino poišče najkrajše poti od polja (x0, y0) do vseh ostalih polj in jih shrani v tabelo dolzine:

```
void IskanjeVSirino(int x0, int y0, int dolzine[8][8])
{
  const int dx[8] = { −2, −2, −1, −1, 1, 1, 2, 2 }; /* 8 možnih premikov skakača */
  const int dy[8] = { −1, 1, −2, 2, −2, 2, −1, 1 };
  int x, y, xx, yy, i, p, vrsta[64], glava = 0, rep = 0;

  for (y = 0; y < 8; y++) for (x = 0; x < 8; x++) dolzine[y][x] = 99;
  dolzine[y0][x0] = 0; vrsta[rep++] = y0 * 8 + x0;

  while (glava < rep) {
    x = vrsta[glava] % 8; y = vrsta[glava] / 8; glava++;

    /* Najkrajšo pot do (x, y) zdaj poznamo; poglejmo, v katera polja
       se da priti iz nje v eni potezi. */
    for (p = 0; p < 8; p++) {
      xx = x + dx[p]; yy = y + dy[p];
      if (xx < 0 || yy < 0 || xx >= 8 || yy >= 8) continue;
      if (dolzine[y][x] + 1 >= dolzine[yy][xx]) continue;

      /* Našli smo najkrajšo pot do (xx, yy). */
      dolzine[yy][xx] = dolzine[y][x] + 1; vrsta[rep++] = yy * 8 + xx; }}
}
```

Vse možne vrstne rede obiskovanja točk $t_1, \ldots, t_4$ lahko preizkusimo z rekurzijo. Spodnji podprogram poišče najkrajšo tako pot, ki se začne v t_z in obišče tiste točke t_i (za $i = 1, \ldots, 4$), za katere je bit i v obiskane ugasnjen. Koordinate točk t_i ji podamo v tabelah tx in ty, dolžine najkrajših poti pa v tabeli dolzine (pri čemer je dolzine[i][y][x] najkrajša pot od točke t_i do točke (x, y)).

```
int Rekurzija(int z, int obiskane, int tx[5], int ty[5], int dolzine[5][8][8])
{
  int naj = −1, kand, i;
  for (i = 1; i <= 4; i++) if (((obiskane >> i) & 1) == 0) {
    kand = dolzine[z][ty[i]][tx[i]] + Rekurzija(i, obiskane | (1 << i), tx, ty, dolzine);
    if (naj < 0 || kand < naj) naj = kand; }
  if (naj < 0) return 0; /* Vse je že obiskano. */
  else return kand; /* Vrnimo najkrajšo pot. */
}
```

Po vsem tem je glavni podprogram zelo preprost:

```
int Skakac(int tx[5], int ty[5])
{
  int dolzine[5][8][8], i;
  for (i = 0; i <= 4; i++) IskanjeVSirino(tx[i], ty[i], dolzine[i]);
  return Rekurzija(0, 0, tx, ty, dolzine);
}
```

23. Flomastri

V nalogi sicer nastopajo tri barve, vendar ena na drugo nič ne vplivajo; potreb po flomastrih barve 1 pač ne moremo reševati s flomastri barve 2 ali 3 in podobno. Zato lahko vsako barvo obravnavamo ločeno od ostalih; če se pri vsaj eni barvi

problem izkaže za nerešljiv, bomo na koncu izpisali -1, sicer pa bomo rešitve za posamezne barve združili v rešitev prvotnega problema. V nadaljevanju te rešitve se torej omejimo na eno barvo; pri tem bomo seveda tudi od flomastrov gledali le tiste, ki so prave barve.

Opazimo lahko, da je za naš problem pomembna le skupna poraba trenutne barve, ne pa tudi to, kako je ta poraba razporejena po urah. Če si ogledamo primer iz besedila naloge: potrebe po barvi 1 znašajo 3 enote prvo uro in 8 enot drugo uro, pokrili pa smo jih s flomastroma dolžine 4 in 7; znotraj druge ure zato neizogibno uporabimo oba flomastra, obenem pa tudi neizogibno vsaj enega od obeh flomastrov uporabljamo v obeh urah. Naj bo torej p skupna poraba (ki jo dobimo tako, da seštejemo porabo po urah, kot je podana v vhodni datoteki).

V naši nalogi se zdaj pravzaprav skriva znani problem nahrbtnika (*backpack problem*): pri (*b*) v klasični obliki, kjer lahko vsak predmet bodisi sprejmemo bodisi zavrnemo, pri (*a*) pa v „zvezni“ različici, kjer smemo posamezni predmet vzeti tudi delno, ne le v celoti.

(*a*) Pri lažji različici naloge lahko razmišljamo takole: za vsak flomaster izračunajmo ceno na enoto dolžine, c_i/d_i. Flomastre uredimo naraščajoče po tem razmerju in jih v tem vrstnem redu oštevilčimo. Najcenejši izbor dobimo tako, da vzamemo prvih nekaj flomastrov v tem vrstnem redu — namreč toliko, kolikor jih je najmanj treba, da njihova skupna dolžina doseže (ali preseže) p. Vsakega od tako izbranih flomastrov bomo porabili v celoti, razen mogoče zadnjega med njimi.

Prepričajmo se, da je to res najcenejši izbor. Recimo, da imamo n flomastrov; izbor lahko opišemo z n-terico števil $a = (a_1, \ldots, a_n)$, pri čemer je $a_i \in [0, d_i]$ število, ki pove, koliko enot dolžine flomastra i se porabi v našem izboru. Veljaven izbor je seveda le tak, pri katerem se dolžine seštejejo v p. Ker smo izbirali flomastre po vrsti, je naš konkretni izbor oblike $a = (d_1, d_2, \ldots, d_{k-1}, a_k, 0, \ldots, 0)$, pri čemer flomaster k, ki je zadnji v našem izboru, mogoče ni porabljen v celoti (ampak le delno: $a_k = p - \sum_{i=1}^{k-1} d_i$). Recimo, da ta izbor ni najcenejši, torej obstaja nek še cenejši izbor $a' = (a'_1, \ldots, a'_k)$; če je možnih več enako dobrih cenejših izborov, vzemimo med njimi leksikografsko največjega (torej tistega z največjim a'_1, med njimi tistega z največjim a'_2 in tako naprej).

Poiščimo najmanjši i, pri katerem se izbora razlikujeta, torej da je $a_i \neq a'_i$. Gotovo je $i \leq k$, kajti če bi do razlike prišlo šele kasneje (kjer ima a same ničle), bi to pomenilo, da je skupna dolžina napisanega pri a' daljša kot pri a, torej a' sploh ni veljaven izbor. Če je $i < k$, je razlika $a_i \neq a'_i$ možna le tako, da je $a'_i < a_i$, saj ima a_i največjo dovoljeno vrednost za ta flomaster (namreč d_i). Podobno velja tudi, če je $i = k$; kajti če se a in a' v prvih $k-1$ flomastrih ujemata, pri k pa je $a'_k > a_k$, potem bo vsota dolžin pri a' večja kot pri a (saj ima v nadaljevanju a same ničle), torej bi bil tak a' neveljaven.

Zdaj torej vidimo, da pri prvem neujemanju (na indeksu i) velja $a'_i < a_i$. To pa pomeni, da mora nekje kasneje obstajati vsaj en indeks $j > i$, pri katerem je $a'_j > a_j$ (saj bi imel drugače izbor a' premajhno vsoto dolžin). Naj bo $\triangle = \min\{a_i - a'_i, a'_j - a_j\}$. Če zdaj v izboru a' povečamo a'_i za $\triangle$ in zmanjšamo a'_j za $\triangle$, izbor ostane veljaven in je bodisi cenejši kot prej (če ima flomaster i manjšo ceno na enoto dolžine kot j) bodisi stane enako kot prej, vendar je leksikografsko večji (ker se je a'_i povečal). V vsakem primeru smo torej prišli v protislovje s predpostavko,

da je a' najcenejši izbor (in leksikografsko največji med takimi).

(*b*) Pri težji različici naloge moramo izbrati nekaj flomastrov tako, da je njihova skupna dolžina vsaj p, znotraj te omejitve pa naj bo njihova skupna cena čim manjša. Naj bo $D = \sum_{i=1}^{n} d_i$ skupna dolžina vseh flomastrov, $C = \sum_{i=1}^{n} c_i$ pa njihova skupna cena. Recimo, da razmišljamo o nekem izboru flomastrov s skupno dolžino d (upajmo, da je $d \geq p$) in skupno ceno c (upajmo, da čim manjšo). Ta izbor lahko pogledamo tudi z druge strani: namesto da bi izbirali, katere flomastre uporabiti, lahko izbiramo flomastre, ki jih *ne* bomo uporabili. V primeru našega izbora imajo neuporabljeni flomastri skupno dolžino $D - d$ (ki mora biti $\leq D - p$, če hočemo, da bo $d \geq p$) in skupno ceno $C - c$ (ki mora biti čim višja, če hočemo, da bo c čim nižja).

To pa je zdaj ravno klasični problem $\{0,1\}$-nahrbtnika: izbrati hočemo nekaj flomastrov (tiste, ki jih ne bomo uporabili) in to tako, da njihova skupna dolžina ne bo presegla $D - p$, znotraj te omejitve pa naj bo njihova skupna cena čim višja.

Naloge se lahko lotimo na primer z dinamičnim programiranjem: imejmo tabelo T, v kateri nam element $T[k,d]$ pove najvišjo ceno, ki jo lahko dosežemo, če izbiramo flomastre le iz prvih k in če njihova skupna dolžina ne sme preseči d. Vrednost $T[k,d]$ lahko računamo z rekurzivnim razmislekom; če flomaster k izberemo, nam ostane problem s $k-1$ flomastri in omejitvijo dolžine $d-d_k$; če pa flomastra k ne izberemo, nam ostane problem s $k-1$ flomastri in nespremenjeno omejitvijo dolžine d. Torej je

$$T[k,d] = \min\{T[k-1,d], c_k + T[k-1,d-d_k]\}.$$

Te vrednosti lahko računamo sistematično po naraščajočem k in pri vsakem k po naraščajočih d:

for $d := 0$ **to** $D - p$ **do** $T[0,d] := 0$;
for $k := 1$ **to** n:
  for $d := 0$ **to** $D - p$:
    if $d \geq d_k$ **then** $T[k,d] := \min\{T[k-1,d], c_k + T[k-1,d-k]\}$; $\qquad (\star)$
    else $T[k,d] := T[k-1,d]$;

Na koncu je cena najboljšega izbora v $T[n, D-p]$. V resnici bi morali rešitev dopolniti še s tem, da bi si pri vsakem (k,d) zapomnili, kako je bil dosežen izbor $T[k,d]$ — ali smo flomaster k pri njem izbrali ali ne (z drugimi besedami, katera od obeh možnosti je dala minimum v vrstici $(\star)$). To bi nam omogočilo, da bi na koncu rekonstruirali najboljši izbor, ne le njegove cene.

Opisana rešitev ni prikladna, če dolžine flomastrov niso celoštevilske (saj jih potem ne moremo uporabljati kot indekse v tabeli T) ali pa če so zelo velika cela števila (tedaj je tabela T zelo velika, algoritem pa zelo počasen). Vsako vrstico tabele T, torej $T[k,\cdot]$ za nek fiksen k, si lahko predstavljamo kot funkcijo, ki za vsako možno dolžino d pove najvišjo ceno, ki jo je mogoče doseči z izborom flomastrov s skupno dolžino največ d. Ta funkcija je stopničasta (odsekoma konstantna in nepadajoča), zato jo lahko predstavimo tudi s seznamom parov (d,c), v katerih nastopajo le tiste dolžine d, pri katerih se vrednost funkcije spremeni (novo vrednost pa podaja število c). Seznam T_k lahko zdaj pripravimo tako, da vzamemo kopijo seznama T_{k-1}, prištejemo vsem parom v njem (d_k, c_k) in ga zlijemo s prvotnim T_{k-1} (pri zlivanju pobrišemo pare, pri katerih bi vrednost funkcije padla). Tako

dobljeni seznami gotovo nimajo po več kot 2^n parov, ne glede na to, kakšne so dolžine naših flomastrov. V najslabšem primeru je torej tudi ta postopek lahko precej neučinkovit, vendar pa res učinkovite rešitve do nadaljnjega ne poznamo, saj je problem nahrbtnika NP-težak.

24. Zbiranje sličic

Označimo z $m_i := |A_i|$ število sličic, ki jih ima zbiratelj i. Število sličic, ki si jih i in j lahko izmenjata, bomo lažje izračunali, če bomo najprej pogledali, koliko sličic jima je skupnih:

$$\begin{aligned} z_{ij} &= \min\{|A_i - A_j|, |A_j - A_i|\} \\ &= \min\{|A_i| - |A_i \cap A_j|, |A_j| - |A_i \cap A_j|\} \\ &= \min\{|A_i|, |A_j|\} - |A_i \cap A_j| \\ &= \min\{m_i, m_j\} - |A_i \cap A_j|. \end{aligned}$$

To, kako prešteti slike v preseku $A_i \cap A_j$, je odvisno od tega, kako so naše množice predstavljene. Če je na primer vsaka množica predstavljena z urejenim seznamom, lahko presek določimo v $O(m_i + m_j)$ časa z zlivanjem obeh seznamov ali pa v $O(m_i \log m_j)$ časa tako, da vsak element prvega seznama z bisekcijo poiščemo v drugem seznamu (to je koristno, če je prvi seznam veliko krajši od drugega; če je obratno, ju lahko seveda v mislih zamenjamo). Če je vsaka množica predstavljena z razpršeno tabelo, v kateri lahko v $O(1)$ časa za poljubno sličico preverimo, ali je v množici ali ne, se lahko sprehodimo po vseh elementih ene od množic A_i in A_j (po možnosti tiste, ki je manjša od druge) in za vsakega preverimo, ali leži tudi v drugi množici.

Ta postopek lahko zdaj ponovimo za vsak par (i, j) in nato pri vsakem i pogledamo, kateri j je dal najmanjšo vrednost z_{ij}. Če je izračun posamezne z_{ij} porabil $O(m_i + m_j)$ časa, bo zdaj časovna zahtevnost celotnega postopka približno

$$\begin{aligned} &\textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} (m_i + m_j) \\ &= \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} m_i + \sum_{i=1}^{n} \sum_{j=1}^{n} m_j \\ &= 2n \textstyle\sum_{i=1}^{n} m_i. \end{aligned}$$

Množenju z 2 se sicer zlahka izognemo, če od parov (i, j) in (j, i) obravnavamo samo po enega (saj vemo, da je $z_{ij} = z_{ji}$). Označimo dobljeno zahtevnost s $T_1 = n \sum_{i=1}^{n} m_i$.

Malo boljšo rešitev dobimo, če si pripravimo kazalo (*inverted index*), v katerem imamo za vsako sličico x seznam L_x zbirateljev, ki to sličico že imajo:

```
for x := 1 to s do L_x := prazen seznam;
for i := 1 to n:
  za vsako x ∈ A_i:
    dodaj i v seznam L_x;
```

Takšni seznami pridejo prav pri računanju velikosti presekov med vsemi pari množic. Naj bo $p_{ij} = |A_i \cap A_j|$. Računamo jih lahko takole:

for $i := 1$ **to** n **do for** $j := 1$ **to** n **do** $p_{ij} := 0$;
for $x := 1$ **to** s:
  za vsakega zbiratelja i s seznama L_x:
    za vsakega zbiratelja j s seznama L_x:
      $p_{ij} := p_{ij} + 1$;

Za vsako sličico $x \in A_i \cap A_j$ bomo prej ali slej opazili, da sta tako i kot j na seznamu L_x, in bomo takrat povečali p_{ij} za 1. Na koncu tega postopka nam torej p_{ij} pove, koliko je sličic v $A_i \cap A_j$, prav to pa smo tudi želeli.

Ko imamo enkrat vse p_{ij}, lahko zelo preprosto računamo tudi z_{ij}, kot smo videli že zgoraj: $z_{ij} = \min\{m_i, m_j\} - p_{ij}$.

Časovna zahtevnost tega postopka je $O(T_2)$ za $T_2 = \sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij}$. Hitro se lahko prepričamo, da to ni nič slabše od prejšnjega postopka, lahko je celo precej bolje: p_{ij} je velikost preseka množic A_i in A_j, ta presek pa ne more biti večji od množic A_i in A_j samih. Torej imamo $p_{ij} \le m_i$ in zato

$$\begin{aligned} T_2 &= \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij} \\ &\le \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} m_i \\ &= n \textstyle\sum_{i=1}^{n} m_i = T_1, \end{aligned}$$

pri čemer imamo zdaj v zadnji vrstici ravno zahtevnost našega prejšnjega postopka. Torej je novi postopek načeloma res hitrejši in to tem bolj, čim manjši so preseki med našimi množicami.

Če so preseki veliki, lahko naš postopek zastavimo tudi malo drugače. Namesto presekov množic glejmo razlike: naj bo $r_{ij} = |A_i - A_j|$. Iz tega ni težko računati z_{ij}, saj je $z_{ij} = \min\{r_{ij}, r_{ji}\}$. Za vsak x si poleg seznama L_x pripravimo še seznam L'_x, na katerem naj bodo vsi tisti zbiratelji, ki slike x še nimajo. Zdaj lahko računamo vse r_{ij} na zelo podoben način, kot smo prej računali vse p_{ij}:

for $i := 1$ **to** n **do for** $j := 1$ **to** n **do** $r_{ij} := 0$;
for $x := 1$ **to** s:
  za vsakega zbiratelja i s seznama L_x:
    za vsakega zbiratelja j s seznama L'_x:
      $r_{ij} := r_{ij} + 1$;

Časovna zahtevnost tega postopka je $O(T_3)$ za $T_3 = \sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij}$. To je naprej enako

$$\begin{aligned} T_3 &= \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij} \\ &= \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} (m_i - p_{ij}) \\ &= \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} m_i - \sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij} \\ &= n \textstyle\sum_{i=1}^{n} m_i - \sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij} = T_1 - T_2. \end{aligned}$$

Vidimo torej, da tudi ta postopek ni nič slabši od prvega; še več, vsi trije so povezani s formulo $T_1 = T_2 + T_3$. Tega, ali je manjši T_2 ali T_3, pa v splošnem ne moremo reči, saj je odvisno od velikosti presekov med množicami A_i.

25. Črni kovčki

Oštevilčimo tako agente kot kovčke s števili od 1 do n (številka kovčka nam seveda pove to, kateri agent bi ga moral imeti, če kovčki ne bi bili pomešani). Začetni

razpored kovčkov med agente si lahko zdaj predstavljamo kot permutacijo; to je tabela, ki ima za vsakega agenta po en stolpec, v njem pa je najprej številka agenta in nato številka kovčka, ki ga trenutno ima ta agent. Primer:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 6 & 3 & 1 & 4 & 2 \end{pmatrix}.$$

Ta permutacija nam pove, da ima agent 1 kovček 5, agent 2 ima kovček 6, agent 3 ima kovček 3 in tako naprej.

Permutacijo lahko predstavimo tudi z enim ali več cikli. Agent 1 ima kovček, ki bi moral v resnici pripadati agentu 5; poglejmo zdaj, čigav kovček ima agent 5. Vidimo, da je to kovček, ki bi moral pripadati agentu 4; agent 4 pa ima trenutno kovček, ki bi moral pripadati agentu 1. Tako smo dobili cikel, ki ga lahko zapišemo kot (1 4 5). Podobno imamo v gornji permutaciji še cikla (2 6) in (3).

Izkaže se, da je takšna razdelitev agentov na cikle zelo koristna, če hočemo razmišljati o menjavah kovčkov. Kaj se zgodi z našimi cikli, če si dva agenta, na primer a in b, zamenjata kovčke? Ločimo dva primera, odvisno od tega, ali sta bila a in b pred menjavo v istem ciklu ali ne.

(1) Recimo, da sta bila v istem ciklu. Ker je cikel pač cikličen, je vseeno, pri katerem agentu ga začnemo pisati, torej ga lahko začnemo pri a in ga zapišemo v obliki $(a\ x_1\ \ldots\ x_s\ b\ y_1\ \ldots\ y_t)$. Pri tem je seveda mogoče, da je kakšna od skupin agentov $x_1, \ldots, x_s$ in $y_1, \ldots, y_t$ tudi prazna (ali pa celo obe).

Ta cikel si bomo lažje predstavljali, če ga napišemo kot permutacijo. Stolpcev za agente, ki niso del opazovanega cikla, ne bomo pisali, saj na naš cikel (tudi po tem, ko si bosta a in b izmenjala svoja kovčka) ne bodo vplivali.

$$\begin{pmatrix} a & x_1 & \cdots & x_s & b & y_1 & \cdots & y_t \\ x_1 & x_2 & \cdots & b & y_1 & y_2 & \cdots & a \end{pmatrix}.$$

Agent a ima torej zdaj kovček x_1, agent b pa kovček y_1. Kaj se zgodi, ko si kovčka izmenjata?

$$\begin{pmatrix} a & x_1 & \cdots & x_s & b & y_1 & \cdots & y_t \\ y_1 & x_2 & \cdots & b & x_1 & y_2 & \cdots & a \end{pmatrix}$$

Če poskusimo to novo permutacijo pretvoriti nazaj v cikle, dobimo zdaj dva cikla namesto enega: $(a\ y_1\ y_2\ \ldots\ y_t)$ in $(b\ x_1\ x_2\ \ldots\ x_s)$. Tako torej vidimo, da je bil učinek menjave ta, da nam je cikel razpadel na dva krajša cikla.

(2) Oglejmo si še primer, ko sta bila agenta v različnih ciklih. Podobno kot prej lahko zapišemo oba cikla tako, da sta naša agenta a in b na začetku: agent a je bil na primer v ciklu $(a\ y_1\ y_2\ \ldots\ y_t)$, agent b pa v ciklu $(b\ x_1\ x_2\ \ldots\ x_s)$. Če to zapišemo s permutacijo in nato izvedemo menjavo kovčkov med agentoma a in b, lahko hitro opazimo, da imamo opravka z istimi razporedi kot pri (1), le da smo spremembo izvedli v nasprotni smeri. Iz dveh ciklov nam pri taki menjavi nastane en sam daljši cikel $(a\ x_1\ \ldots\ x_s\ b\ y_1\ \ldots\ y_t)$.

Tako torej vidimo, da se število ciklov po vsaki zamenjavi spremeni za 1. Na koncu bi radi prišli do razporeda, v katerem ima vsak agent svoj kovček; tedaj tvori vsak agent sam svoj cikel dolžine 1, torej bomo imeli na koncu n ciklov. Če smo

imeli na začetku recimo k ciklov, potem vidimo, da lahko do končnega razporeda pridemo z $n - k$ menjavami, z manj pa ne.

Naš algoritem mora torej predvsem prešteti cikle v začetnem razporedu kovčkov med agente. Začnimo pri poljubnem agentu in se sprehajajmo naprej po ciklu, dokler ne pridemo nazaj do agenta, pri katerem smo začeli; nato se postavimo na poljubnega agenta, ki ga nismo obiskali v okviru prvega cikla, in z enakim postopkom obiščimo njegov cikel; tako nadaljujemo, dokler niso obiskani vsi agentje. Koristno je imeti tabelo, v kateri si označujemo, katere agente smo že obiskali in katerih še ne; v spodnjem postopku je to tabela v. Kovček, ki ga (pri razporedu iz vhodnih podatkov) drži agent a, označimo s $\pi(a)$.

for $a := 1$ **to** n **do** $v[a] :=$ **false**;
$k := 0$;
for $a := 1$ **to** n:
  if not $v[a]$:
    $k := k + 1$;
    while not $v[a]$:
      $v[a] :=$ **true**; $a := \pi(a)$;
izpiši $n - k$;

Razmislimo še o težji različici naloge, pri kateri moramo izpisati primerno zaporedje menjav, ne le njihovega števila. Da bomo dosegli želeno ciljno stanje z najmanjšim številom menjav, mora vsaka menjava povečati število ciklov za 1, torej morata v menjavi vedno nastopiti agenta, ki sta doslej pripadala istemu ciklu. Recimo, da imamo cikel oblike $(a\ x_1\ \ldots\ x_s)$. Na njem lahko na primer izvedemo menjavo med a in x_1, s čimer nam cikel razpade na (x_1) in $(a\ x_2\ \ldots\ x_s)$. Agent x_1 torej zdaj že ima pravi kovček, na preostanku cikla pa lahko ponovimo enak razmislek, izvedemo menjavo med a in x_2 in tako naprej. Cikel bomo torej razdrobili tako, da bomo izvedli menjave med a in vsemi ostalimi agenti tega cikla. Naš zgoraj opisani postopek, s katerim smo se sprehajali po ciklih (in jih šteli), lahko z majhno spremembo izvaja tudi te zamenjave:

for $a := 1$ **to** n **do** $v[a] :=$ **false**;
for $a := 1$ **to** n:
  if not $v[a]$:
    $b := \pi(a)$; $v[a] :=$ **true**;
    while $b \neq a$:
      izpiši menjavo med a in b;
      $v[b] :=$ **true**; $b := \pi(b)$;

Če lahko razporeditev π po vsaki menjavi popravljamo, tako da odraža stanje po zadnji menjavi (namesto začetnega stanja iz vhodne datoteke), pa tabele v sploh ne potrebujemo:

for $a := 1$ **to** n:
  while $\pi[a] \neq a$:
    $b := \pi[a]$;
    izpiši menjavo med a in b;
    $\pi[a] := \pi[b]$; $\pi[b] := b$;

26. Tihotapec

Vprašanje najhitrejšega pobega iz labirinta lahko prevedemo na vprašanje najkrajše poti v grafu. Stanje zapornika lahko opišemo s trojico (x, y, r), pri čemer sta (x, y) koordinati celice, v kateri se nahaja, r pa je število sten, ki jih je doslej razbil. Podatek r je koristen zaradi omejitve v besedilu naloge, ki pravi, da smemo stene razbiti le p-krat. Za vsako trojico (x, y, r) bomo imeli v grafu po eno točko (za $1 \leq x \leq w$, $1 \leq y \leq h$ in $0 \leq r \leq p$), poleg tega pa še posebno točko z, ki predstavlja zunanjost labirinta.

Povezave definirajmo v našem labirintu takole: če sta (x, y) in (x', y') dve sosednji točki (torej če je $x = x'$ in $y = y' \pm 1$ ali pa $x = x' \pm 1$ in $y = y'$), naj za vsak r obstaja povezava od (x, y, r) do (x', y', r); dolžina vsake od teh povezav je bodisi a (če je med celicama (x, y) in (x', y') prehod) bodisi b (če prehoda ni). Poleg tega pa, če med celicama ni prehoda, dodajmo še povezave od (x, y, r) do $(x', y', r + 1)$ z dolžino c; te povezave potrebujemo za $0 \leq r < p$, pri $r = p$ pa seveda ne (ker če smo že razbili p sten, zdaj ne smemo razbiti še ene).

Če je celica (x, y) na robu labirinta, dodajmo še povezave iz vseh (x, y, r) v točko z. Če je med celico (x, y) in zunanjostjo kakšen prehod, naj bo dolžina teh povezav a, sicer pa c (razen pri $r = p$, kjer naj bo dolžina povezave b).

V tako dobljenem grafu predstavlja vsaka povezava en možen premik zapornika, dolžina te povezave pa ponazarja trajanje tega premika. V tem grafu moramo zdaj poiskati najkrajšo pot od točke $(x_0, y_0, 0)$ do z, če je (x_0, y_0) začetni položaj zapornika. Dolžina te poti nam pove najkrajši čas, v katerem lahko zapornik pobegne iz labirinta. Če sledimo točkam na tej poti, lahko tudi rekonstruiramo potek njegovega pobega; videli bomo tudi, kje mora razbiti stene (namreč tam, kjer imamo na naši poti korak iz (x, y, r) v $(x', y', r + 1)$).

Za iskanje najkrajših poti v grafu lahko uporabimo kakšnega od dobro znanih algoritmov, na primer Dijsktrovega.

27. Vlaki

Za vsak časovni trenutek t (od 1 do T) naj bo A_t množica vlakov, ki ob času t obiščejo našo postajo (to so torej tisti vlaki i, pri katerih obstaja neko celo število $k \geq 0$, za katero velja $a_i + kb_i = t$). Naloga zdaj v bistvu sprašuje, kako iz vsake A_t izbrati po en vlak tako, da bo na koncu čim več vlakov izbranih.

Ta problem lahko predstavimo z dvodelnim grafom: na levi strani imejmo n točk $V = \{v_1, \ldots, v_n\}$, ki predstavljajo vlake, na desni pa T točk $U = \{u_1, \ldots, u_T\}$, ki predstavljajo časovne trenutke. Povezava (v_i, u_t) naj obstaja natanko v primeru, če vlak i obišče našo postajo ob času t (torej če $i \in A_t$). Dejavnost našega vandala si lahko zdaj predstavljamo kot izbiranje neke podmnožice povezav: to, da izberemo povezavo (v_i, u_t) v našo podmnožico, pomeni, da je vandal ob času t narisal grafit na vlak i. Ker ni nobene koristi od tega, da bi na isti vlak narisali več grafitov (ob različnih časih), se lahko omejimo na izbore, v katerih vsak vlak nastopa največ enkrat. Podobno tudi vemo, da v našem izboru gotovo vsak čas nastopa največ enkrat, saj naloga pravi, da če se v istem trenutku na postaji pojavi več vlakov, lahko narišemo grafit na največ enega od njih. Tako torej vidimo, da za naš izbor povezav velja, da nimata nobeni dve izbrani povezavi nobenega skupnega krajišča, niti na levi strani (pri vlakih) niti na desni (pri časih).

Tako smo prevedli našo nalogo na vprašanje, kako v dobljenem dvodelnem grafu izbrati čim več povezav, ne da bi imeli kakšni dve povezavi kakšno skupno krajišče. Takemu izboru povezav pravijo v teoriji grafov *ujemanje* (*matching*) in zanj obstaja več algoritmov. Oglejmo si na primer algoritem dopolnilnih poti.

Našemu ujemanju recimo M; začeli bomo s prazno množico in jo postopoma povečevali. Povezavam, ki so trenutno v M, recimo *izbrane*; točki, ki ni krajišče nobene izbrane povezave, pa recimo *prosta* točka. V grafu poiščimo poljubno *dopolnilno pot* (*augmenting path*): to je pot, ki se začne v neki prosti točki, konča v neki prosti točki in ki izmenično vsebuje izbrane in neizbrane povezave. Zdaj za vsako povezavo s te poti naredimo naslednje: če ta povezava ni v M, jo vanjo dodajmo, sicer pa jo pobrišimo iz M. Pokazati je mogoče, da po tej spremembi množica M ostane veljavno ujemanje in vsebuje za eno povezavo več kot prej. Postopek se ustavi, ko v grafu ni nobene dopolnilne poti več; pokazati je mogoče, da je takratna množica M res največje možno ujemanje.

Dopolnilne poti lahko iščemo na primer z iskanjem v širino. Iz naše definicije dopolnilne poti sledi, da mora imeti dopolnilna pot liho mnogo povezav, torej se mora začeti na eni strani grafa, končati pa na drugi strani. Zato se lahko omejimo na poti, ki se začnejo v V, končajo pa v U (če bi dovolili še obratno možnost, bi dobili iste poti, le da bi jih prehodili v obratni smeri).

```
Q := prazna vrsta;
za vsako v ∈ V:
  if v je prosta: d[v] := true; p[v] := NIL; dodaj v v Q;
  else: d[v] := false;
za vsako u ∈ U: d[u] := false;
while Q ni prazna:
  naj bo x prva točka vrste Q; pobriši x iz Q;
  if x ∈ V:
    za vsako povezavo (x, y), ki ni izbrana v M:
      if not d[y]: d[y] := true; p[y] := x; dodaj y v Q;
  else: (* x ∈ U *)
    za vsako povezavo (y, x), ki je izbrana v M:
      if not d[y]: d[y] := true; p[y] := x; dodaj y v Q;
```

V tabeli d označujemo točke, ki smo jih že dosegli z neko dopolnilno potjo, $p[x]$ pa je neposredna prehodnica točke x na tej poti. Ko se ta algoritem konča, moramo preveriti, ali obstaja kakšna prosta točka $u \in U$, za katero je $d[u] =$ **true**. Če take točke ni, potem vemo, da tudi dopolnilne poti ni, drugače pa lahko sestavimo takšno pot korak za korakom s pomočjo tabele p. Pokazati je mogoče, da bo ta algoritem gotovo našel neko dopolnilno pot, če kakšna dopolnilna pot sploh obstaja. Slabost pri tem algoritmu je, da nam vrne le eno dopolnilno pot; ko bomo z njo povečali množico M, ga bomo morali pognati znova in tako naprej. Obstajajo tudi učinkovitejši algoritmi, na primer Hopcroft-Karpov algoritem, ki najde po več dopolnilnih poti naenkrat.

28. Okvarjena ura

Na naši uri so štiri mesta za števke (dve za ure in dve za minute); oštevilčimo jih od

1 do 4. Vsako mesto je sestavljeno iz sedmih diod, vsaka dioda pa je bodisi okvarjena bodisi deluje pravilno. Okvarjenost mesta lahko torej predstavimo s sedmerico bitov iz $\{0,1\}^7$, v kateri je za vsako diodo po en bit, ki pove, ali je tista dioda okvarjena ali ne. Sedmerico, ki opisuje okvarjenost diod na i-tem mestu, označimo z x_i.

Podobno lahko opišemo tudi opažanja dejanskega stanja ure, ki jih dobimo kot vhodne podatke. Naj bo $s_{ji} \in \{0,1\}^7$ stanje i-tega mesta pri j-tem opažanju; celotno j-to opažanje pa naj bo $s_j = (s_{j1}, \ldots, s_{j4})$. Pri tem gre j lahko od 1 do m, saj naloga pravi, da imamo m opažanj.

Mimogrede, naloga sicer pravi, da je lahko dioda okvarjena na dva načina (tako, da je ves čas prižgana, ali pa tako, da je ves čas ugasnjena), vendar nam v x_i ni treba razlikovati med tema dvema možnostma. Če smo diodo vedno (v vseh naših opažanjih) videli prižgano, potem gotovo ni pokvarjena na tak način, da bi bila ves čas ugasnjena; in podobno, če smo jo vedno videli ugasnjeno, potem gotovo ni pokvarjena na tak način, da bi bila ves čas prižgana. (Če pa smo jo v nekaterih opažanjih videli prižgano, v drugih pa ugasnjeno, potem lahko že takoj zaključimo, da sploh ni okvarjena.) Vedno je torej pri posamezni diodi možna največ ena od obeh vrst okvare; to, katera vrsta okvare je pri posamezni diodi možna, lahko ugotovimo na začetku s pregledom seznama opažanj.

Zdaj, ko vemo, kako je posamezna dioda okvarjena (če je okvarjena), lahko definiramo funkcijo funkcijo $f_i(x,d)$, ki naj pove, kakšno bi bilo stanje i-tega mesta na uri, če bi bile na njem okvarjene diode $x \in \{0,1\}^7$ in bi ura poskušala tam prikazati številko $d \in \{0,\ldots,9\}$. Rezultat funkcije $f_i(x,d)$ je sedmerica bitov iz $\{0,1\}^7$, ki opisujejo stanje mesta na enak način kot v vhodnih podatkih (s_{ji}).

Z njeno pomočjo lahko za vsako opažanje $j \in \{1,\ldots,m\}$, vsako mesto $i \in \{1,\ldots,4\}$ in vsako kombinacijo okvarjenih številk $x_i \in \{0,1\}^7$ ugotovimo, katere številke bi pri tej okvari privedle do stanja, kot smo ga na uri dejansko opazili, torej s_{ji}. Množici teh številk bomo rekli $D[i,j,x_i]$. Če je pri kakšnem j ta množica prazna, lahko takoj zaključimo, da nabor okvar x_i za mesto i ni mogoč. Na koncu si torej lahko pripravimo množico X_i vseh možnih naborov okvar na mestu i:

for $i := 1$ **to** 4:
  for $j := 1$ **to** m **do for each** $x_i \in \{0,1\}^7$:
    $D[i,j,x_i] := \{d \in \{0,\ldots,9\} : f_i(d,x_i) = s_j\}$;
  $X_i := \{x_i \in \{0,1\}^7 : D[i,j,x_i]$ so neprazne za vse $j \in 1..m\}$;

Če je kakšna od množic X_i prazna, lahko takoj zaključimo, da so vhodni podatki popolnoma neveljavni (ni nobenega nabora okvarjenih diod, ki bi lahko v resnici pripeljal do takšnega zaporedja opažanj).

Zdaj lahko začnemo sistematično pregledovati vse tiste kombinacije okvar $x = (x_1, x_2, x_3, x_4)$, ki so skladne z doslej dobljenimi omejitvami (torej ki imajo $x_i \in X_i$ za vse i). Za vsako kombinacijo okvar in za vsako opažanje j se lahko vprašamo: kateri je najzgodnejši čas t, ob katerem bi se dalo (pri kombinaciji okvar x) priti do opažanja s_j (in pred njim še v pravem vrstnem redu do opažanj $s_1, \ldots, s_{j-1}$)? Čas je seveda tudi opisan s štirimi številkami, $t = t_1t_2 : t_3t_4$. Pri opažanju s_1 dobimo najzgodnejši čas preprosto tako, da za vsako številko t_i vzamemo najmanjšo možno vrednost, torej minimum množice $D[i,1,x_i]$. Za kasnejša opažanja pa lahko čas t na vsakem koraku povečamo takole: najprej premaknemo t_4 (torej številko z najmanjšo težo — to so enice pri minutah) na prvo večjo vrednost (večjo od dosedanje), ki leži

v $D[i, j, x_4]$; če pa take ni, postavimo t_4 na najmanjšo možno vrednost (iz $D[i, j, x_4]$) in nato poskušamo z enakim razmislekom povečati t_3; in tako naprej. Če pridemo do t_1, ne da bi uspeli čas povečati, to pomeni, da opažanja s_j ne moremo doseči v prvem dnevu opazovanja; naloga pa pravi, da so vsa opažanja nastala znotraj enega dneva, torej lahko v tem primeru zaključimo, da trenutna kombinacija okvar ni mogoča.

for $x_1 \in X_1, x_2 \in X_2, x_3 \in X_3, x_4 \in X_4$:
  for $i := 1$ **to** 4 **do** $t_i := \min D[i, 1, x_i]$;
  $j := 2$; **while** $j \leq m$:
    (* *postavimo t na prvi naslednji trenutek (kasnejši od $t = t_1t_2 : t_3t_4$), ki je konsistenten z opažanjem s_j (pri okvarah $x = x_1x_2x_3x_4$)* *)
    $i := 4$; **while** $i \geq 1$:
      if $t_i \geq \max D[i, j, x_i]$:
        $t_i := \min D[i, j, x_i]$; $i := i - 1$;
      else:
        $t_i :=$ najmanjši element $D[i, j, x_i]$, ki je večji od dosedanje t_i;
        break;
    if $i < 1$ **then break else** $j := j + 1$;
  if $j > m$ **then** $x_1x_2x_3x_4$ je eno od možnih stanj okvarjenosti ure;

29. Stolp iz kock

Pri gradnji stolpov se lahko omejimo na stolpe, v katerih je vsak kvader orientiran tako, da je njegova širina v smeri x vsaj tolikšna kot v smeri y. O tem se lahko prepričamo takole. Oštevilčimo kvadre v stolpu od zgoraj navzdol in naj bo $x_i \times y_i$ velikost i-tega kvadra. Mislimo si najvišji možni stolp, če pa je možnih več enako visokih, vzemimo med njimi takega, ki ima na dnu največje možno število kvadrov v taki orientaciji, da je $x_i \geq y_i$. Če so v našem stolpu celo vsi kvadri taki, je trditev dokazana. Recimo zdaj, da vendarle niso vsi taki, in naj bo i prvi tak kvader (gledano od spodaj navzgor), ki ima $x_i < y_i$. Pod njim je torej kvader $i + 1$, ki ima $x_{i+1} \geq y_{i+1}$. Ker je stolp veljaven, vemo, da je $x_i < x_{i+1}$ in $y_i < y_{i+1}$. Potemtakem pa je tudi $x_i < y_i < y_{i+1}$ in $y_i < y_{i+1} \leq x_{i+1}$; torej lahko kvader i (skupaj z vsemi nad njim) zasukamo za 90 stopinj okoli navpične osi (tako da postane bivša y_i vzporedna z x_{i+1}, bivša stranica x_i pa vzporedna z y_{i+1}), pa stolp ostane veljaven; ima pa zdaj en kvader več v želeni orientaciji kot prej, kar pa je v protislovju s predpostavko, da smo imeli že prej stolp z maksimlnim številom kvadrov (na dnu stolpa) v želeni orientaciji. Tako je trditev dokazana.

Razmislimo zdaj o tem, kako bi sestavili čim višji stolp. Recimo, da gledamo kvader z dimenzijami $a \times b \times c$. V skladu z gornjo omejitvijo obstajajo zdaj največ tri možnosti, kako ga obrniti: izberemo si lahko, katera od stranic a, b in c bo postala višina (torej v smeri z); ostali dve pa moramo potem obrniti tako, da pride daljša od njiju v smeri x, krajša pa v smeri y. Pripravimo si graf, v katerem je po ena točka za vsako od teh treh možnih orientacij kvadra. Posamezno točko bomo označili kar s trojico (x, y, z), ki nam pove dimenzije kvadra v izbrani orientaciji. Iz kvadra $a \times b \times c$ torej nastanejo točke $(\max\{a, b\}, \min\{a, b\}, c)$, $(\max\{a, c\}, \min\{a, c\}, b)$ in $(\max\{b, c\}, \min\{b, c\}, a)$.

V graf dodajmo še usmerjene povezave: od (x, y, z) do (x', y', z') naj obstaja povezava natanko v primeru, če je $x < x'$ in $y < y'$; dolžina te povezave pa naj bo z'. Poleg tega dodajmo še eno posebno točko, recimo ji s, s koordinatami $(0, 0, 0)$; od nje naj bo zato speljana povezava do vsake izmed preostalih točk (x, y, z), dolžina te povezave pa naj bo z.

Vidimo lahko, da če začnemo v točki s in se poljubno sprehajamo po grafu (pri tem pa seveda spoštujemo smer povezav), nam tako dobljena pot predstavlja nek veljaven stolp, torej tak, v katerem je vsak naslednji kvader večji od prejšnjega in bi zato res lahko ležal pod prejšnjim. Dolžina te poti (torej vsota dolžin posameznih povezav na poti) pa je ravno višina tega stolpa.

Razmisliti moramo še o naslednjem: naloga pravi, da imamo dva kompleta n enakih kvadrov; mi pa smo v našem grafu za vsakega od teh n kvadrov naredili tri točke. Ali bi se lahko zgodilo, da bi se na neki poti v grafu pojavile vse tri točke posameznega kvadra? Taka pot bi bila za nas neugodna, saj bi predstavljala stolp, ki ga v resnici ne moremo sestaviti (ker bi zahtevala tri izvode nekega kvadra, mi pa imamo od vsakega kvadra le po dva izvoda). Na srečo se izkaže, da se to ne more zgoditi. Recimo, da imamo kvader $a \times b \times c$; brez izgube za splošnost predpostavimo, da je $a \geq b \geq c$. Naš kvader nam torej v grafu dá točke (a, b, c), (b, c, a) in (a, c, b). Če bi hotela neka pot uporabiti vse tri točke, bi torej morala obiskati tako (a, b, c) kot (a, c, b), kar pa je nemogoče, kajti tidve točki imata enako x-koordinato, povezave pa so v našem grafu speljane tako, da se x-koordinata na vsakem koraku poveča (ne pa ostane enaka). Tako torej vidimo, da nobena pot po grafu ne bo uporabila istega kvadra več kot dvakrat, tako da vsaka pot opisuje stolp, ki ga z dvema kompletoma res lahko sestavimo.

Najvišji stolp lahko torej poiščemo tako, da poiščemo najdaljšo pot v našem grafu. Ker je graf acikličen (to sledi iz dejstva, da se na vsaki povezavi x-koordinata kvadra strogo poveča), lahko najdaljšo pot poiščemo s topološkim urejanjem grafa. Če označimo s p_u najdaljšo dolžino poti z začetkom v točki u, imamo zvezo $p_u = \max_v\{d_{uv} + p_v\}$, pri čemer gre maksimum po vseh takih točkah v, za katere obstaja povezava $u \to v$ (dolžina te povezave pa je d_{uv}). Očitno je koristno najprej obdelati točke, ki nimajo izhodnih povezav (te imajo $p_u = 0$); nato pa lahko na vsakem koraku obdelamo tiste točke u, pri katerih vse izhodne povezave kažejo na take točke v, za katere p_v že poznamo. V spodnjem postopku smo množico točk našega grafa označili z V, koordinate točke $u \in V$ pa z (x_u, y_u, z_u). V d_u hranimo število povezav, ki kažejo iz u na take točke v, za katere še ne poznamo prave vrednosti p_v.

```
za vsako u ∈ V:
  d_u := 0; p_u := 0;
  za vsako v ∈ V:
    if x_u < x_v and y_u < y_v then d_u := d_u + 1;
Q := prazna vrsta;
za vsako u ∈ V:
  if d_u = 0 then dodaj u v vrsto Q;
while Q ni prazna:
  naj bo u poljubna točka iz Q; pobriši u iz Q;
  za vsako v ∈ V:    (* preglejmo naslednice *)
```

if $x_u < x_v$ **and** $y_u < y_v$ **then** $p_u := \max\{p_u, p_v + z_v\}$;
za vsako $v \in V$: (* *preglejmo predhodnice* *)
if $x_v < x_u$ **and** $y_v < y_u$:
$d_v := d_v - 1$;
if $d_v = 0$ **then** dodaj v v vrsto Q;

Na koncu tega postopka imamo v d_s dolžino najdaljše poti z začetkom pri s, to pa je tudi višina najvišjega možnega stolpa.

30. Ogled dirke

Naj bo w dolžina opazovanega intervala (v našem primeru je to ena minuta), T pa čas, ob katerem se ta interval konča. Recimo, da bi počasi povečevali T (in tako premikali naš interval naprej po času) in opazovali, kako se pri tem spreminja število srečnih gledalcev. Če se ob času T kakšen od avtomobilov pripelje mimo nekega gledalca, se število srečnih gledalcev poveča za 1; ob času $T + w$ pa tisti gledalec pade iz intervala in se število srečnih gledalcev (v intervalu) zmanjša za 1.

Lahko si torej pripravimo sezname časov, ob katerih se število srečnih gledalcev spremeni. Časi, ob katerih se avtomobil a (za $a \in \{1, 2\}$) zapelje mimo posameznih gledalcev (in se zaradi tega število srečnih gledalcev poveča za 1), so po vrsti $x_1/v_a, x_2/v_a, \ldots, x_n/v_a$, nato pa (ko avtomobil vozi drugi krog) $(x_1 + d)/v_a, (x_2 + d)/v_a, \ldots, (x_n + d)/v_a$, nato $(x_1 + 2d)/v_a$ in tako naprej. Če tem časom prištejemo w, pa dobimo seznam časov, ob katerih se število srečnih gledalcev zmanjša za 1. Tako imamo za vsak avtomobil dva seznama, skupaj torej štiri sezname. Te štiri sezname lahko zdaj zlivamo in sproti opazujemo, kako se spreminja število srečnih gledalcev, če počasi povečujemo T. Največje število srečnih gledalcev si zapomnimo, skupaj z njim pa tudi vrednost T-ja, ob kateri je bilo doseženo; to je rezultat, po katerem sprašuje naloga.

Vsak od teh štirih seznamov ima $k \cdot n$ elementov, vendar pa jih ni treba predstaviti eksplicitno; dovolj je le, če si zapomnimo, pri katerem krogu in katerem gledalcu se trenutno nahajamo.

```
typedef struct Seznam_
{
  int krog, g;    /* številka kroga, številka gledalca */
  int ds;         /* sprememba števila srečnih gledalcev (+1 ali −1) */
  double t;       /* čas trenutnega dogodka v seznamu */
  double v;       /* hitrost */
  double dt;      /* zamik glede na čas, ko je avtomobil srečal tega gledalca */
} Seznam;

double OgledDirke(int k, double w, double v1, double v2, double d, int n, double xi[])
{
  Seznam s[4]; int a, i, j, stSrecnih = 0, najSrecnih = 0; double najT = −1;
  if (v1 * w >= d || v2 * w >= d) return w;
  /* Inicializirajmo vse štiri sezname. */
  for (j = 0, i = 0; j < 2; j++) for (a = 0; a < 2; a++, i++) {
    s[i].v = (a == 0) ? v1 : v2;
    s[i].dt = (j == 0) ? w : 0; s[i].ds = (j == 0) ? −1 : 1;
    s[i].krog = 0; s[i].g = 0; s[i].t = xi[s[i].g] / s[i].v + s[i].dt; }
```

```
  /* Preglejmo vse možne položaje intervala, pri katerih pride do sprememb. */
  while (true) {
    /* Poiščimo prvi naslednji čas, ob katerem se število gledalcev spremeni. */
    for (j = 0, i = −1; j < 4; j++) {
      if (s[j].krog >= k) continue; /* smo že na koncu tega seznama */
      if (i < 0 || s[j].t < s[i].t) i = j; }
    if (i < 0) break;
    /* Popravimo število srečnih gledalcev pri novem položaju intervala. */
    stSrecnih += s[i].ds;
    /* Če je to najboljša rešitev doslej, si jo zapomnimo. */
    if (stSrecnih > najSrecnih) najSrecnih = stSrecnih, najT = s[i].t;
    /* Premaknimo se naprej po seznamu. */
    if (++s[i].g == n) s[i].krog++, s[i].g = 0;
    s[i].t = (xi[s[i].g] + s[i].krog * d) / s[i].v + s[i].dt; }
  return najT; /* vrnemo čas, ob katerem se konča najboljši interval */
}
```

Če v istem trenutku nastopijo spremembe v več seznamih, je koristno najprej obdelati tiste, ki zmanjšujejo število srečnih gledalcev, šele nato pa tiste, ki ga povečujejo. S tem zagotovimo, da ne bomo nekje vmes (ko so obdelane šele nekatere spremembe v tem trenutku, ne pa vse) dobili nerealistično visokega števila gledalcev, ki bi nam lahko pokvarilo rezultat (vplivalo na končno vrednost najT). V zgornji rešitvi smo za to poskrbeli tako, da med več istočasnimi dogodki najprej obdelamo tistega iz seznama z najmanjšim indeksom, seznami pa so oštevilčeni tako, da najprej pridejo tisti, ki zmanjšujejo število gledalcev v intervalu.

Gornja rešitev posebej obravnava primer, ko je kakšen od avtomobilov tako hiter, da za cel krog porabi w ali manj časa; v tem primeru bi naš postopek z zlivanjem narobe preštel potnike v intervalu, saj bi istega potnika štel po večkrat (ker ga npr. avtomobil sreča že drugič, še preden mine w časa od prvega srečanja). (Res pa je, da bi bil rezultat, ki bi ga funkcija vrnila, še vseeno pravilen, saj funkcija vrača le čas intervala, ne pa tudi števila potnikov v njem.)

31. Paradižniki

Naloge se lahko lotimo z dinamičnim programiranjem. Zastavimo si na primer podprobleme oblike „koliko paradižnikov lahko naberemo, če imamo na voljo le g energije (namesto h) in le prvih k rastlin (namesto vseh n)?“ Odgovoru na to vprašanje recimo $f(k, g)$; rešitev, po kateri sprašuje naloga, je potem seveda $f(n, h)$. Te podprobleme lahko rešujemo z rekurzivnim razmislekom: če imamo pred seboj podproblem $f(k, g)$, se moramo najprej odločiti, koliko paradižnikov (recimo t) bomo pobrali na k-ti rastlini, potem pa nam ostane problem s $k-1$ paradižniki in $g - x_{kt}$ energije (kajti x_{kt} energije smo porabili, da smo na k-ti rastlini pobrali t paradižnikov). Zdaj moramo le preizkusiti vse možne t, da bomo videli, kateri pripelje do najboljše rešitve. Tako smo dobili:

$$f(k, g) = \max\{t + f(k-1, g - x_{kt}) \ : \ 0 \le t \le a_t,\ x_{kt} \le g\}.$$

Funkcijo f bi lahko računali sistematično po naraščajočem številu rastlin k (od 1 do n) in pri vsakem k po vseh možnih g (od 0 do h). Že izračunane vrednosti si

shranjujmo v tabeli, da nam bodo pri roki, ko jih bomo potrebovali za reševanje večjih podproblemov. Tako dobimo naslednji postopek:

```
for g := 0 to h do f[0, g] := 0;

for k := 1 to n:
  for g := 0 to h:
    r := f[k − 1, g];
    for t := 1 to a_k:
      if x_kt > g then break;
      r := max{r, t + f[k − 1, g − x_kt]};
    f[k, g] := r;
return f[n, h];
```

Opazimo lahko še, da ko končamo z izračunom vseh $f[k, g]$ za nek konkretni k, lahko vrednosti $f[k-1, g]$ (za vse g) takoj pozabimo, saj jih v prihodnje (pri $k+1$, $k+2$ in tako naprej) ne bomo več potrebovali. Prostorska zahtevnost tega postopka je torej $O(h)$, časovna pa $O(nha)$, če je $a = \max_i a_i$ največje število sadežev na posamezni rastlini.

Ta rešitev je torej lahko precej neučinkovita, če so višine, ki se pojavljajo v vhodnih podatkih, zelo velike (torej če je h velik). Prav to pa se pri naši nalogi lahko zgodi, saj besedilo naloge pravi, da so rastline zelo visoke. Do boljše rešitve pridemo, če si zastavimo malo drugačne podprobleme: naj bo zdaj $d(k, u)$ najmanjša količina energije, ki jo potrebujemo, da narabutamo u paradižnikov, če uporabljamo le prvih k rastlin. Razmišljamo lahko podobno kot prej: če s k-te rastline poberemo t paradižnikov, bomo za to porabili x_{kt} energije, nato pa bomo morali z ostalih $k-1$ rastlin pobrati še $u - t$ paradižnikov. Podobno kot zgoraj bomo morali preizkusiti vse možne t in uporabiti tistega, ki nam da najboljšo rešitev:

$$d(k, u) = \min\{x_{kt} + d(k-1, u-t) \ : \ 0 \le t \le a_t,\ t \le u\}.$$

(Pri $t = 0$ si moramo predstavljati, da je $x_{k0} = 0$; tako pokrijemo možnost, da na k-ti rastlini ne poberemo nobenega paradižnika.)

Podobno kot prej $f(k, g)$ lahko tudi $d(k, u)$ računamo sistematično po naraščajočem k in pri vsakem k po vseh možnih u (od 0 do skupnega števila vseh paradižnikov, recimo mu $s := a_1 + \ldots + a_n$). Ko računamo $d(k, u)$ za nek konkretni k, potrebujemo le rezultate $d(k-1, u')$ za $u' \le u$. Zato je koristno računati (pri posameznem k) vrednosti $d(k, u)$ po padajočih u namesto po naraščajočih u; tedaj za shranjevanje vrednosti funkcije d sploh ne potrebujemo dvodimenzionalne tabele, ampak le enodimenzionalno: $d[u']$ naj hrani za $u' < u$ vrednosti $f(k-1, u')$, za $u' > u$ pa vrednosti $f(k, u')$. Naš postopek je torej:

```
d[0] := 0; for u := 1 to s do d[u] := ∞;

for k := 1 to n:
  for u := s downto 0:
    r := d[u];
    for t := 1 to a_k:
      if t > u then break;
      r := min{r, x_kt + d[u − t]};
    d[u] := r;
```

Spomnimo se, da naloga sprašuje po največjem številu paradižnikov, ki jih lahko oberemo s h enotami energije. Da dobimo ta rezultat, moramo le poiskati največji tak u, pri katerem je $d[u] \leq h$.

Prostorska zahtevnost tega postopka je $O(s)$, časovna pa $O(nsa)$. Zdaj torej nismo več odvisni od tega, kakšne višine se pojavljajo v nalogi, ampak le od skupnega števila paradižnikov.

32. Binomski koeficienti

Vrednost $\binom{n}{k} \bmod m$ bi seveda lahko izračunali tako, da bi najprej izračunali $\binom{n}{k}$ in nato njegov ostanek po deljenju z m; za izračun $\binom{n}{k}$ pa bi lahko uporabili rekurzivno zvezo $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ ali pa eksplicitno formulo $\binom{n}{k} = n!/(k!(n-k)!) = n(n-1)\ldots(n-k+1)/k!$. Težava takšne rešitve je, da je za velike n in k, s kakršnimi imamo opravka pri tej nalogi, prepočasna. Rekurzivna zveza bi zahtevala $O(n^2)$ seštevanj, eksplicitna formula pa $O(n)$ množenj in eno deljenje, za povrh vsega pa bi imeli pri tem opravka z zelo velikimi celimi števili, tako da je že vsaka posamezna aritmetična operacija na njih precej zamudna.

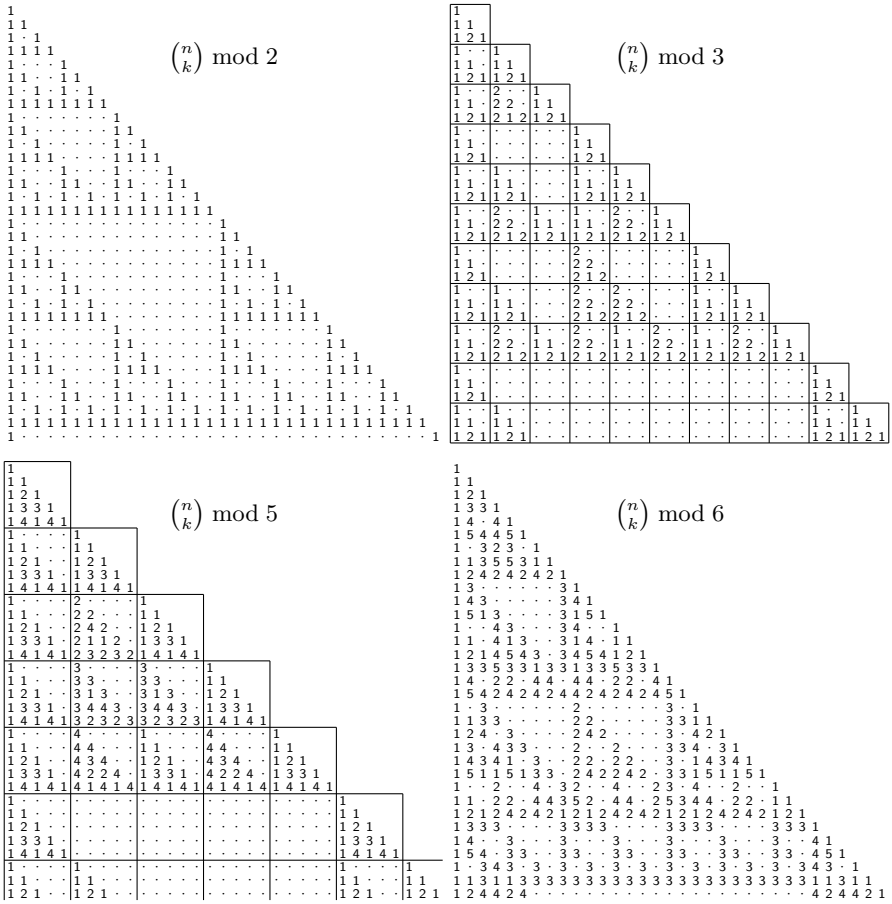
Malo učinkovitejši postopek bi bil, da bi v ulomku $\binom{n}{k} = n(n-1)\ldots(n-k+1)/k!$ predstavili števec in imenovalec vsakega posebej kot produkt prafaktorjev; to bi nam močno olajšalo izračun vrednosti $\binom{n}{k} \bmod m$ in ne bi več potrebovali velikih celih števil. Vendar pa je različnih praštevil do n tudi kar precej, približno $n/\ln n$, in pri tako velikih n, s kakršnimi imamo opravka pri tej nalogi, si ne moremo privoščiti, da bi pregledali vsa ta praštevila; tudi za iskanje praštevil z Eratostenovim rešetom bi nam zmanjkalo pomnilnika.

Pomagajmo si torej z namigoma iz besedila naloge. Oglejmo si za začetek prvih nekaj vrstic Pascalovega trikotnika po modulu m, torej vrednosti $\binom{n}{k} \bmod m$ za majhne n (glej sliko na str. 147). Pri $m = 6$, ki nas načeloma še najbolj zanima, bomo opazili, da se v trikotniku sicer nakazuje nekakšen red, vendar ni tako predvidljiv, da bi znali kar opaziti kakšno učinkovito formulo za računanje $\binom{n}{k} \bmod 6$. Precej lepše in bolj predvidljive vzorce pa dobimo pri $m = 2$, $m = 3$ in (kot se izkaže) sploh pri vseh praštevilskih m. Pri $m = 2$ tvorijo ostanki $\binom{n}{k} \bmod 2$ dobro znani fraktal — trikotnik Sierpińskega (če si predstavljamo enice kot črna polja in ničle kot bela polja). Še lažje pa bomo splošno formulo za $\binom{n}{k} \bmod m$ opazili pri $m = 3$ in $m = 5$. Pri $m = 3$ na primer vidimo, da se kvadrat, ki ga tvorijo prve tri vrstice, torej

```
1
1 1
1 2 1
```

v nadaljevanju trikotnika precej ponavlja. Naslednje tri vrstice sestavljata dve kopiji tega osnovnega kvadrata. Nato pridejo tri vrstice, kjer so tri kopije osnovnega kvadrata, vendar je druga pomnožena z 2. Nato pridejo tri vrstice, kjer so štiri kopije osnovnega kvadrata, vendar sta druga in tretja pomnoženi z 0. Če tako nadaljujemo in si zapisujemo faktorje, s katerimi smo pomnožili posamezne kopije osnovnega kvadrata, dobimo:

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



Ilustracija k rešitvi naloge z binomskimi koeficienti. Prikazanih je prvih nekaj vrstic Pascalovega trikotnika po modulu m za $m = 2, 3, 5, 6$. Namesto ničel so zaradi večje preglednosti prikazane pike.

---

in tako naprej — to pa je popolnoma enak trikotnik kot tisti, s katerim smo začeli. Pri $m = 5$ lahko opazimo enako obnašanje, le da je osnovni kvadrat zdaj velik 5×5 števil namesto le 3×3. Dobljeno ugotovitev lahko zapišemo takole:

$$\binom{am+n}{bm+k} \equiv \binom{a}{b}\binom{n}{k} \pmod{m}.$$

To velja za vsak praštevilski m in za vse $a \geq 0$, $b \geq 0$, $0 \leq k < m$, $0 \leq n < m$.

Prepričajmo se, da je to res. Za začetek je koristno razmisliti, kaj se dogaja pri $n = m$ — to je vrstica, ki pride tik pod prvim osnovnim kvadratom. Pri $k = 0$ in $k = m$ imamo $\binom{n}{k} = 1$, zato je tudi ostanek po modulu m enak 1. Pri $0 < k < m$ pa nam primeri na sliki kažejo, da je takrat $\binom{m}{k} \bmod m = 0$. Kako bi to dokazali?

Spomnimo se, da je $\binom{m}{k} = m(m-1)\dots(m-k+1)/k!$. Ker je m praštevilo in ker je $k < m$, vemo, da so števila $1, \dots, k$ vsa tuja m-ju, torej je tudi $k!$ tuja m-ju; torej tistega m-ja, ki se pojavlja v številcu našega ulomka $m(m-1)\dots(m-k+1)/k!$, imenovalec ne more nikakor okrajšati; torej je tudi celoten ulomek večkratnik m-ja. Tako torej vidimo, da je $\binom{m}{k}$ večkratnik m-ja, tako da je $\binom{m}{k} \bmod m$ res enako 0.

Od tu naprej lahko dokazujemo z indukcijo, pri čemer se opiramo na dejstvo, da je vrednost binomskega koeficienta odvisna le od dveh neposredno nad njim: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ (enaka zveza velja tudi, če gledamo le njihove ostanke po modulu m, le da moramo tudi po seštevanju vzeti ostanek po modulu m). Ker imamo torej v vrstici $n = m$ dve enici in vmes same ničle, nastane iz vsake enice nova kopija Pascalovega trikotnika (po modulu m). Ti dve kopiji druga na drugo nič ne vplivata, dokler ju ločujejo ničle; pri $n = 2m$ pa se začneta prekrivati. Vsako vrstico trikotnika od $n = m$ naprej lahko torej dobimo tako, da vzamemo dve kopiji vrstice $n - m$, eno zamaknemo za m mest v desno in ju nato seštejemo. Pri $n = 2m$ zato nastane vrstica oblike

$$\begin{array}{cccccccccccccc} & 1 & 0 & 0 & \cdots & 0 & 1 & & & & & & & \\ + & & & & & & 1 & 0 & 0 & \cdots & 0 & 1 \\ \hline = & 1 & 0 & 0 & \cdots & 0 & 2 & 0 & 0 & \cdots & 0 & 1. \end{array}$$

Iz te pa zato pri $n = 3m$ nastane vrstica

$$\begin{array}{cccccccccccccccccc} & 1 & 0 & 0 & \cdots & 0 & 2 & 0 & 0 & \cdots & 0 & 1 & & & & & & \\ + & & & & & & 1 & 0 & 0 & \cdots & 0 & 2 & 0 & 0 & \cdots & 0 & 1 \\ \hline = & 1 & 0 & 0 & \cdots & 0 & 3 & 0 & 0 & \cdots & 0 & 3 & 0 & 0 & \cdots & 0 & 1 \end{array}$$

in tako naprej. Tako torej vidimo, da je $\binom{am}{bm} \equiv \binom{a}{b} \pmod m$, drugod v vrstici $n = am$ pa so same ničle. Iz tega pa tudi sledi, da se kvadrat $m \times m$ z zgornjim levim kotom v $\binom{am}{bm}$ obnaša enako kot osnovni kvadrat (tisti iz vrstic $0 \le n < m$), pomnožen z $\binom{a}{b}$. □

Formulo, ki smo jo pravkar dokazali, lahko zapišemo tudi takole: če imamo n in k izražena v m-iškem zapisu kot $n = \sum_{i=0}^{d} n_i m^i$ in $k = \sum_{i=0}^{d} k_i m^i$, potem je

$$\binom{n}{k} \equiv \binom{n_0}{k_0}\binom{n_1}{k_1}\cdots\binom{n_d}{k_d} \pmod m.$$

Kakorkoli že, na ta način bomo lahko učinkovito računali $\binom{n}{k} \bmod m$ tudi za velike n in k.

Doslej smo se omejili na primere, ko je m praštevilo; naloga pa nas pravzaprav sprašuje o $m = 6$, ki ni praštevilo, je pa produkt dveh praštevil, 2 in 3. Tu nam pride prav drugi namig iz besedila naloge: ostanek $x \bmod 6$ lahko izračunamo iz $x \bmod 2$ in $x \bmod 3$, četudi samega x sploh ne poznamo. Da se o tem prepričamo, si oglejmo te ostanke za nekaj majhnih x:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	$\cdots$
$x \bmod 2$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	$\cdots$
$x \bmod 3$	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	$\cdots$
$x \bmod 6$	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	$\cdots$

Ostanki po modulu m se seveda ponavljajo s periodo m; ker je 6 večkratnik števil 2 in 3, se ostanki po deljenju z 2 in 3 ponavljajo tudi s periodo 6; zato, če se več x-ov ujema v $x \bmod 6$, se ujemajo tudi v $x \bmod 2$ in $x \bmod 3$. Poleg tega iz tabele tudi vidimo, da se nikoli ne zgodi, da bi se dva x ujemala v $x \bmod 2$ in $x \bmod 3$, ne pa v $x \bmod 6$. Z malo razmisleka lahko zapišemo zvezo med temi ostanki celo s formulo:

$$x \bmod 6 = (3 \cdot (x \bmod 2) + 4 \cdot (x \bmod 3)) \bmod 6.$$

Tako torej vidimo, da lahko po prej dobljeni formuli poceni izračunamo $\binom{n}{k} \bmod 2$ in $\binom{n}{k} \bmod 3$, iz teh dveh pa znamo zdaj izračunati tudi $\binom{n}{k} \bmod 6$.[24]

Zdaj vemo dovolj, da lahko zapišemo našo rešitev v C-ju:

```
typedef unsigned long long int_t;

/* Izračuna binomski koeficient po definiciji; primerna za majhne n in k. */
int binom(int n, int k)
{
  int i, stevec = 1, imenovalec = 1;
  for (i = 0; i < k; i++) stevec *= n - i, imenovalec *= i + 1;
  return stevec / imenovalec;
}

/* Z Lucasovim izrekom izračuna ostanek po deljenju binomskega koeficienta
   s praštevilom p. */
int binom_mod_p(int_t n, int_t k, int p)
{
  int rezultat = 1, r;
  while (n > 0) {
    /* Vzemimo najnižjo števko n-ja in k-ja v p-iškem zapisu. Ti dve števki,
       n % p in k % p, nam v rezultatu prispevata faktor binom(n % p, k % p). */
    r = binom((int) (n % p), (int) (k % p));
    rezultat = (rezultat * r) % p;
    /* Pobrišimo najnižjo števko v p-iškem zapisu n-ja in k-ja. */
    n /= p; k /= p; }
  return rezultat;
}

/* S kitajskim izrekom o ostankih izračuna ostanek po deljenju
   binomskega koeficienta s 6. */
int binom_mod_6(int_t n, int_t k)
{
  return (3 * binom_mod_p(n, k, 2) + 4 * binom_mod_p(n, k, 3)) % 6;
}
```

[24] Formulo za izračun $\binom{n}{k} \bmod m$ za praštevilske m je odkril že Édouard Lucas leta 1878. Kasneje jo je Andrew Granville posplošil tudi na take m, ki so potence praštevil (A. Granville, *Arithmetic Properties of Binomial Coefficients I: Binomial coefficients modulo prime powers*, Can. Math. Soc. Conf. Proceedings 20:253–275, 1997). Formula, ki smo jo uporabili za izračun $x \bmod 6$ iz $x \bmod 2$ in $x \bmod 3$, pa je poseben primer ugotovitve, ki je v matematiki znana kot „kitajski izrek o ostankih“; gl. npr. Wikipedijo *s. v.* Chinese remainder theorem. S tem izrekom in z Granvillovo formulo bi se dalo precej učinkovito računati $\binom{n}{k} \bmod m$ za poljuben m. Mimogrede, s kitajskim izrekom o ostankih smo se pred leti že srečali: glej rešitev naloge 2008.3.1, str. 61 v biltenu 2008.

Naloge so sestavili: pravokotnik, pismo iz zapora, knjige, zbiranje sličic, paradižniki — Nino Bašić; neskončne stopnice, najdaljši palindrom — Primož Gabrijelčič; KvadMars — Boris Gašperin; primerjava IPjev — Matija Grabnar; dvigalo, T9, stolp iz kock — Tomaž Hočevar; transfuzije, pikavost — Boris Horvat; številski sestavi, flomastri, črni kovčki — Nace Hudobivnik; kemijske spojine, torta, vlaki — Jurij Kodre; branje luknjanega traku, pari besed, trgovina, skakač na šahovnici — Mitja Lasič; puškomitraljez — Polona Novak; spirala, histogram, tihotapec, ogled dirke — Mitja Trampuš; disemvowelling, okvarjena ura, binomski koeficienti — Janez Brank.

NASVETI ZA MENTORJE O IZVEDBI ŠOLSKEGA TEKMOVANJA IN OCENJEVANJU NA NJEM

[Naslednje nasvete in navodila smo poslali mentorjem, ki so na posameznih šolah skrbeli za izvedbo in ocenjevanje šolskega tekmovanja. Njihov glavni namen je bil zagotoviti, da bi tekmovanje potekalo na vseh šolah na približno enak način in da bi ocenjevanje tudi na šolskem tekmovanju potekalo v približno enakem duhu kot na državnem.—*Op. ur.*]

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). Pri reševanju si lahko tekmovalci pomagajo tudi z literaturo in/ali zapiski, ni pa mišljeno, da bi imeli med reševanjem dostop do interneta ali do kakšnih datotek, ki bi si jih pred tekmovanjem pripravili sami. Čas reševanja je omejen na 180 minut.

Nekatere naloge kot odgovor zahtevajo program ali podprogram v kakšnem konkretnem programskem jeziku, nekatere naloge pa so tipa „opiši postopek“. Pri slednjih je načeloma vseeno, v kakšni obliki je postopek opisan (naravni jezik, psevdokoda, diagram poteka, izvorna koda v kakšnem programskem jeziku, ipd.), samo da je ta opis dovolj jasen in podroben in je iz njega razvidno, da tekmovalec razume rešitev problema.

Glede tega, katere programske jezike tekmovalci uporabljajo, naše tekmovanje ne postavlja posebnih omejitev, niti pri nalogah, pri katerih je rešitev v nekaterih jezikih znatno krajša in enostavnejša kot v drugih (npr. uporaba perla ali pythona pri problemih na temo obdelave nizov).

Kjer se v tekmovalčevem odgovoru pojavlja izvorna koda, naj bo pri ocenjevanju poudarek predvsem na vsebinski pravilnosti, ne pa na sintaktični. Pri ocenjevanju na državnem tekmovanju zaradi manjkajočih podpičij in podobnih sintaktičnih napak odbijemo mogoče kvečjemu eno točko od dvajsetih; glavno vprašanje pri izvorni kodi je, ali se v njej skriva pravilen postopek za rešitev problema. Ravno tako ni nič hudega, če npr. tekmovalec v rešitvi v C-ju pozabi na začetku **#include**ati kakšnega od standardnih headerjev, ki bi jih sicer njegov program potreboval; ali pa če podprogram `main()` napiše tako, da vrača **void** namesto **int**.

Pri vsaki nalogi je možno doseči od 0 do 20 točk. Od rešitve pričakujemo predvsem to, da je pravilna (= da predlagani postopek ali podprogram vrača pravilne rezultate), poleg tega pa je zaželeno tudi, da je učinkovita (manj učinkovite rešitve dobijo manj točk).

Če tekmovalec pri neki nalogi ni uspel sestaviti cele rešitve, pač pa je prehodil vsaj del poti do nje in so v njegovem odgovoru razvidne vsaj nekatere od idej, ki jih rešitev tiste naloge potrebuje, naj vendarle dobi delež točk, ki je približno v skladu s tem, kolikšen delež rešitve je našel.

Če v besedilu naloge ni drugače navedeno, lahko tekmovalčeva rešitev vedno predpostavi, da so vhodni podatki, s katerimi dela, podani v takšni obliki in v okviru takšnih omejitev, kot jih zagotavlja naloga. Tekmovalcem torej načeloma ni treba pisati rešitev, ki bi bile odporne na razne napake v vhodnih podatkih.

V nadaljevanju podajamo še nekaj nasvetov za ocenjevanje pri posameznih nalogah.

1. Lemingi

- Naloga pravi, da če leming pri padanju zgolj oplazi krajišče ploščadi, se to ne šteje kot padec na ploščad. V naši rešitvi to na primer pomeni, da v pogoju „**if** $x_i < x$ **and** $x < x_i + d_i$“ nastopata znaka $<$ in ne $\leq$. Rešitvam, ki stik s krajiščem pomotoma štejejo kot padec na ploščad, naj se zaradi tega odbije dve točki.

- Naloga pravi, da se lemingu smer gibanja spremeni, ko prileti na ploščad. Iz tega na primer sledi, da če je ob začetku letenja obrnjen v desno ($s_0 = \texttt{D}$), se bo po prvi ploščadi, na katero bo priletel, gibal v levo (kot vidimo tudi v primeru v besedilu naloge). Rešitvam, ki pomotoma predpostavljajo, da se leming po prvi ploščadi, na katero prileti, giblje v smeri s_0 namesto v nasprotni smeri, naj se zaradi tega odbije dve točki.

- Mišljeno je, da učinkovita rešitev pri tej nalogi porabi $O(n)$ časa, če je n število ploščadi. Rešitev, ki porabi $O(n^2)$ časa (npr. ker po vsakem padcu pregleda celoten seznam ploščadi, da ugotovi, katera je naslednja ploščad, na katero bo leming priletel), naj dobi največ 10 točk (če drugače daje pravilne rezultate).

2. 3-d šah

- V naših rešitvah smo predstavili dve različici rešitve: pri eni imamo tabelo $8 \times 8 \times 8$ celic in za vsako kraljico označimo polja, ki jih ta kraljica napada (in povečamo ustrezne elemente tabele), pri drugi pa nimamo tabele in za vsako polje zgolj z izračunom preverimo, ali ga posamezna kraljica napada ali ne. Čeprav je druga različica elegantnejša in porabi manj pomnilnika, naj se pri ocenjevanju obe vrsti rešitev šteje kot dovolj dobri. Tudi rešitev s tabelo lahko torej dobi vse točke, če daje pravilne rezultate.

- Če bi rešitev pomotoma predpostavila, da so koordinate kraljic v vhodnih podatkih podane v razponu 0..7 namesto 1..8, naj se ji zaradi tega odšteje dve točki.

- Če bi rešitev z označevanjem napadenih polj v tabeli pomotoma kakšno polje označila po večkrat pri isti kraljici (in zato pri kakšnem polju mislila, da ga napada več kraljic, kot jih v resnici), naj se ji zaradi tega odbije največ pet točk.

- Če bi rešitev z označevanjem napadenih polj v tabeli pomotoma poskušala dostopati do elementov na neveljavnih indeksih, naj se ji zaradi tega odšteje največ pet točk.

3. Brisanje parov

- Mišljeno je, da bi učinkovita rešitev pri tej nalogi porabila $O(n)$ časa. Rešitve, ki porabijo $O(n^2)$ časa, naj dobijo največ 12 točk (če so drugače pravilne).

- Nič ni narobe, če rešitev namesto funkcij JeMala in DajVeliko, ki ju omenja besedilo naloge, uporablja ekvivalentne funkcije iz standardne knjižnice svojega jezika (npr. islower in toupper v C/C++).

- Besedilo naloge pravi, da funkcija DajVeliko sproži izjemo, če ji kot parameter podamo znak, ki ni mala črka. Če rešitev ne pazi na to in včasih pokliče DajVeliko s parametrom, ki ni mala črka, naj se ji odbije tri točke.

4. Ta5nik

- Naloga pravi, da mora rešitev izpisati okrajšano različico niza le, če je pri krajšanju izvedla vsaj dve zamenjavi. Rešitvam, ki tega pogoja ne upoštevajo (in npr. okrajšano različico niza izpišejo v vsakem primeru, ali pa je nikoli ne izpišejo in jo zgolj vrnejo kot funkcijsko vrednost), naj se zaradi tega odbije štiri točke.

- Naloga pravi, da če je možno na nekem nizu izvesti več zamenjav, moramo niz okrajšati tako, kot da bi najprej izvedli najbolj levo možno zamenjavo (na primer `petrijevka` → `5rijevka` in ne `pe3jevka`). Če rešitev tega pogoja ne upošteva, naj se ji zaradi tega odbije pet točk.

- Drugače pa rešitev, ki zamenja vseh deset vzorcev v fiksnem vrstnem redu (npr. najprej zamenja vse pojavitve podniza `dve` z `2`, nato zamenja vse pojavitve podniza `ena` z `1` itd.), tudi lahko dobi vse točke, če je vrstni red izbran tako, da daje pravilne rezultate (torej `dve` → `2` pred `ena` → `1` ipd.).

5. Koalicije

- V tej nalogi se skriva problem barvanja točk grafa z dvema barvama (točke grafa predstavljajo stranke, povezave pa pare sovražnih strank), pri čemer točki ne smeta biti iste barve, če sta neposredno povezani s povezavo. Ta naloga je za šolsko tekmovanje precej težka, zato od rešitev pričakujemo predvsem opažanje, da ko določimo barvo ene točke, iz tega enolično izhaja barva njenih sosed, iz tega potem barva njihovih sosed in tako naprej. Ni pa nujno, da rešitev graf pregleduje učinkovito. Naša rešitev porabi $O(n + m)$ časa za graf z n točkami in m povezavami, dovolj dobra pa bi bila tudi rešitev s časovno zahtevnostjo $O(n^2)$ (tudi taka lahko dobi vseh 20 možnih točk, če je tudi sicer pravilna).

- Če rešitev ne upošteva, da je graf lahko sestavljen iz več ločenih komponent, ki med seboj niso povezane, in zato pobarva samo eno od njih, naj se ji odbije šest točk. Če rešitev sicer pobarva vse komponente, vendar rezultatov za posamezne komponente ne skombinira tako, da bi nastala največja možna koalicija za graf kot celoto (npr. ker ne upošteva, da mora pri vsaki komponenti od dveh možnih koalicij vzeti tisto, ki ima več poslancev), naj se ji odbije tri točke.

- Naloga pravi tudi, naj rešitev prepozna primere, ko sploh ne obstaja nobena koalicija, ki bi bila v skladu z omejitvami (do tega pride, ko je v grafu kakšen cikel lihe dolžine). Rešitvi, ki takih primerov ne zazna (in takrat npr. vrne neko neveljavno koalicijo), naj se odšteje štiri točke.

Težavnost nalog

Državno tekmovanje ACM v znanju računalništva poteka v treh težavnostnih skupinah (prva je najlažja, tretja pa najtežja); na tem šolskem tekmovanju pa je skupina ena sama, vendar naloge v njej pokrivajo razmeroma širok razpon zahtevnosti. Za občutek povejmo, s katero skupino državnega tekmovanja so po svoji težavnosti primerljive posamezne naloge letošnjega šolskega tekmovanja:

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM in IJS
1. Lemingi	lažja naloga v prvi skupini
2. 3-d šah	srednje težka naloga v prvi skupini
3. Brisanje parov	težja v prvi ali lahka naloga v drugi skupini
4. Ta5nik	težja v prvi ali lažja naloga v drugi skupini
5. Koalicije	srednje težka 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 prvi skupini izjemoma podelili tri tretje nagrade, v drugi skupini eno prvo, tri druge in tri tretje, v tretji pa eno prvo in tri druge. 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 243) in smo jih letos podelili devet. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot v prejšnjih letih pohvale; prejmejo jih tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil zlatega priznanja; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 20 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen srebrnih priznanj je, da izkažemo priznanje in spodbudo vsem, ki se po rezultatu prebijejo v zgornjo polovico svoje skupine. Podobno prakso poznajo tudi na nekaterih mednarodnih tekmovanjih; na primer, na mednarodni računalniški olimpijadi (IOI) prejme medalje kar polovica vseh udeležencev. Poleg zlatih in srebrnih priznanj obstajajo tudi bronasta, ta pa so dobili najboljši tekmovalci v okviru šolskih tekmovanj.

V tabelah na naslednjih straneh so prejemniki nagrad označeni z „1“, „2“ in „3“ v prvem stolpcu, prejemniki priznanj pa z „Z“ (zlato) in „S“ (srebrno).

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Žiga Željko	OŠ	ZRI	20	19	20	20	20	99
1Z	2	Metod Medja	2	ŠC Kranj, SŠER	20	18	19	20	20	97
2Z	3	Jakob Košir	4	ŠC Novo mesto	17	20	16	20	20	93
2Z	4	Samo Remec	2	Vegova Ljubljana	20	20	12	19	20	91
3S	5	Luka Kolar	2	Gimnazija Vič	20	20	11	19	20	90
3S	6	Jan Likar	3	SŠ V. Pilon Ajdovščina	18	20	12	19	20	89
3S	7	Bor Brecelj	1	ZRI + Gimnazija Vič	18	20	10	20	20	88
S	8	Andrej Muhič	4	ŠC Novo mesto	18	20	9	20	20	87
S		Jakob Erzar	2	Gimnazija Kranj	20	20	8	19	20	87
S	10	Marko Grešak	4	ŠC Novo mesto	17	19	11	18	20	85
S	11	Jaka Kordež	2	ŠC Kranj, SŠER	18	20	9	17	20	84
S	12	Miloš Ljubotina	3	SŠJJ Ivančna Gorica	17	20	9	19	18	83
S	13	Jaka Mohorko	2	II. gimnazija Maribor	14	19	10	19	20	82
S	14	Anže Bertoncelj	3	ŠC Kranj, Str. gim.	18	18	6	19	18	79
S		Domen Vidovič	4	SERŠ Maribor	15	20	10	14	20	79
S	16	Tadej Pečar	3	ŠC Novo mesto	10	18	10	20	20	78
S	17	Žan Pevec	4	ŠC Celje, Gimn. Lava	17	16	7	18	19	77
S		Žan Skamljič	4	SERŠ Maribor	14	18	10	20	15	77
S	19	Žiga Šmelcer	3	Škof. klas. gimn. Lj.	13	18	5	20	20	76
S	20	Luka Pogačnik	2	Gimnazija Vič	18	18	7	17	15	75
S	21	Franko Jančič	4	STŠ Koper	18	16	10	10	20	74
S	22	Dominik Korošec	4	SERŠ	20	2	11	20	20	73
S	23	Vid Leskovar	3	II. gimnazija Maribor	20	0	12	19	19	70
S	24	Urban Lavbič	4	ŠC Celje, SŠ za KER	19	12	8	10	20	69
S		Žiga Kokelj	3	Gimnazija Škofja Loka	15	20	9	12	13	69
S	26	Andraž Jelenc	2	Gimnazija Škofja Loka	9	19	10	9	20	67
S		Janez Štular	4	ŠC Kranj, SŠER	17	17	7	9	17	67
S	28	Gregor Šturm	4	ŠC Kranj, SŠER	18	18	9	0	20	65
S		Klemen Kogovšek	2	Vegova Ljubljana	13	18	8	14	12	65
S	30	Matej Logar	3	Gimnazija Vič	18	12	11	6	17	64
S	31	Miha Černe	1	Gimnazija Vič	17	15	10	1	20	63
S	32	Rok Poje	4	Vegova Ljubljana	0	20	5	17	20	62
S	33	Gašper Romih	3	Gimnazija Vič	13	18	7	3	20	61
S	34	Ivan Kolundžija	3	Gimnazija Vič	19	20	10	3	8	60
S		Martin Oprešnik	4	ŠC Celje, SŠ za KER	17	18	5	0	20	60
S	36	Luka Ernestini	1	ZRI	13	1	10	13	18	55
S		Nejc Kišek	4	Gimnazija Kranj	12	18	5	0	20	55
S		Nejc Savodnik	2	Gimnazija Šentvid	3	15	9	9	19	55

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
	39	Aleš Papič	4	ŠC Celje, SŠ za KER	15	18	10	0	10	53
	40	Žiga Vodušek Resnik	4	ŠC Celje, Gimn. Lava	18	3	7	5	19	52
	41	David Šket	3	ŠC Celje, SŠ za KER	11	15	10	0	15	51
	42	Gašper Jelovčan	2	Škof. klas. gimn. Lj.	2	20	8	0	20	50
	43	Gregor Ažbe	2	ŠC Kranj, SŠER	17	0	9	8	13	47
		Matej Vovko	3	ŠC Novo mesto	15	0	12	0	20	47
		Rok Šeško	1	II. gimnazija Maribor	18	5	11	3	10	47
	46	Domen Balantič	3	ŠC Kranj, Str. gim.	14	4	6	16	5	45
	47	Igor Borenovič	1	ŠC Kranj, SŠER	0	0	7	14	20	41
	48	Matej Tomc	2	Škof. klas. gimn. Lj.	0	10	10	1	19	40
	49	Pavle Iliev	4	ŠC Ptuj, ERŠ	14	7	7	0	9	37
	50	Alen Verk	4	ŠC Celje, SŠ za KER	18	0	5	1	7	31
	51	Andrej Dremelj	3	SŠJJ Ivančna Gorica	0	12	5	0	13	30
		Jan Gašperlin	4	Gimnazija Kranj	11	5	0	0	14	30
	53	Dejan Benedik	4	Gimnazija Kranj	18	0	11	0	0	29
	54	Stane Lokar	2	SŠ V. Pilon Ajdovščina	0	0	10	0	18	28
	55	Peter Jare	4	Gimnazija Šentvid	1	3	8	6	8	26
	56	Lucija Gruden	1	Gimnazija Vič	1	0	11	3	10	25
		Simon Šenk	4	Gimnazija Kranj	0	10	11	0	4	25
	58	Aljaž Kočevar	2	ŠC Velenje, ERŠ	3	0	5	7	8	23
		Jože Fartek	2	SPTŠ Murska Sobota	19	0	4	0	0	23
	60	Dominik Baligač	4	SPTŠ Murska Sobota	9	0	7	0	6	22
	61	Alen Vuk	4	ŠC Ptuj, ERŠ	0	0	6	0	15	21
		Gašper Gračner	4	ŠC Celje, SŠ za KER	3	0	6	0	12	21
	63	Gašper Matos	4	Gimnazija Šentvid	1	9	0	0	10	20
		Žan Pogačnik	1	ŠC Kranj, SŠER	11	0	8	0	1	20
	65	Klemen Gumzej	2	ŠC Celje, SŠ za KER	12	0	3	0	4	19
	66	Tadej Hribar	2	Gimnazija Litija	16	0	0	0	0	16
	67	Matija Šteblaj	2	Gimnazija Vič	0	0	11	0	4	15
	68	Blaž Stojanovič	1	Gimnazija Kranj	0	0	9	0	3	12
		Nejc Kmet	1	ŠC Kranj, SŠER	0	0	4	1	7	12
	70	Damjan Časar	3	SPTŠ Murska Sobota	0	0	3	0	8	11
	71	Luka Murn		Gimn. in SŠ Kočevje	0	0	9	0	0	9
	72	Jani Pezdevšek	3	ŠC Celje, SŠ za KER	0	0	7	0	0	7
	73	Aljaž Šešo	2	ŠC Ptuj, ERŠ	0	0	1	0	3	4
	74	Blaž Rojc	1	Gimnazija Nova Gorica	0	0	0	0	0	0
		Ruben Kurinčič	2	Gimnazija Nova Gorica	0	0	0	0	0	0
		Vid Trtnik	2	Gimnazija Poljane	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	Lojze Žust	3	Škof. klas. gimn. Lj.	20	19	18	19	15	91
2Z	2	Žan Kusterle	4	Vegova Ljubljana	18	18	13	17	7	73
2Z		Blaž Blokar	4	ŠC Nova Gorica	15	8	20	17	13	73
2S	4	Filip Peter Lebar	3	Gimnazija Vič	10	12	20	18	10	70
3S	5	Žiga Simončič	3	Vegova Ljubljana	15	12	15	7	13	62
3S	6	Tobias Mihelčič	3	Vegova Ljubljana	20	11	5	18	7	61
3S	7	Aljaž Jeromel	3	II. gimnazija Maribor	7	12	10	13	15	57
S	8	Rok Kos	1	ZRI + Gimnazija Vič	0	9	17	9	12	47
S	9	Matej Mohar	4	ŠC Nova Gorica	3	20	15	1	5	44
S	10	Tim Weisseisen	3	Vegova Ljubljana	10	7	20	2	3	42
S	11	Marko Lavrinec	4	ŠC Kranj, Str. gim.	3	7	12	9	10	41
S	12	Nejc Smrkolj Koželj	3	Vegova Ljubljana	5	11	5	7	12	40
S		Matej Horvat	3	Gimnazija Moste	0	13	19	3	5	40
S	14	Sebastjan Čolić	3	Gimnazija Moste	5	0	12	7	15	39
S		Jan Makovecki	4	Gimnazija Poljane	12	17	5	0	5	39
S	16	Vid Pavše	3	ŠC Ravne na Koroškem	0	13	10	10	5	38
S	17	Jaka Konda	4	Vegova Ljubljana	12	12	3	5	5	37
S		Matej Reberc	4	ŠC Ptuj, ERŠ	10	9	18	0	0	37
	19	Miha Štravs	2	ZRI	0	15	15	6	0	36
	20	Michel Adamič	4	Gimnazija Bežigrad	0	10	15	0	8	33
		Jan Živkovič	4	Vegova Ljubljana	0	13	12	1	7	33
	22	Gregor Mrak	4	ŠC Nova Gorica	5	9	13	0	3	30
		Mark Štokelj	3	ŠC Nova Gorica	0	7	12	1	10	30
	24	Tadej Medved	3	ŠC Nova Gorica	10	10	0	0	8	28
	25	Žan Knafelc	2	ZRI	2	9	8	2	6	27
		Nejc Prikeržnik	2	ŠC Ravne na Koroškem	0	9	13	0	5	27
		Vid Štrancar	2	SŠ V. Pilon Ajdovščina	7	0	12	0	8	27
	28	Simon Weiss	3	ZRI	5	7	8	0	2	22
	29	Matej Slemič	3	ŠC Nova Gorica	0	5	3	0	10	18
	30	Sandi Pangerc	3	ŠC Nova Gorica	0	3	10	1	3	17
	31	Drago Miklauc	2	ŠC Ravne na Koroškem	0	7	0	2	7	16
	32	Žan Žerdin Furlan	2	ZRI	0	7	0	0	7	14
	33	Aljaž Razpotnik	4	ŠC Ravne na Koroškem	0	0	0	1	5	6
		Rok Lesjak	3	ŠC Velenje, ERŠ	2	1	3	0	0	6
	35	Darko Kušer	3	ŠC Velenje, ERŠ	0	0	0	0	5	5
	36	Tina Lekše	1	ZRI + Gimnazija Vič	0	0	0	2	2	4
	37	Jernej Žolger	4	ŠC Ravne na Koroškem	0	0	2	0	0	2

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Patrik Zajec	3	ZRI + ŠC S. Kosovela Sežana	97	77	84	97	97	452
2Z	2	Vid Kocijan	3	ZRI	97	31	40	40		208
2S	3	Žiga Gradišar	3	ZRI + Šk. kl. gim. Lj.	100	54	0	30		184
2S	4	Rok Lekše	3	ZRI + Šk. kl. gim. Lj.	97	37	0	40		174
3	5	Filip Koprivec	3	Gimnazija Vič	100		40			140
3	6	Jan Aleksandrov	3	ZRI	97	25				122
	7	Tadej Ciglarič	4	Gimnazija Bežigrad	50	34	37	0		121
	8	Marko Novak	4	ZRI	67	37	0			104
	9	David Gričar	4	Gimnazija Moste		0	30	40		70

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Žiga Željko	tablični računalnik GoClever R76.2
1	1	Metod Medja	tablični računalnik GoClever R76.2
1	2	Jakob Košir	2 TB zunanji disk
1	2	Samo Remec	2 TB zunanji disk
1	3	Luka Kolar	64 GB USB flash disk
1	3	Jan Likar	64 GB USB flash disk
1	3	Bor Brecelj	64 GB USB flash disk
2	1	Lojze Žust	tablični računalnik GoClever R76.2 Dasgupta *et al.: Algorithms*
2	2	Žan Kunsterle	tablični računalnik GoClever R76.2 Dasgupta *et al.: Algorithms*
2	2	Blaž Blokar	2 TB zunanji disk Dasgupta *et al.: Algorithms*
2	2	Filip Peter Lebar	2 TB zunanji disk
2	3	Žiga Simončič	miška Razer Naga 2012
2	3	Tobias Mihelčič	2 TB zunanji disk
2	3	Aljaž Jeromel	64 GB USB flash disk
3	1	Patrik Zajec	tablični računalnik GoClever R76.2 Cormen *et al.: Introduction to algorithms*
3	2	Vid Kocijan	tablični računalnik GoClever R76.2 Cormen *et al.: Introduction to algorithms*
3	2	Žiga Gradišar	2 TB zunanji disk Knuth: *The Art of Computer Programming*, Vol. 4A
3	2	Rok Lekše	2 TB zunanji disk
3	3	Filip Koprivec	miška Razer Naga 2012
3	3	Jan Aleksandrov	miška Razer Naga 2012
Off-line naloga — Zlaganje likov			
	1	Tomaž Hočevar	magnetne kroglice NeoCube
	2	Patrik Zajec	magnetne kroglice NeoCube

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

SODELUJOČE ŠOLE IN MENTORJI

Druga gimnazija Maribor	Mirko Pešec
Gimnazija Bežigrad	Andrej Šuštaršič
Gimnazija in srednja šola Kočevje	Tanja Masterl
Gimnazija Kranj	Zdenka Vrbinc, Mateja Žepič
Gimnazija Litija	Jan Maver
Gimnazija Moste	Gorazd Kovačič, Aleš Razinger
Gimnazija Nova Gorica	Jurij Knez
Gimnazija Poljane	Janez Malovrh, Boštjan Žnidaršič
Gimnazija Šentvid	Nastja Lasič, Klemen Blokar
Gimnazija Škofja Loka	Anže Nunar
Gimnazija Vič	Klemen Bajec, Andreja Likar Cerc, Marjan Greselj, Jure Slak, Natan Žabkar
Srednja elektro-računalniška šola Maribor (SERŠ)	Slavko Nekrep
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Igor Kutoš, Karel Maček
Srednja šola Josipa Jurčiča Ivančna Gorica	Darko Pandur
Srednja šola Veno Pilon Ajdovščina	Marko Pregeljc
Srednja tehniška šola (STŠ) Koper	Andrej Florjančič
Škofijska klasična gimnazija Šentvid	Helena Medvešek, Nace Hudobivnik
Šolski center Celje, Gimnazija Lava	Karmen Kotnik, Tomislav Viher
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Dušan Fugina
Šolski center Kranj, Srednja šola za elektrotehniko in računalništvo (SŠER)	Aleš Hvasti
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Nova Gorica	Barbara Pušnar, Boštjan Vouk, Tomaž Mavri

Šolski center Novo mesto	Simon Vovko
Šolski center Ptuj, Elektro in računalniška šola	Zoltan Sep, Franc Vrbančič
Šolski center Srečka Kosovela Sežana	
Šolski center Ravne na Koroškem, Srednja šola Ravne	Gorazd Geč, Zdravko Pavleković
Šolski center Velenje, Elektro in računalniška šola	Miran Zevnik
Tretja gimnazija Maribor	Maja Čelan
Vegova Ljubljana	Nataša Makarovič, Marko Kastelic, Darjan Toth
Zavod za računalniško izobraževanje (ZRI), Ljubljana	

OFF-LINE NALOGA — ZLAGANJE LIKOV

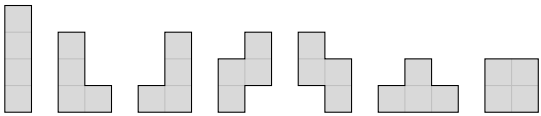
Na računalniških tekmovanjih, kot je naše, je čas reševanja nalog precej omejen in tekmovalci imajo za eno nalogo v povprečju le slabo uro časa. To med drugim pomeni, da je marsikak zanimiv problem s področja računalništva težko zastaviti v obliki, ki bi bila primerna za nalogo na tekmovanju; pa tudi tekmovalec si ne more privoščiti, da bi se v nalogo poglobil tako temeljito, kot bi se mogoče lahko, saj mu za to preprosto zmanjka časa.

Off-line naloga je poskus, da se tovrstnim omejitvam malo izognemo: besedilo naloge in testni primeri zanjo so objavljeni več mesecev vnaprej, tekmovalci pa ne oddajajo programa, ki rešuje nalogo, pač pa oddajajo rešitve tistih vnaprej objavljenih testnih primerov. Pri tem imajo torej veliko časa in priložnosti, da dobro razmislijo o nalogi, preizkusijo več možnih pristopov k reševanju, počasi izboljšujejo svojo rešitev in podobno. Takšne naloge smo razpisovali že v letih 2007 in 2008, letos pa smo s tem poskusili znova. Opis naloge smo objavili septembra 2012 skupaj z razpisom za tekmovanje v znanju, testne primere pa v začetku januarja 2013; tekmovalci so imeli čas do 22. marca 2013 (dan pred tekmovanjem), da pošljejo svoje rešitve.

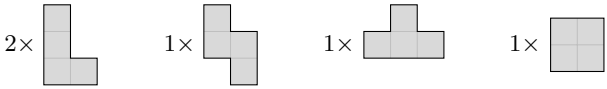
Opis naloge

Dano je veliko število likov iz igre Tetris. Naloga je zložiti like v večji lik s čim manjšim obsegom, pri čemer se liki med seboj ne smejo prekrivati. Pri tem je novi „lik“ lahko tudi sestavljen iz več nepovezanih delov, lahko vsebuje luknje in podobno. Obseg je definiran kot skupna dolžina vseh robov, pri katerih liki mejijo na belo podlago naše kariraste mreže (namesto na druge like).

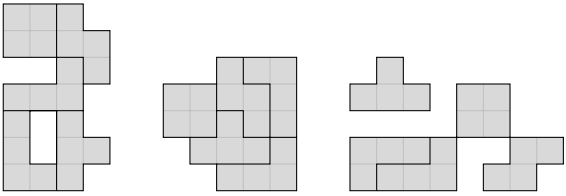
Možne oblike likov so naslednje:



Primer: recimo, da imamo naslednje like:



Teh pet likov lahko zložimo na veliko različnih načinov in dosežemo različno velike obsege. Naslednja slika prikazuje tri izmed njih in pod vsakim še njegov obseg:



Med temi tremi razporedi je torej najboljši tisti v sredini, ki ima obseg samo 20 enot.

Testni primeri
Pripravili smo 300 testnih primerov, pri vsakem od njih pa velja omejitev, da je število likov posamezne oblike kvečjemu 300. Skupno število likov pri vsakem testnem primeru je torej lahko največ 2100; ni pa nujno, da so v vsakem testnem primeru prisotni liki vseh sedmih oblik. Število testnih primerov je veliko zato, ker smo hoteli odvrniti ljudi od oddajanja ročno sestavljenih razporedov (po naših izkušnjah je z nekaj truda pogosto mogoče ročno dobiti zelo dobre razporede likov).

Rezultati
Sistem točkovanja je bil tak kot pri off-line nalogi v letih 2007 in 2008. Pri vsakem testnem primeru smo razvrstili tekmovalce po obsegu njihovega razporeda likov nato pa je prvi tekmovalec (tisti z najmanjšim obsegom) dobil 10 točk, drugi 9 in tretji 8. Na koncu smo za vsakega tekmovalca sešteli njegove točke po vseh tristo testnih primerih.

Žal smo letos dobili rešitve le od treh tekmovalcev, pri čemer je eden rešil le šest od 300 testnih primerov. Končna razvrstitev je naslednja:

Tomaž Hočevar (FRI)	2996 točk
Patrik Zajec (ZRI)	2423 točk
Matjaž Leonardis	55 točk

V upanju, da bo odziv prihodnje leto boljši, bomo enako nalogo (z novimi testnimi primeri) izvedli tudi v letu 2014. Zato bomo tudi razmislek o reševanju te naloge objavili v biltenu 2014.

UNIVERZITETNI PROGRAMERSKI MARATON

Društvo ACM Slovenija sodeluje tudi pri pripravi študenskih tekmovanj v programiranju, ki v zadnjih letih potekajo pod imenom Univerzitetni programerski maraton (UPM, `www.upm.si`) in so odskočna deska za udeležbo na ACMovih mednarodnih študentskih tekmovanjih v programiranju (International Collegiate Programming Contest, ICPC). Ker UPM ne izdaja samostojnega biltena, bomo na tem mestu na kratko predstavili to tekmovanje in njegove letošnje rezultate.

Na študentskih tekmovanjih ACM v programiranju tekmovalci ne nastopajo kot posamezniki, pač pa kot ekipe, ki jih sestavljajo po največ trije člani. Vsaka ekipa ima med tekmovanjem na voljo samo en računalnik. Naloge so podobne tistim iz tretje skupine našega srednješolskega tekmovanja, le da so včasih malo težje oz. predvsem predpostavljajo, da imajo reševalci že nekaj več znanja matematike in algoritmov, ker so to stvari, ki so jih večinoma slišali v prvem letu ali dveh študija. Časa za tekmovanje je pet ur, nalog pa je praviloma 6 do 8, kar je več, kot jih je običajna ekipa zmožna v tem času rešiti. Za razliko od našega srednješolskega tekmovanja pri študentskem tekmovanju niso priznane delno rešene naloge; naloga velja za rešeno šele, če program pravilno reši vse njene testne primere. Ekipe se razvrsti po številu rešenih nalog, če pa jih ima več enako število rešenih nalog, se jih razvrsti po času oddaje. Za vsako uspešno rešeno nalogo se šteje čas od začetka tekmovanja do uspešne oddaje pri tej nalogi, prišteje pa se še po 20 minut za vsako neuspešno oddajo pri tej nalogi. Tako dobljeni časi se seštejejo po vseh uspešno rešenih nalogah in ekipe z istim številom rešenih nalog se potem razvrsti po skupnem času (manjši ko je skupni čas, boljša je uvrstitev).

UPM poteka v štirih krogih (dva spomladi in dva jeseni), pri čemer se za končno razvrstitev pri vsaki ekipi zavrže najslabši rezultat iz prvih treh krogov, četrti (finalni) krog pa se šteje dvojno. Najboljše ekipe se uvrstijo na srednjeevropsko regijsko tekmovanje (CERC, ki je bilo letos 15.–17. novembra 2013 v Krakowu), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bo 21.–25. junija 2014 v Ekaterinburgu v Rusiji).

Na letošnjem UPM je sodelovalo 49 ekip s skupno 137 tekmovalci, ki so prišli z vseh treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednjih dveh straneh prikazuje vse ekipe, ki so se pojavile na vsaj enem krogu tekmovanja.

	Ekipa	Št. rešenih nalog*	Čas
1	Jure Slak, Žiga Gosar, Maks Kolman (FMF)	20	26:55:59
2	Erik Grabljevec, Blaž Sobočan, Sara Pohl (FMF)	15	16:50:53
3	Klemen Kloboves (FRI), Matej Aleksandrov (FMF), Matjaž Leonardis	14	15:49:41
4	Žiga Zalokar, Andraž Dobnikar (FRI + FMF), Tibor Djurica Potpara (FMF)	14	25:07:43
5	Andraž Bajt, Blaž Repas, Luka Zakrajšek (FRI)	12	14:12:21
6	Filip Kozarski, Matej Petković, Tomaž Stepišnik Perdih (FMF)	11	13:47:14
7	Sven Cerk, Martin Šušterič (FRI), Veno Mramor (FMF)	11	13:59:30
8	Igor Lalić, Janez Majdič, Miha Ravnikar (FRI)	11	20:03:57
9	Žiga Šmelcer, Žiga Gradišar, Lojze Žust (Šk. klas. gimn. Lj.)	10	11:25:57
10	Tomaž Kariž, Primož Kariž, Domen Mladovan (FRI)	10	12:51:35
11	Primož Godec, Manca Žerovnik, Luka Krsnik (FRI)	9	13:12:49
12	Ernest Beličič, Milutin Spasić, Jure Kolenko (FRI)	9	16:12:00
13	Jasna Urbančič, Rok Kaufman (FMF), Marko Novak (Vegova Ljubljana)	9	17:27:37
14	Beno Šircelj (FRI), Jaka Dolenc, Jure Kukovec (FMF)	9	17:36:05
15	Aleksandar Todorović, Marko Tavčar (FAMNIT)	9	18:24:30
16	Jure Senegačnik (F. za strojništvo, Lj.), Žiga Emeršič (FRI), David Fabijan (FMF)	8	8:42:41
17	Jure Grabnar, Matej Zrimšek, Matej Vehar (FRI)	8	10:35:12
18	Miha Eleršič, Fedja Beader, Petra Hvala (FRI + FMF)	8	10:43:53
19	Luka Firm (FRI), Gašper Medved, Žiga Lesar	8	14:32:45
20	Dragana Božović, Gregor Pirš, Martin Duh (FNM Maribor)	8	14:41:31
21	David Jesenko, Sebastijan Kužner, Robi Cvirn (FERI)	8	16:07:07
22	Matej Kren, Tim Kos, Aleksander Kelenc (FNM Maribor)	7	8:22:53
23	Žan Kustrle, Rok Poje, Jan Živkovič (Vegova Ljubljana)	7	9:51:53
24	Miha Zidar, Anže Pečar, Matic Potočnik (FRI)	7	11:13:02
25	Sandi Mikuš, Jan Vatovec (FRI), Aleksander Novaković (Medicinska f., Lj.)	7	15:19:04
26	Anže Žitnik, Matej Nanut, Kristian Zupan (FRI)	4	8:43:49
27	Tjaž Brelih, Jani Bevk (FRI), Rok Krhlikar (FE)	4	10:07:25
28	Jurij Slabanja, Mark Hočevar, Jan Jug (FRI)	3	5:36:00
29	Matjaž Kavčič, Manja Kocet, Andrej Dolenc (FERI)	3	7:11:38
30	Martin Frlin, Tomaž Žniderič (FRI)	2	3:18:58

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

(*nadaljevanje na naslednji strani*)

	Ekipa	Št. rešenih nalog*	Čas
31	Nejc Smrkolj Koželj, Tobias Mihelčič, Žiga Simončič (Vegova Ljubljana)	2	3:21:25
32	Andrej Maršič, Tomislav Luetić (FMF)	2	3:34:53
33	Klemen Bratec, Petra Mihalič, Primož Ocepek (FRI)	2	4:50:35
34	Domen Urh, Aleš Omerzel (FRI), Neža Žager Korenjak (FMF)	2	6:50:34
35	Rok Fortuna, Urban Marovt (FRI)	2	7:06:46
36	Klemen Forstnerič, Miha Krajnc (FERI)	2	7:33:01
37	Aljoša Mrak, Marko Čavdek (FRI)	1	1:55:10
38	Filip Koprivec, Filip Peter Lebar, Tina Lekše (Gim. Vič)	1	2:47:56
39	Matej Kramberger, Jure Zgorelec, Boštjan Budna (FERI)	1	2:53:39
40	Rudi Kovač, Tomaž Tomažinčič, Matej Kavrečič (FAMNIT)	1	4:18:15
41	Denis Kranjc, Damijan Račel, Danijel Cigula (FERI)	1	5:53:32
42	Simon Weiss, Jan Aleksandrov, Jan Kremser (Gim. Bežigrad)	0	0:00:00
	Klemen Lorenčič, Tomaž Sabadin, Matej Brlec (FAMNIT)	0	0:00:00
	Tomaž Šuen, Martin Kraner (FERI), Tjaša Hrovatič (FNM Maribor)	0	0:00:00
	Matej Pelko, Svit Timej Zebec (FRI)	0	0:00:00
	Jan Obu (FERI)	0	0:00:00
	Luka Arnečič, Niko Tratnik, Janez Dolšak (FNM Maribor)	0	0:00:00
	Jurij Podgoršek, Marija Stanojević (FERI)	0	0:00:00
	Marko Jovčeski (FNM Maribor), Tadej Vajdl, Robi Pritržnik (FERI)	0	0:00:00

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

Na srednjeevropskem tekmovanju so nastopile ekipe 1, 2 in 3 kot predstavnice Univerze v Ljubljani, 20 in 21 kot predstavnici Univerze v Mariboru in kombinacija ekip 15 + 40 kot predstavnica Univerze na Primorskem. V konkurenci 72 ekip s 30 univerz iz 6 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
28	Matej Aleksandrov, Žiga Ham, Klemen Kloboves	5	4:42
32	Maks Kolman, Žiga Gosar, Jure Slak	5	9:19
35	Blaž Sobočan, Sara Pohl, Erik Grabljevec	5	12:33
51	Aleksandar Todorović, Marko Tavčar, Tomaž Tomažinčič	3	1:44
66	Gregor Pirš, Dragana Božović, Martin Duh	2	1:23
69	Sebastijan Kužner, Robi Cvirn, David Jesenko	2	3:52

Na srednjeevropskem tekmovanju je bilo 12 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. 173–179.

Letnik: ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

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

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

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

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

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

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

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

Ali bi znal v programu uporabiti naslednje podatkovne strukture:

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

Ali bi znal v programu uporabiti naslednje algoritme:

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

[če misliš, da to pomeni uporabo new, GetMem, malloc ipd., potem obkroži „ne“]

Katerega od algoritmov za urejanje ☐ da ☐ ne

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

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

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

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

[Le pri 1. in 2. skupini.] Kakšno je tvoje mnenje o sistemu za oddajanje odgovorov prek računalnika? _____
_____

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

Katere od naslednjih jezikovnih konstruktov in programerskih prijemov znaš uporabljati?	ne poznam	da, slabo	da, dobro
Ali bi znal prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal prebrati kakšno celo število in kakšen niz iz datoteke ali pa ju zapisati v datoteko?	☐	☐	☐
Tabele (**array**):			
— enodimenzionalne	☐	☐	☐
— dvodimenzionalne	☐	☐	☐
— večdimenzionalne	☐	☐	☐
Znaš napisati svoj podprogram (**procedure**, **function**)	☐	☐	☐

Poznaš rekurzijo	☐	☐	☐
Kazalce, dinamično alokacijo pomnilnika (New/Dispose, GetMem/FreeMem, malloc/free, **new**/**delete**, ...)	☐	☐	☐
Zanka **for**	☐	☐	☐
Zanka **while**	☐	☐	☐
Gnezdenje zank (ena zanka znotraj druge)	☐	☐	☐
Naštevni tipi (*enumerated types* — **type** ImeTipa = (Ena, Dve, Tri) v pascalu, **typedef enum** v C/C++)	☐	☐	☐
Strukture (**record** v pascalu, **struct**/**class** v C/C++)	☐	☐	☐
and, **or**, **xor**, **not** kot aritmetični operatorji (nad biti celoštevilskih operandov namesto nad logičnimi vrednostmi tipa boolean) (v C/C++/C#/javi: &, |, ^, ~)	☐	☐	☐
Operatorja **shl** in **shr** (v C/C++/C#/javi: «, »)	☐	☐	☐
Znaš uporabiti kakšnega od naslednjih razredov iz standardnih knjižnic:			
hash_map, hash_set, unordered_map, unordered_set (v C++), Hashtable, HashSet (v javi/C#), Dictionary (v C#)	☐	☐	☐
map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#)	☐	☐	☐
priority_queue (v C++), PriorityQueue (v javi)	☐	☐	☐

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

Ostali komentarji o tej nalogi: _____

_____

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

_____

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

_____

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

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

Kaj ti je bilo pri tekmovanju všeč? _____

Kaj te je najbolj motilo? _____

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

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

Udeležil bi se oz. z veseljem bi spremljal:

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

Vesel(a) bi bil pomoči pri:

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

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

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 59 tekmovalcev prve skupine, 31 tekmovalcev druge skupine in (vseh) 9 tekmovalcev tretje skupine. (Opozorimo na to, da je zaradi majhnega števila tekmovalcev v tretji skupini iz tamkajšnjih anket še posebej težko vleči kakšne pametne posplošitve in zaključke.) Vprašanja so bila pri letošnji anketi 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. 174. 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: 3,28 v prvi skupini (v prejšnjih letih 3,39, 3,56, 3,34, 3,56), 3,35 v drugi skupini (prejšnja leta 3,50, 3,39, 3,38, 3,46) in 3,40 v tretji skupini (prejšnja leta 3,21, 3,57, 3,92).

Med tem, kako težka se je naloga zdela tekmovalcem, in tem, kako dobro so jo zares reševali (npr. merjeno s povprečnim številom točk pri tej nalogi), je bila šibka negativna korelacija, močnejša kot ponavadi ($R^2 = 0{,}52$, lani 0,21, predlani 0,11, pred tem okoli 0,4).

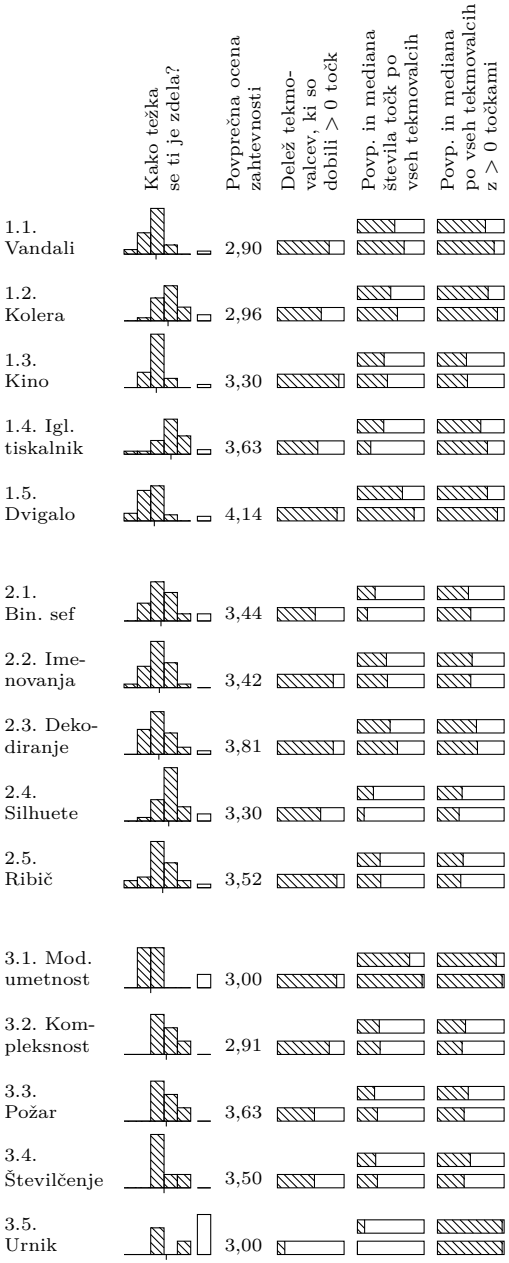
Največ pripomb o tem, kako da je naloga težka, je bilo pri nalogah 1.2 (kolera), 1.4 (iglični tiskalnik) in 2.4 (silhuete). Pri slednji si lahko mnenje, da je težki, mogoče razložimo s tem, da je malo bolj nestandardnega tipa in zahteva bolj razmislek kot nek dobro znan algoritem; 1.4 pa zahteva poznavanje operacij za delo z biti, s čimer ima mnogo tekmovalcev težave. Mnenje, da je naloga 1.2 težka, pa nas je presenetilo, saj ni bila mišljena kot ena od težjih nalog v prvi skupini.

Kot najlažjo so tekmovalci ocenili nalogo 1.5 (dvigalo), pri kateri je bilo celo tudi nekaj pripomb, da je prelahka. To je rahlo neobičajno, saj gre za „realnočasovni“ tip naloge in take se tekmovalcem običajno zdijo težje.

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 175. Nad razumljivostjo besedil ni veliko pripomb (podobno kot lani in še malo manj kot prejšnja leta); daleč najtežje razumljiva se jim je zdela naloga 1.4 (iglični tiskalnik), deloma pa tudi 1.2 (kolera), 2.4 (silhuete) in 3.5 (natrpan urnik).

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, približno enako kot v prejšnjih letih. Še največ pripomb na dolžino je pri nalogah 1.2 (kolera) in 2.2 (sumljiva imenovanja), ki sta se nekaterim zdeli predolgi.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta, le v drugi skupini malo nižje. Največ pripomb glede dolgočasnosti nalog je bilo v drugi skupini, še posebej pri nalogi 2.4 (silhuete). Kot bolj zanimivi



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

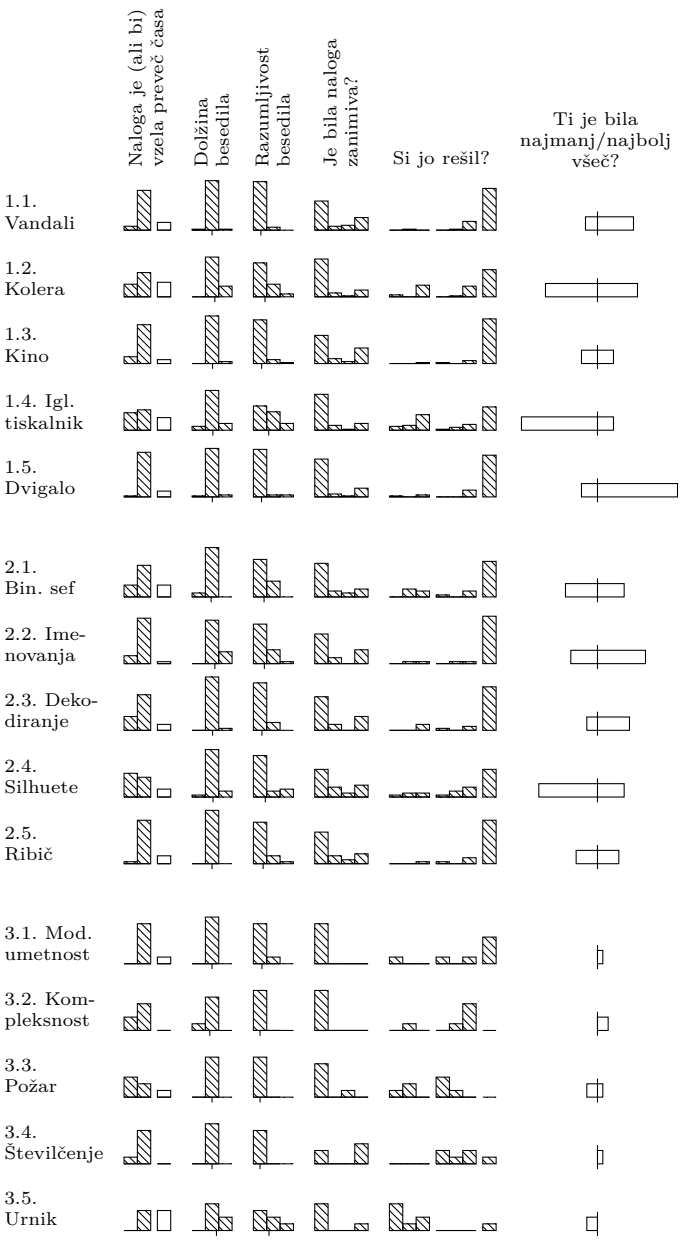
Pomen stolpcev v vsaki vrstici:

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?

izstopata nalogi 1.2 (kolera) in 1.5 (dvigalo). Največ pripomb, češ da je že znana, pa je bilo pri nalogi 1.1 (vandali).

Pripomb, da bi naloga vzela preveč časa, je bilo manj kot v prejšnjih letih. Največ takih pripomb je bilo pri tistih nalogah, ki so se zdele tekmovalcem težke: 1.4 (iglični tiskalnik), 2.4 (silhuete) in 3.3 (požar).

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, sta kot bolj popularni izstopali nalogi 1.5 (dvigalo; ker se jim je zdela zanimiva, mnogim pa tudi uporabna) in 2.2 (sumljiva imenovanja; v veliki meri zaradi zgodbe, v katero je zavita), kot manj popularni pa 1.4 (iglični tiskalnik) in 2.4 (silhuete), ker sta se jim zdeli težki in nerazumljivi. Zanimiv primer je še naloga 1.2 (kolera), ki je dobila veliko glasov v v obeh kategorijah (nekaterim se je zdela zanimiv izziv, nekaterim pa nerazumljiva in dolgočasna).

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.

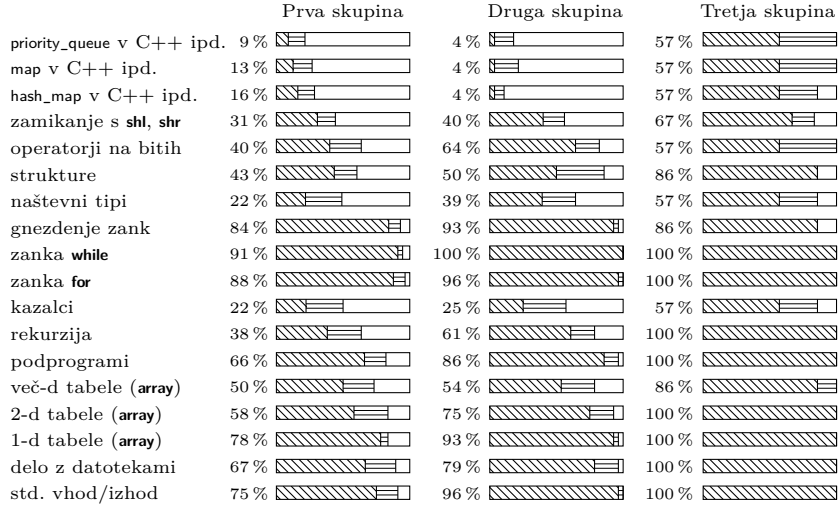


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: rekurzija, kazalci, naštevni tipi in operatorji na bitih, v prvi skupini tudi strukture.

V izvorni kodi rešitev, ki jih tekmovalci oddajajo med tekmovanjem, je videti, da tisti, ki uporabljajo C++ namesto C, vse pogosteje posegajo po tistih elementih jezika, ki jih v C-ju ni, na primer razreda vector in string namesto tradicionalnih tabel (*arrays*).

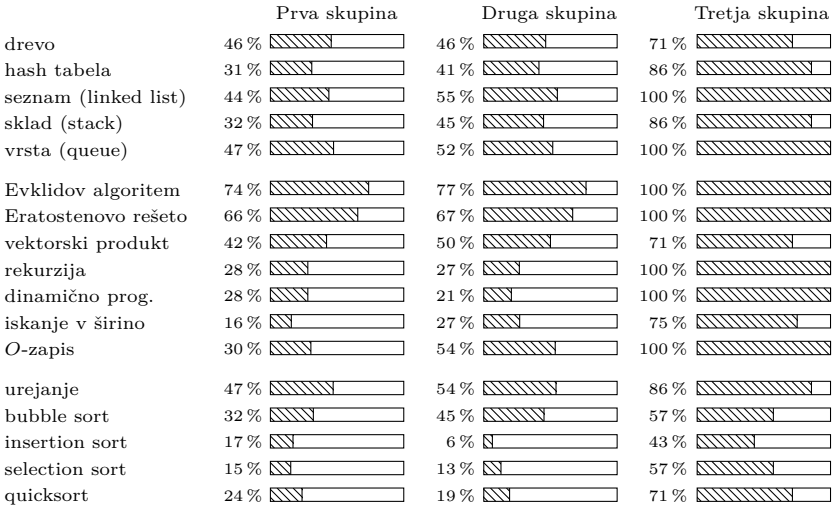


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

Uporaba programskih jezikov

Največ tekmovalcev tudi letos uporablja C/C++ (podobno kot zadnja leta je čisti C pravzaprav že redek), je pa njegova prednost pred drugimi jeziki manjša. V prvi skupini so po pogostosti uporabe zdaj praktično izenačeni C++, C#, java in python, drugi jeziki pa so zelo redki. Zanimivo pa je, da čeprav je C# zdaj tako pogost v prvi skupini, ga v drugi ni uporabljal skoraj nihče (podobno je bilo že lani); v drugi skupini je najpogostejši C/C++, drugo mesto pa si delita python in java. V tretji skupini so vsi razen dveh tekmovalcev pisali v C++. Dolgoletni trend upadanja uporabe pascala je prišel letos že tako daleč, da sta ga uporabljala le še dva tekmovalca.

Podobno kot prejšnja leta se je tudi letos pojavilo nekaj tekmovalcev (in tekmovalk), ki oddajajo le rešitve v psevdokodi ali pa celo naravnem jeziku, tudi tam, kjer naloga sicer zahteva izvorno kodo v kakšnem konkretnem programskem jeziku. Iz tega bi človek mogoče sklepal, da bi bilo dobro dati več nalog tipa „opiši postopek“ (namesto „napiši podprogram“), vendar se v praksi običajno izkaže, da so takšne naloge med tekmovalci precej manj priljubljene in da si večinoma ne predstavljajo preveč dobro, kako bi opisali postopek (pogosto v resnici oddajo dolgovezne opise izvorne kode v stilu „nato bi s stavkom **if** preveril, ali je spremenljivka x večja od spremenljivke y“). V bodoče bomo poskusili navodila takih nalog formulirati kot „opiši postopek ali napiši podprogram (kar ti je lažje)“.

Podobno kot v prejšnjih letih je v anketi precej tekmovalcev napisalo, da dobro poznajo tudi PHP, vendar sta ga na tekmovanju uporabljala le dva.

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže tabela na str. 178. 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

	Leto in skupina																	
	2013			2012			2011			2010			2009			2008		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	1		1	6	1	4	3	4	3	$4\frac{1}{2}$	5	2	4	2	1	$1\frac{1}{2}$	2	2
C	2	7		7	2	1	7	2		6	6	1	$9\frac{1}{2}$	$3\frac{1}{2}$	$\frac{1}{2}$	$4\frac{1}{2}$	11	$2\frac{1}{2}$
C++	17	$12\frac{1}{2}$	7	26	16	9	$23\frac{1}{2}$	19	8	33	$17\frac{1}{2}$	13	$26\frac{1}{2}$	2	$12\frac{1}{2}$	$17\frac{1}{2}$	11	$9\frac{1}{2}$
java	12	8	1	17	$6\frac{1}{2}$	1	6	5	3	5	9	4	8	8	11	$9\frac{1}{2}$	3	
PHP	1	$\frac{1}{2}$	–		1	–	$\frac{1}{2}$		–	1	1	–	2	1	–		2	–
basic	1		–			–			–			–			–			–
C#	18	$\frac{1}{2}$		17	1	3	4	2	3		$\frac{1}{2}$	1						3
python	16	8	–	25	5	–	20	6	–	12	2	–	4	$\frac{1}{2}$	–	6	1	–
NewtonScript		$\frac{1}{2}$	–		$\frac{1}{2}$	–			–			–			–			–
psevdokoda	6		–	3		–	6		–	4		–	8		–	–	–	–
nič	2			2			1	1		1	5		1				1	

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“.

katerih se C++ loči od C-ja, sčasoma povečuje (na primer `string` namesto **`char *`** in tip `vector` namesto tradicionalnih tabel).

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, C#, pythonu in javi. Ta nabor jezikov očitno kar dobro pokrije znanje naših tekmovalcev, saj je delež tekmovalcev, ki pravijo, da deklaracije razumejo, visok (50/57 v prvi skupini in 27/27 v drugi). Nenavadno je, da so pri vprašanju, ali bi želeli deklaracije še v kakšnem jeziku, nekateri tekmovalci navedli jezike, v katerih deklaracije že imamo, na primer javo in C#.

V rešitvah nalog zadnja leta objavljamo izvorno kodo le v C-ju; tekmovalce smo v anketi vprašali, če razumejo C dovolj, da si lahko kaj pomagajo s to izvorno kodo, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Večina je s C-jem zadovoljna (51/98 v prvi skupini, 21/26 v drugi, 7/7 v tretji; to je približno enak ali še malo boljši delež kot lani). Med jeziki, ki bi jih radi videli namesto (ali poleg) C-ja, jih največ omenja javo, C# in python.

Letnik

Po pričakovanjih so tekmovalci zahtevnejših skupin v povprečju v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot prejšnja leta; v prvi skupini je povprečja starost malo višja kot lani, v drugi pa malo nižja. V prvi skupini je nastopil tudi en osnovnošolec; tega pri izračunu povprečnega letnika v spodnji tabeli nismo upoštevali.

	Št. tekmovalcev po letnikih					Povprečni
Skupina	OŠ	1	2	3	4	letnik
prva	1	10	22	16	26	2,8
druga		2	6	17	12	3,1
tretja				6	3	3,3

Druga vprašanja

Podobno kot prejšnja je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!). V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale.

Pri vprašanju, kje so se naučili programirati, podobno kot prejšnja leta prevladujeta odgovora „sam“ in „v šoli“, malo manj kot prejšnja leta pa je tekmovalcev, ki pravijo, da so se programirati naučili na krožkih.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. Nekaj več tekmovalcev kot prejšnja leta si želi, da bi tekmovanje v prvi in drugi skupini trajalo manj časa (pri nespremenjenem številu nalog). Časa za reševanje nalog sicer do nadaljnjega ne nameravamo skrajševati, saj si pri sestavljanju nalog želimo, da bi izziv za tekmovalce predstavljala predvsem zahtevnost nalog, ne pa pomanjkanje časa za reševanje.

	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	52	0	3	3	40	24	10	2	3	3	8	38	5	4	40	19	21	14	19	18	16	23	27
II	29	0	1	1	16	17	6	4	2	1	6	20	1	6	20	8	11	9	9	10	7	13	15
III	7	0	0	0	6	1	2	2	2	0	0	5	0	2	4	4	3	4	2	6	5	3	3

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. Najpogostejša (in povsem upravičena) pripomba tekmovalcev v zvezi s sistemom za oddajo odgovorov na računalniku je bila, da ta sistem na začetku tekmovanja nekaj časa ni deloval, kot bi moral.

Veliko tekmovalcev si je tudi želelo, da bi imeli v prvi in drugi skupini na računalniki prevajalnike in podobna razvojna orodja. Razlog, zakaj se v teh dveh skupinah izogibamo prevajalnikom, je predvsem ta, da hočemo s tem obdržati poudarek tekmovanja na snovanju algoritmov, ne pa toliko na lovljenju drobnih napak; in radi bi tekmovalce tudi spodbudili k temu, da se lotijo vseh nalog, ne pa da se zakopljejo v eno ali dve najlažji in potem večino časa porabijo za testiranje in odpravljanje napak v svojih rešitvah pri tistih dveh nalogah.

CVETKE

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

(1.1) Iz rešitve z veliko komentarji:

```
char cur = t1.charAt(i); // Komentar, da se bo porabilo več črnila na printerju...
                                                        // Prosim ignoriraj.
```

(1.1) Komentar na koncu ene od rešitev (ki je tudi res pravilna):

```
return T1; // vrne string T1 in če je string T1 pravilna rešitev bo rdeča kapica srečna...
```

(1.1) Nov prispevek v boju proti zapostavljanju stavka else:

```
if i in T2:
   pass # ČE JE NOTRI NE NAREDI NIČESAR
else:
   ⋮
```

(1.1) V nekaterih jezikih je **if** lahko tudi zanka:

To izvedemo z zanko **if** v primeru uporabe pythona.

(1.1) Rešitev z veliko spremenljivkami:

```
String A1 = "še dobro, da lahko z volitvami vplivamo na dobrobit naroda.";
⋮
// Definirane vse črke v verigi A1
char b1 = A1.charAt(0);
char b2 = A1.charAt(1);
⋮
char b24 = A1.charAt(23);
```

Vseh 24 vrstic je bilo zapisanih eksplicitno. Zanimivo vprašanje je, zakaj se je ustavil pri 24, saj je niz A1 dolg kar 59 znakov.

(1.2) Podobno kot lani imamo tudi letos rešitev, ki skrbi za čiščenje podatkov:

```
if (vodnjak != null) { // preveri, če so bili vodnjaki pravilno preprani; če niso bili,
                                                        // nič ne izpiše
```

(1.2) Rešitev za ljubitelje velikih celoštevilskih konstant:

```
minimum = 99999999999999 # najmanjša razdalja; na začetku je zelo velika vrednost,
                         # da jo prepišemo s prvo naslednjo dejansko razdaljo
```

Še dlje je šel nek drug tekmovalec:

```
razdalja = 10000000000000000
```

(1.2) Komentar iz rešitve, ki bere vhodne podatke in izpiše rezultate — manjka le del, ki bi rezultate izračunal:

```
// Če bi nalogo razumel, bi tukaj napisal nek FOR, ki bi nekaj zračunal.
```

(1.2) Rešitev za ljubitelje zelo ne-evklidske geometrije:

```
razdalja = math.sqrt(((x1[i] * x1[i]) − (x * x)) +
                                        ((y1[i] * y1[i]) − (y * y))); /* Pitagorov izrek */
if (najrazdalja > math.abs(razdalja)) /* absolutna vrednost */
```

To napačno razumevanje Pitagorovega izreka je še toliko bolj presenetljivo, ker smo pravo formulo podali kar v besedilu naloge. Sodeč po drugi vrstici je tekmovalca tudi skrbelo, da bi bil kvadratni koren lahko negativen...

(1.2) Rešitev za ljubitelje Windowsov:

> Začni program. Vprašaj za datoteko, kjer je podana slika `.jpg`, zemljevida. Prikliči kordinatni sistem iz windowsa. Kordinatni sistem priklican iz windowsa poimenuj „Kor1“. Prečitaj sliko z pomočjo Kor1, in si označi kordinatne točke: [...]

V samem programu potem sicer od teh idej ni ostalo veliko:

```
int main()
{
  int PATH_MAX;
  cout << "Doloci pot do slike .jpg" << endl;
  cin >> PATH_MAX;
  system("PAUSE");
  return 0;
}
```

(1.3) O nacionalizaciji spremenljivk:

> [...] najkrajši čas se posebej napiše v spremenljivko, ki smo jo deklarirali in inacionalizirali že prej.

(1.3) Rešitev z veliko posmehovanja junaku iz besedila naloge:

> Nato pogledamo, ali je Mirko (haha) že zamudil zadnji avtobus. [...] Na koncu še izpišemo število filmov, ki jih naj Mirko (haha) pogleda, [...]

(1.3) Rešitev z veliko pomnilniki:

> Začni program. Rezerviraj pomnilnike. Določi, a = ure, b = minute. [...] Čitaj iz pomnilnikov a in b, kakšna sta podatka. [...] Rezultat zapiši v pomnilnik c.

(1.3) Eden od tekmovalcev je opise postopkov (za razliko od izvorne kode) najbrž vnašal telepatsko:

```
// Opisi postopkov so tako čudni... Stalno se je treba boriti skušnjavi pred tipkanjem
```

(1.4) Impresivno nepotrebna uporaba števil s plavajočo vejico:

```
for (float q = 128.0; q >= 1.0; q /= 2.0) {
```

(1.4) Prijetno ekscentričen način za preverjanje, ali je spremenljivka večja ali enaka 128:

```
if (stevilka / 128 >= 1)
```

(1.4) Zakaj bi napisali preprosto j = j/2, če pa lahko zakompliciramo (in tako nehote povzročimo, da se zanka pri j = 1 zacikla):

```
for (int j = 128; j > 0; j = j − j/2)
```

(1.4) Še ena rešitev z velikim zaupanjem v Windows API:

> Začni podprogram. Vprašaj za pot do slike. Sliko razbij na bite, z pomočjo windowsovega razdelilca bitov.

(1.5) Prijetno staromoden zaimek:

> Tam se v zanki, ki se ponovi tolikokrat, kolikoršnje je število zabojev, kliče metoda Razlozi(), ki premakne zaboje na trak.

(1.5) Prijetno ekscentričen pristop k neskončni zanki:

```
x = 1
y = 7
while (x + 1 − 5 + 2/3 <> y):
```

(1.5) Še en prispevek proti zapostavljanju stavka else:

```
if (vsota == 500)
   vsota = vsota;
else
   vsota = vsota − a;
```

(1.5) Ustavitveni pogoj, ki mu je težko oporekati:

> Zanko ponavljaj tako dolgo, dokler uporabnik ne izklopi programa, z izključitvijo elektrike.

(2.1) Rešitev za ljubitelje pravljic:

```
vector<string> kombinacije; // v to košaro rdeča kapica shranjuje vse možne kombinacije
⋮
kombinacije.push_back(k); // rdeča kapica shrani trenutno kombinacijo v košaro
```

(2.1) Rešitev z metodo grobe sile:

```
public static boolean Sef(String s, int n) {
   // najdi vsako varianto v binarnem (pretvarjanje), pretvori v string in nato z
   // indexOf preveri, ce je v nizu. Ce je, pojdi naprej, če ne pa se ne da reci,
   // da bo niz odklenil... Mah, ni cajta za tako, lahko vse rešiš z pajserjem =)
}
```

(2.1) Pogumen poskus, kako priti do posameznih bitov v 8-bitnem celem številu:

```
// Unite kjer bomo postavili char in vn dobili 8 bitov!
unite binary
{
  char ch;
  bool boolean[8];
};
```

Žal bo tabela boolean dolga vsaj 8 bytov, saj jezik prevajalniku ne dopušča, da bi za posamezni element tabele porabil manj kot 1 byte.

(2.1) Rešitev z veliko izmišljenimi funkcijami:

```
// pretvorimo char v int z ukazom ctoint (na novo izumljen ukaz :D)
int st = ctoint(a); // dobimo stevilko v binarnem zapisu
⋮
// PS: ker na žalost ne poznam ukaza za pretvorbo iz binarnega v decimalni in obratno,
// sem si ga izmislil in sicer sta to tadva ukaza
// tobin(int) (v binarno)
// todec(int) (v decimalno)
// Prav tako sem si izmislil ukaz za preverjanje, če je le to zaporedje v stringu s, in ta
// ukaz je: include(s, int) — ukaz vrne 1, če string vsebuje to zaporedje, ter 0, če ga ne
```

(2.1) Rešitev z zgražanjem nad avtorji nalog:

```
# verjetno bi deloval, če bi želeli preveriti le, če je koda prava.
# Če to ni bil namen, potem spremenite pisca navodil naloge.
```

(2.2) Rešitev z velikim nezaupanjem v državne organe:

```
print "zakonito" # drugače sprintamo, da je zakonito zaposlen, čeprav vsi vemo,
                                                    # da to ni res
```

(2.3) Eden od tekmovalcev je oddal rešitev v psevdokodi, ki je v celoti zapisana z velikimi črkami:

DEKLARIRAMO SPREMENJLJIVKE, ŠTEVCE;
PREBEREMO NIZ IN GA ZAPIŠEMO SVOJO SPREMENLJIVKO;
DOKLER SO PODATKI NA STANDARDNEM VHODU:

in tako naprej.

(2.3) Rešitev, ki se dela z nizi loti zelo resno:

Najprej bi vključil knjižnico lenght ali pa bi vse to izpeljal z stringi.

(2.4) Zanimiv pristop k branju vhodnih podatkov:

```
FILE* f = fopen("SILHUETE.txt", "r");
fprintf(f);
// nato bi prebral zapise v tabeli
int w, h; // prvi dve števili bi povedali w ter h
cin << w, "f";
cin << h, "f";
```

(2.5) Zelo pazljiva rešitev:

```
// Predno začnemo se lahko prepričamo, če smo v ta pravem svetu
if (! true)
{
   cout << "V NAPAČNEM SVETU SMO!!!";
   exit(0);
}
```

(2.5) Rešitev brez pretirane naglice:

> 1. ribič si najprej lepo počasi ogleda ribnik, ter si v program shrani natančne lokacije rib (naprimer v **char** ribe)

(2.5) Ena od neugodnih posledic tega, da besedilo naloge ovijemo v zgodbico, kot je na primer tale z ribičem, je, da se začnejo ljudje potem nad zgodbo pritoževati, ker je preveč skregana z zdravo pametjo:

```
# ne vem, zakaj človek ni zadovoljen, če ujame več rib, kot jih želi.
```

Mogoče ima ribolovno dovolilnico samo za k rib ali kaj podobnega `:P`

(3.2) Zakaj po tistem, ko smo že preverili, da je x večkratnik i-ja (oba sta tipa **int**), preverjamo še, če je x tudi večkratnik števila x / i, in celo računamo količnik x / (x / i)?

```
if (x % i == 0 && (x % (x / i)) == 0)
   res = min(res, f(x / i) + f(x / (x / i)));
```

(3.2) Najdaljša oddaja letos: 5242 vrstic, 408 kilobytov. Program je vseboval med drugim tabelo vseh praštevil do 300 000 in njihove domnevne kompleksnosti (že v teh so napake). Za sestavljena števila je potem predpostavil, da najmanjši izraz za število z vrednostjo $p_1 \cdot p_2 \cdot \ldots \cdot p_k$ (pri čemer so p_i praštevila, ne nujno različna) dobimo tako, da vzamemo najmanjše izraze za števila $p_1, \ldots, p_k$ in jih povežemo z operatorjem $\cdot$. V splošnem to ne drži; na primer, ta formula bi nam za $n = 46 = 2 \cdot 23$ napovedala $f(46) = f(2) + f(23) = 2 + 11 = 13$, v resnici pa je mogoče 46 dobiti že z izrazom, ki vsebuje samo 12 enic (izraz je oblike $1 + 3 \cdot 3 \cdot 5$).

Isti tekmovalec je podobno, a krajšo rešitev oddal že tričetrt ure prej; tam je imel le praštevila do $\sqrt{300\,000}$.

(3.3) Nekdo je bil med programiranjem očitno zelo lačen:

```
for (int majoneza = 0; majoneza < st; majoneza++) {
   ⋮
   queue<veza> rostilj;
   ⋮
   vector<bool> cebula;
   ⋮
   veza zrezek;
   ⋮
   rostilj.push(zrezek);
```

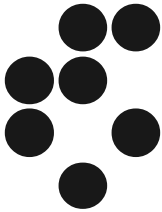
Rešitev je bila sicer drugače pravilna, le malo počasna.

(3.3) Letošnjo nagrado za najglobljo indentacijo dobi:

```
int main()
{
   ⋮
   for (int q = 0; q < p; q++)
   {
      ⋮
      while (count < k)
      {
         for (int i = 0; i < h; i++)
         {
            for (int j = 0; j < w; j++)
            {
               if (office[i][j] == −1)
               {
                  if (0 < i && i < h − 1)
                  {
                     if (office[i − 1][j] == z)
                     {
                        office[i][j] = z + 1;
```

SODELUJOČE INŠTITUCIJE

Institut Jožef Stefan



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

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

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

Tekmovanje sta podprla naslednja odseka IJS:

CT3 — Center za prenos znanja na področju informacijskih tehnologij

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

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

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

E3 — Laboratorij za umetno inteligenco
Področje dela Laboratorija za umetno inteligenco so informacijske tehnologije s poudarkom na tehnologijah umetne inteligence. Najpomembnejša področja raziskav in razvoja so: (*a*) analiza podatkov s poudarkom na tekstovnih, spletnih, večpredstavnih in dinamičnih podatkih, (*b*) tehnike za analizo velikih količin podatkov v realnem času, (*c*) vizualizacija kompleksnih podatkov, (*d*) semantične tehnologije, (*e*) jezikovne tehnologije.

Laboratorij za umetno inteligenco posveča posebno pozornost promociji znanosti, posebej med mladimi, kjer v sodelovanju s Centrom za prenos znanja na področju informacijskih tehnologij (CT3) razvija izobraževalni portal VideoLectures.NET in vrsto let organizira tekmovanja ACM v znanju računalništva.

Laboratorij tesno sodeluje s Stanford University, University College London, Mednarodno podiplomsko šolo Jožefa Stefana ter podjetji Quintelligence, Cycorp Europe, LifeNetLive, Modro Oko in Envigence.

*

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

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

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

Fakulteta za računalništvo in informatiko
Glavna dejavnost Fakultete za računalništvo in informatiko Univerze v Ljubljani je vzgoja računalniških strokovnjakov različnih profilov. Oblike izobraževanja se razlikujejo med seboj po obsegu, zahtevnosti, načinu izvajanja in številu udeležencev. Poleg rednega izobraževanja skrbi fakulteta še za dopolnilno izobraževanje računalniških strokovnjakov, kot tudi strokovnjakov drugih strok, ki potrebujejo znanje informatike. Prav posebna in zelo osebna pa je vzgoja mladih raziskovalcev, ki se med podiplomskim študijem pod mentorstvom univerzitetnih profesorjev uvajajo v raziskovalno in znanstveno delo.

Fakulteta za elektrotehniko, računalništvo in informatiko

Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. K temu bo fakulteta v prihodnjih letih (2009–2013) z razvojem kakovostnega oblikovanja in izvajanja naravoslovnih študijskih programov tudi stremela. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija

ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

REPUBLIKA SLOVENIJA
MINISTRSTVO ZA IZOBRAŽEVANJE, ZNANOST IN ŠPORT

Ministrstvo za izobraževanje, znanost in šport

Ministrstvo za izobraževanje, znanost in šport opravlja upravne in strokovne naloge na področjih predšolske vzgoje, osnovnošolskega izobraževanja, osnovnega glasbenega izobraževanja, nižjega in srednjega poklicnega ter srednjega strokovnega izobraževanja, srednjega splošnega izobraževanja, višjega strokovnega izobraževanja, izobraževanja otrok in mladostnikov s posebnimi potrebami, izobraževanja odraslih, visokošolskega izobraževanja, znanosti, ter športa.

BRONASTI POKROVITELJ

XLAB
NOT IDLE