14. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 23. marca 2019

Bilten

Bilten 14. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2020

Elektronska izdaja

Uredil Janez Brank

Avtorji nalog: Urban Duh, Primož Gabrijelčič, Matija Grabnar, Tomaž Hočevar, Boris Horvat, Branko Kavšek, Vid Kocijan, Samo Kralj, Mitja Lasič, Matjaž Leonardis, Matija Lokar, Mark Martinec, Polona Novak, Jure Slak, Jasna Urbančič, Patrik Zajec, Janez Brank.

Naklada: 170 izvodov

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

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

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

37.091.27:004(497.4)
004.42(497.4)(079.1)(082)

TEKMOVANJE ACM v znanju računalništva (14 ; 2019 ; Ljubljana)
Bilten / 14. tekmovanje ACM v znanju računalništva, Ljubljana, 23. marca 2019 ; [avtorji nalog Urban Duh ... [et al.] ; uredil Janez Brank]. — Ljubljana : Institut Jožef Stefan, 2020

ISBN 978-961-264-178-8
1. Duh, Urban 2. Brank, Janez, 1979–
COBISS.SI-ID 303535872

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. Ponavadi lahko tekmovalci v teh dveh skupinah pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom in velika večina se jih odloči za slednje, letos pa so morali žal zaradi tehničnih težav vsi pisati odgovore na papir.

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 23.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C#, java in python.

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, pa tudi z dokumentacijo raznih programskih jezikov, ki je nameščena na tekmovalnih računalnikih.

Na začetku smo tekmovalcem razdelili tudi list z nekaj nasveti in navodili (str. 7–9 za 1. in 2. skupino, str. 23–25 za 3. skupino).

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

Od leta 2017 objavljamo v biltenu rešitve v C++17, za prvo skupino pa tudi v pythonu, ker precej tekmovalcev v tej skupini še ne pozna nobenega drugega jezika.

Poleg tekmovanja v znanju računalništva smo organizirali tudi tekmovanje v off-line nalogi, ki je podrobneje predstavljeno na straneh 157–160.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 18. januarja 2019. 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. 143–146). 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 335 tekmovalcev s 31 šol (dve od njih sta bili osnovni, ostale pa srednje).

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 prek računalnika. 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 na približno dve minuti samodejno shrani. 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 želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku (uporabi ikono „Urejevalnik teksta“ na namizju). **Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave in želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici.**

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. Smučarski užitki

Smučar Matej se odpravlja na sobotno smučanje. V petek zvečer si je podrobno ogledal shemo smučarskih prog, teren in vremensko napoved. Na podlagi tega je za vsako progo določil oceno užitka — realno število med 1 in 10, ki označuje, koliko uživa med smučanjem po tej progi. Zanima ga skupna vsota užitka po vseh vožnjah, pri čemer pa, če se po posamezni progi zapelje večkrat, se ocena užitka smučanja po tej progi vsakič zmanjša na 90 % prejšnje vrednosti (ne glede na to, ali je vmes smučal po drugih progah ali pa so bile vožnje zaporedne).

Smučar Matej je že v petek zvečer vedel, da maksimalnega možnega smučarskega užitka ne bo mogel doseči, saj ne more vnaprej vedeti, koliko ljudi bo smučalo in kakšne bodo zato vrste za žičnice. V soboto si je beležil vrstni red prog, po katerih je smučal, zdaj pa te prosi, da iz njegovih zapiskov izračunaš, kakšen je bil njegov smučarski užitek v soboto.

Listek z vhodnimi podatki, ki ti ga je dal smučar Matej, vsebuje:

- v prvi vrstici je zapisano število prog, p, ki jih ima smučišče (p je največ 1000);

- v naslednjih p vrsticah so zapisane začetne ocene užitka za vsako progo (proge so oštevilčene od 1 do p);
- sledi neznano število vrstic; v vsaki je po ena številka med 1 in p — številka proge, po kateri je smučal smučar Matej.

Napiši program, ki prebere te podatke in izračuna ter izpiše vsoto užitka po vseh opravljenih vožnjah. Podatke lahko bereš s standardnega vhoda ali pa iz datoteke `smucanje.txt` (karkoli ti je lažje). Pri izračunu in izpisu ne skrbi glede drobnih zaokrožitvenih napak, do katerih lahko pride pri delu z realnimi števili.

Primer vhoda:

```
3
7
3.3
6
1
3
2
1
1
2
```

Pripadajoči izhod:

```
31.24
```

Razlaga: ko progo številka 1 prevozimo drugič, njena ocena ni več 7, ampak le še 6,3; ko jo prevozimo tretjič, pa le še 5,67. Podobno, ko progo številka 2 prevozimo drugič, njena ocena ni več 3,3, ampak le še 2,97. Vsota užitka po vseh vožnjah je zato $7 + 6 + 3{,}3 + 6{,}3 + 5{,}67 + 2{,}97 = 31{,}24$.

2. Razmazani seznam

Dana je množica $A = \{a_1, \ldots, a_n\}$, katere elementi so cela števila, večja od 0. Poleg tega sta dani še dve konstanti m in v, ki sta tudi celi števili, večji od 0. Definirajmo novo množico B, ki je „razmazana“ različica množice A, in sicer z naslednjim pravilom: celo število x pripada množici B natanko tedaj, ko je za kvečjemu m manjše ali za kvečjemu v večje od nekega elementa množice A (z drugimi besedami: ko za nek i velja $a_i - m \leq x \leq a_i + v$; ali pa še drugače: B vsebuje poleg vsakega elementa množice A še prejšnjih m in naslednjih v celih števil).

Množice A ne dobimo podane v datoteki ali v kakšni podatkovni strukturi, pač pa je za dostop do množice A na voljo funkcija Naslednji(), ki nam ob vsakem klicu vrne po en element množice A. Vrača jih v naraščajočem vrstnem redu. Ko pride do konca množice (torej od vključno $(n+1)$-vega klica naprej), pa odtlej vrača vrednost -1. Vrednosti n vnaprej ne poznamo (dokler torej ne pridemo do konca množice A, ne vemo, kako velika je; lahko je tudi zelo velika).

Napiši program ali podprogram (funkcijo), ki prebere množico A in izpiše elemente množice B. Izpisuje naj jih v naraščajočem vrstnem redu (vsakega natanko enkrat) in to čim bolj sproti; z drugimi besedami, preden prebere naslednji element množice A, naj izpiše vse tiste elemente množice B, za katere lahko iz že doslej prebranih podatkov sklepa, da res pripadajo množici B. Podrobnosti pri formatu izpisa niso pomembne; pišeš lahko na standardni izhod ali pa v datoteko `seznam.txt` (karkoli ti je lažje). Za konstanti m in v lahko predpostaviš, da sta že deklarirani na

začetku tvojega programa, ali pa ju prebereš s standardnega vhoda ali iz datoteke `vhod.txt` (karkoli ti je lažje).

Primer: če imamo $m = 2$ in $v = 1$ ter množico $A = \{1, 3, 4, 9\}$, ima množica B naslednje elemente: $\{-1, 0, 1, 2, 3, 4, 5, 7, 8, 9, 10\}$. (**Pozor:** tvoja rešitev **ne sme** predpostaviti, da so m, v in A takšni kot v tem primeru, ampak mora delovati za poljubne vrednosti m, v in A.)

3. Veriga

V daljšem besedilu, natisnjenem s pisavo konstantne širine (vsi znaki so enako široki), bi radi poiskali najdaljšo *verigo* enakih znakov.

Veriga je zaporedje enakih znakov (lahko gre za črko, številko, presledek, ločilo ali kaj drugega), ki se ponavlja v zaporednih vrsticah na enakem položaju znotraj vrstice. Pri tem razlikujemo velike in majhne črke („`a`“ ni enak „`A`“).

V spodnjem besedilu je na primer najdaljša veriga dolga šest znakov „`a`“, ki jih najdemo na drugem mestu druge, tretje, ... in sedme vrstice.

```
Ne prav dolgo potem se je pot na Čatež zopet krebri obrnil. Mrtoláz
pa se je menil po nemško in zmerom od pijače, kar je Dolenjčev
najljubši pogovor, ki ga ne konča tako hitro, ako se ga je polotil.
Nagovarjal je naju, da naj zložíva imenitno pesem letošnjemu vincu
na čast. »Že sam sem se prtil ž njo,« pravi, »pa mi ni po volji,
kar sem zverižil. Časi je bil Kančnik, tudi šmarski šomašter je
zaokrožil katero. Zdaj ga pa ni, da bi kaj znal. Kar sta le-ta dva
pomrla, nimamo kaj peti, stare so se pozabile, novih ni!«
                        - Fran Levstik, Popotovanje iz Litije do Čateža
```

Napiši program ali podprogram (funkcijo), ki analizira besedilo in izpiše dolžino najdaljše verige, zaporedno številko vrstice, v kateri se veriga začne, ter položaj znotraj vrstice. Če mu podamo zgornji primer, mora program izpisati 6 (dolžina verige), 2 (zaporedna številka vrstice) in 2 (položaj znotraj vrstice). Oblika izpisa ni pomembna.

Na voljo imaš naslednja podprograma (funkciji):

- NaKoncu() vrne **true**, če je program prebral vse vrstice, in **false**, če na vhodu še čakajo podatki.
- Vrstica() vrne vsebino naslednje vrstice besedila. Predpostaviš lahko, da ne bo vrstica nikoli daljša od 80 znakov. Funkcijo lahko kličeš le, če NaKoncu vrne **false**.

Teh dveh funkcij torej ne piši ti, pač pa predpostavi, da že obstajata, ti pa ju moraš uporabljati za branje vhodnega besedila. Funkciji sta takšne oblike:

```
bool NaKoncu();                    /* v C/C++ */
public static bool NaKoncu();      // v C#
public static boolean NaKoncu();   // v javi
function NaKoncu: boolean;         { v pascalu }
def NaKoncu(): ...                 # v pythonu; vrne vrednost tipa bool

void Vrstica(char *s);             /* v C/C++; kot parameter „s“ podaj kazalec na tabelo
                                      vsaj 81 znakov, funkcija pa bo vsebino naslednje vrstice
                                      skopirala vanjo */
```

```
string Vrstica();                    // v C++ (lahko uporabiš to ali prejšnjo)
public static string Vrstica();      // v C#
public static String Vrstica();      // v javi
function Vrstica: string;            { v pascalu }
def Vrstica(): ...                   # v pythonu; vrne vrednost tipa str
```

Ker je besedilo izredno dolgo, poskusi zasnovati program ali podprogram tako, da ne bo prebral celotnega besedila v pomnilnik. Rešitve, ki bodo prebrale celotno besedilo in ga šele nato analizirale, bodo dobile največ 15 točk.

Prazen prostor desno od konca vrstice (torej od zadnjega znaka tistega niza, ki ga vrne funkcija Vrstica) ne more postati del nobene verige (ne smeš se torej na primer delati, da so tam presledki ali kaj podobnega).

Za potrebe te naloge predpostavi, da vsaka vrednost tipa **char** predstavlja en znak, torej naj te ne motijo posebni znaki, npr. šumniki, ki nastopajo v zgornjem primeru.

4. Jezero

V jezeru sredi letoviškega parka želimo regulirati globino vode. Zato na izpust iz jezera namestimo računalniško krmiljeno zapornico, v najglobljo točko jezera pa postavimo tipalo za merjenje globine vode, ki vrne eno meritev na uro. Na voljo sta nam naslednji dve funkciji oz. podprograma:

- Funkcija GlobinaVode vrne višino vode kot celo število od 0 do 100 (0 = jezero se je posušilo; 100 = jezero je poplavilo letališče). Funkcija vrne rezultat šele, ko tipalo pošlje novo meritev (ta je lahko tudi enaka prejšnji meritvi). Do takrat funkcija stoji in čaka.

- Funkcija PremakniZapornico(odpri) odpre ali zapre (odvisno od tega, ali je parameter odpri enak **true** ali **false**) zapornico na izpustu iz jezera in se vrne takoj. Če je bila zapornica že od prej v takem stanju, se ne zgodi nič (to torej ne šteje za napako). Če to funkcijo kličemo večkrat zaporedoma v kratkem časovnem obdobju, obvelja zadnji klic.

Funkciji sta takšne oblike:

```
int GlobinaVode();                                  /* v C/C++ */
public static int GlobinaVode();                    /* v C# in javi */
function GlobinaVode: integer;                      { v pascalu }
def GlobinaVode(): ...                              # v pythonu; vrne vrednost tipa int

void PremakniZapornico(bool odpri);                 /* v C/C++ */
public static void PremakniZapornico(bool odpri);   // v C#
public static void PremakniZapornico(boolean odpri); // v javi
procedure PremakniZapornico(Odpri: boolean);        { v pascalu }
def PremakniZapornico(odpri): ...                   # v pythonu
```

Za 10 točk **napiši program** ali podprogram (funkcijo), ki se vrti v neskončni zanki, spremlja globino vode in odpira ali zapira zapornico po naslednjih pravilih:

- Če je globina vode pod 33, je gladina jezera prenizka. Program naj zapre zapornico, da se bo začela gladina vode dvigati.

- Če je globina vode nad 66, je gladina jezera previsoka. Program naj odpre zapornico.

Ker pa taka regulacija jezera ob velikih nalivih ali dolgotrajni suši reagira prepozno, si uprava letoviškega parka želi, da bi program za regulacijo spremljal gibanje globine vode v krajšem obdobju in v določenih primerih reagiral predčasno (še preden gladina vode postane previsoka ali prenizka).

Za dodatnih 10 točk dodaj v program naslednji dve pravili:

- Če se je v zadnjih 12 urah gladina vode stalno dvigala (vsaka meritev je strogo večja od prejšnje), naj program odpre zapornico.
- Če se je v zadnjih 12 urah gladina vode stalno spuščala (vsaka meritev je strogo manjša od prejšnje), naj program zapre zapornico.

Upoštevaj, da imata pravili iz prvega dela naloge prednost pred dodatnima pravilo-ma.

5. Stolpci in vrstice

Imamo tabelo (razpredelnico) velikosti 10 000 vrstic in nekaj manj kot 20 000 stolpcev. Vrstice so označene s številkami od 1 do 10 000, stolpci pa so označeni s črkami oz. nizi črk od A do ZZZ (A, B, ..., U, V, W, X, Y, Z, AA, AB, AC, ..., AZ, BA, ..., ZY, ZZ, AAA, AAB, ..., ZZX, ZZY, ZZZ — oznake stolpcev so torej urejene najprej po dolžini, tiste z enako dolžino pa po abecedi). Uporabljena je angleška abeceda, ki ima 26 črk; to pomeni, da ima Z vrednost 26, naslednji stolpec od Z pa je označen s črkama AA in ima vrednost 27.

V tabeli imamo nekatera polja pobarvana. Seznam teh polj imamo zapisan v kompaktni obliki tako, da si paroma sledijo podatki za stolpec in vrstico, vmes pa ni nobenih ločil ali presledkov. Na primer:

```
A1A3AA3457BB54NTL1
```

Napiši program, ki bere podatke o celicah po znakih in jih izpisuje v prijaznejši obliki tako, da sta stolpec in vrstica tabele zapisana s številko ter vsak par podatkov za stolpec in vrstico izpisana v svoji vrstici, ločena z vejico. Tvoja rešitev lahko bere s standardnega vhoda in piše na standardni izhod ali pa uporablja datoteki `vhod.txt` in `izhod.txt` (karkoli ti je lažje). Predpostaviš lahko, da je vhodni niz dolg največ 1000 znakov.

Za gornji primer je pravilen izpis takšen:

```
1, 1
1, 3
27, 3457
54, 54
9996, 1
```

Pozor: tvoja rešitev mora delovati za poljubne vhodne podatke, ki so v skladu z zgoraj opisanimi pravili, ne samo za gornji primer.

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 prek računalnika. 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 na približno dve minuti samodejno shrani. 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 želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku (uporabi ikono „Urejevalnik teksta“ na namizju). **Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave in želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici.**

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. Anagramska razdalja

Dana sta dva niza, s in t, sestavljena le iz malih črk angleške abecede (od `a` do `z`). Niz s bi radi predelali tako, da bi iz njega nastal poljuben anagram t-ja, torej poljuben tak niz, ki ga je mogoče dobiti iz t tako, da v njem premešamo vrstni red črk. Pri tem smemo uporabljati naslednje osnovne operacije:

- Dodajanje črke: po eno črko naenkrat lahko vrinemo na enem mestu kjerkoli v nizu, tudi na začetku ali na koncu. Primeri: `sol` → `stol`; ali pa `tal` → `stal`; ali pa `cel` → `celo`.
- Brisanje črke: to je ravno obratna operacija od prejšnje. Kjerkoli v nizu lahko pobrišemo po eno pojavitev ene črke naenkrat; na primer: `steze` → `stez`; ali pa `steze` → `teze`; ali pa `otrok` → `otok`.
- Sprememba črke: po eno pojavitev ene črke lahko spremenimo v poljubno drugo črko; na primer: `teta` → `meta`; ali pa `teta` → `trta`; ali pa `teta` → `tete`.

Napiši podprogram (funkcijo) `AnagramskaRazdalja(s, t)`, ki kot parametra dobi niza s in t in vrne najmanjše število teh osnovnih operacij, s katerim je mogoče iz s dobiti kakšen tak niz, ki je anagram niza t.

Primer: `AnagramskaRazdalja("stol", "volt")` mora vrniti 1. Z eno operacijo lahko iz `stol` dobimo `vtol`, to pa je anagram besede `volt`.

`AnagramskaRazdalja("arbalest", "balasta")` mora vrniti 2. Primerno zaporedje dveh operacij (ni pa edino tako) je `arbalest` → `aabalest` → `aabalst`, slednje pa je anagram niza `balasta`.

2. Zaboji

V skladišču stoji v vrsti n zabojev, ki so oštevilčeni s števili od 1 do n (nobena dva zaboja nimata enake številke), vendar v nekem premešanem vrstnem redu. Na voljo imamo tri operacije, ki jih za nas izvaja sistem robotskih rok v skladišču:

- zamakni vse zaboje ciklično za eno mesto v levo (pri tem torej zaboj, ki je bil prej najbolj levi, postane najbolj desni);
- zamakni vse zaboje ciklično za eno mesto v desno (pri tem torej zaboj, ki je bil prej najbolj desni, postane najbolj levi);
- poberi zadnji zaboj (najbolj desnega) in ga pošlji iz skladišča (npr. v proizvodnjo).

Radi bi s čim manj operacijami pobrali vse zaboje iz skladišča in to v naraščajočem vrstnem redu; najprej hočemo torej dobiti zaboj številka 1, nato zaboj številka 2 in tako naprej, nazadnje pa zaboj številka n. **Opiši postopek** (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki kot vhodne podatke dobi začetni vrstni red zabojev v skladišču in izračuna najmanjše število zgoraj omenjenih operacij, s katerim lahko poberemo vse zaboje iz skladišča v naraščajočem vrstnem redu.

Zaželeno je, da je tvoj postopek čimbolj učinkovit, da bo hitro izračunal potrebno število operacij tudi pri velikih n (npr. več deset tisoč zabojev).

Primer: recimo, da je začetno zaporedje zabojev (od leve proti desni) $4, 1, 3, 5, 2$. Najkrajše zaporedje operacij, s katerim lahko dobimo vse zaboje iz skladišča v naraščajočem vrstnem redu, je potem takšno:

Operacija	Stanje po njej
(*začetno stanje*)	4, 1, 3, 5, 2
zamik v levo	1, 3, 5, 2, 4
zamik v levo	3, 5, 2, 4, 1
poberi zadnji zaboj	3, 5, 2, 4
zamik v desno	4, 3, 5, 2
poberi zadnji zaboj	4, 3, 5
zamik v desno	5, 4, 3
poberi zadnji zaboj	5, 4
poberi zadnji zaboj	5
poberi zadnji zaboj	(*prazno*)

Odgovor, ki ga mora v tem primeru izračunati tvoja rešitev, je torej ta, da potrebujemo 9 operacij. **Pozor:** tvoja rešitev mora delovati za skladišča s poljubnim številom zabojev, ne le s točno petimi zaboji, kolikor jih je pri tem primeru.

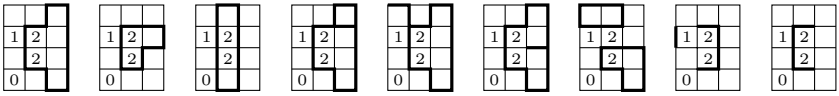
3. Ograje

Dana je pravokotna karirasta mreža, ki ima h vrstic in w stolpcev. Posamezno polje mreže ima obliko kvadrata in je lahko prazno ali pa je v njem eno od števil 0, 1, 2 ali 3. Na stranice, kjer se stikata dve polji ali pa kjer polje meji na zunanjost mreže, so ponekod postavljene ograje. (Ograja, če je prisotna na neki stranici, pokriva to stranico v celoti, od oglišča do oglišča, ne pa na primer le delno.)

Radi bi preverili, če so ograje postavljene v skladu z naslednjimi pravili:

1. Število v polju pove, koliko stranic tega polja mora biti ograjenih.
2. Če je polje prazno, ni predpisano, koliko stranic tega polja mora biti ograjenih (če sploh katera).
3. Veriga ograj mora tvoriti en sam cikel, ki ne sme biti nikjer prekinjen ali razvejen in se mora zaključiti sam vase.
4. V mreži mora biti prisotna vsaj ena ograja.

Naslednja slika kaže nekaj primerov; debele črte predstavljajo ograje. Na mreži (a) so ograje postavljene v skladu z vsemi gornjimi pravili, na ostalih mrežah pa ne. Pri (b) je narobe to, da je eno od polj s številom 2 ograjeno s tremi ograjami, moralo pa bi biti z dvema. Podobno je pri (c) polje s številom 0 ograjeno z eno ograjo, ne bi pa smelo biti z nobeno. Pri ostalih primerih ograja ne tvori enega samega cikla, ampak je na razne načine razvejena ali pa tvori več ločenih ciklov, sploh ni sklenjena v cikel in podobno.



Opiši podatkovne strukture, s katerimi bi lahko v svojem programu predstavil stanje mreže (kar vključuje tako števila na poljih kot položaj ograj), nato pa **napiši program** (ali opiši postopek, če ti je lažje), ki kot vhod dobi w, h in stanje mreže (v takšnih podatkovnih strukturah, kot si jih prej opisal) in preveri, ali so ograje postavljene v skladu z zgoraj opisanimi pravili. Tvoj postopek mora delovati za mreže poljubne velikosti, ne le takšne, karkšne so prikazane na gornji sliki.

4. Past za žvižgače

Komisija za nadzor obveščevalnih služb je ugotovila, da je prejšnja leta njihovo zaupno poročilo pricurljalo v javnost, čeprav so vsi prejemniki poročila trdili, da je njihova kopija ostala varno pri njih.

Da bi odkrili, kdo izdaja zaupne dokumente, so letos sklenili, da bodo poročilo na nekaj nevpadljivih mestih rahlo spremenili in pripravili 32 kopij, ki se vse med

seboj razlikujejo. Če bo kakšna od teh kopij potem pricurljala v javnost, se bo dalo ugotoviti, katera kopija je to bila.

Ker sta besedi „in“ in „ter“ pomensko precej podobni in tudi dovolj pogosti v besedilu, je načrt takšen: v besedilu najdemo prvih 5 pojavitev ene ali druge besede (katerekoli od obeh; upoštevamo le tiste, ki so zapisane samo z malimi črkami) in na teh petih mestih namesto teh pojavitev vstavimo primerno kombinacijo besed „in“ in „ter“ tako, da bo v vsaki od 32 kopij dokumenta kombinacija drugačna (in si nekako zabeležimo, komu smo katero kopijo izročili — a to ni del naloge).

Napiši program, ki bo najprej prebral celo število med 1 in 32, potem pa besedilo v preostanku vhodne datoteke prepisal na izhod in ga pri tem spremenil tako, kot je opisano v prejšnjem odstavku. Pri tem naj kombinacija izbranih besed „in“ in „ter“ enolično ustreza prebrani številki kopije. Naloga ne predpisuje točno, kako naj kombinacija ustreza številki kopije; to izberi sam in **pojasni** svojo izbrano preslikavo.

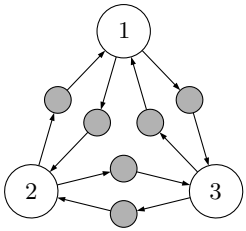
Predpostavimo lahko, da se v besedilu nahaja vsaj pet besed „in“ ali „ter“ (npr.: „... in ... in ... ter ... in ... ter ...“). Lahko tudi predpostavimo, da vrstice niso daljše od 100 znakov. Število vrstic dokumenta ni omejeno. Da ne zapletamo po nepotrebnem, lahko predpostaviš, da besede med seboj loči natanko en presledek ali pa meja med vrsticami. Druga ločila, posebni znaki ali številke v besedilu ne nastopajo.

5. Pekarna

Pekarna Križkraž se je odločila avtomatizirati svoj sistem za peko kruha. Ker gre za kritičen sistem, so se odločili, da bodo namesto enega računalnika uporabili tri. Celoten sistem enkrat na leto ugasnejo, da spihajo moko iz napajalnikov, v vmesnem času morajo računalniki neprekinjeno delovati.

Ker ima vsaka programska oprema napake, so računalnike povezali tako, da vsak nadzoruje druga dva. Če se začne eden od računalnikov napačno obnašati, ga druga dva lahko ugasneta. Vzdrževalec sistema bo potem našel razlog za napako ter računalnik prižgal nazaj.

Vsak računalnik ima dva napajalnika, vsakega od njiju pa nadzoruje po eden od preostalih dveh računalnikov. Računalnika 2 in 3 kontrolirata vsak po en napajalnik računalnika 1; računalnika 1 in 3 kontrolirata vsak po en napajalnik računalnika 2; računalnika 1 in 2 pa kontrolirata vsak po en napajalnik računalnika 3. Na naslednji sliki beli krogi predstavljajo računalnike, sivi pa napajalnike:



Vsak računalnik deluje, če ima vključen vsaj en napajalnik, in je ugasnjen, če sta ugasnjena oba.

Računalniki so povezani z zanesljivim komunikacijskim omrežjem. Na vsakem je pognan program Vahtar, ki predstavlja jedro nadzornega sistema. Program sprejema sporočila (izzive), iz vsakega sporočila po neki funkciji izračuna odgovor in ga vrne računalniku, ki je poslal izziv.

Nadzorni sistem deluje tako, da vsak računalnik enkrat na sekundo pošlje izziv drugima dvema, sprejema odgovore, ki jih vrača program Vahtar, preverja odgovore in po potrebi ugasne napajalnik. No, vsaj delal naj bi tako. Celoten sistem izziv-odgovor (challenge-response) je zasnoval pogodbeni programer, ki je zatem odšel boljšim poslom nasproti in ni dokončal svojega dela. Manjka program, ki bo pošiljal izzive, preverjal odgovore in ugašal napajalnike.

Napiši program, ki bo tekel v vsakem računalniku in ugašal napajalnike računalnikov, ki se ne odzivajo ali se odzivajo z veliko zamudo.

Na razpolago imaš funkcije:

- Pocakaj() — počaka, da nastopi naslednja sekunda, in nato vrne čas od zagona programa v sekundah (kot celo število);
- KdoSem() — vrne številko računalnika, na katerem teče program (1, 2 ali 3);
- Vprasanje(X, msg) — pošlje sporočilo (izziv) msg računalniku X. X je lahko 1, 2 ali 3. Sporočilo je poljubno celo število. Sporočila ne moreš poslati sam sebi — tako na primer klic Vprasanje(KdoSem(), msg) velja za napako.
- Odgovor(X) vrne odgovor računalnika X. X je lahko 1, 2 ali 3. Rezultat funkcije je 0, če sistem X ni poslal nobenega sporočila, oziroma najstarejše sporočilo (celo število), ki čaka na obdelavo. Odgovora od samega sebe ne moreš dobiti (Odgovor(KdoSem()) vedno vrne 0).
- UgasniNapajalnik(X) — ugasne napajalnik računalnika X, ki mora biti eden od tistih dveh, ki ju nadzoruje računalnik, na katerem je pognan program. Če ugasnemo že ugasnjen napajalnik, ne bo nobene škode.

Program Vahtar vsak izziv pomnoži z 2 in vrne dobljeni zmnožek. Če naš program na primer pokliče Vprasanje(3, 21), bo funkcija Odgovor(3) prej ali slej vrnila 42. Dokler računalnik deluje pravilno, seveda...

Izkazalo se je, da lahko pride do dveh vrst napak.

- Program Vahtar začne včasih delovati izredno počasi. Včasih se situacija popravi sama od sebe, ob daljšem počasnem delovanju pa želimo računalnik izklopiti. Kadar Vahtar deluje počasi, tudi ne odgovarja pravočasno na izzive. Zato se lahko zgodi, da od računalnika nekaj sekund ne dobimo nobenega odgovora (funkcija Odgovor vztrajno vrača 0), nato pa v eni sekundi dobimo več odgovorov.
- Vahtar lahko včasih obvisi in odtlej na vsa nadaljnja vprašanja vrača enak odgovor kot na zadnje vprašanje, ki ga je dobil, preden je obvisel.

Drugačnih napak Vahtar ne dela. Na vsak izziv vedno odgovori pravilno, odgovor pa — kadar deluje normalno hitro — vrne praktično takoj. Čas obdelave izziva lahko zanemariš.

Tudi s komunikacijskim kanalom ni težav. Celo če računalnik pošlje več deset sporočil v hitrem zaporedju in jih ciljni računalnik ne obdela dovolj hitro, bodo sporočila počakala na obdelavo. Prav tako ni pri komunikaciji nobenih napak — sporočila se pri prenosu ne izgubljajo in ne kvarijo. Sporočila vedno dobimo v enakem vrstnem redu, kot so bila poslana.

Tvoj program naj enkrat na sekundo obema drugima računalnikoma pošlje izziv. Sprejema naj odgovore (tudi če jih v eni sekundi dobi veliko) in ugasne napajalnik oddaljenega računalnika v dveh primerih:

- kadar od računalnika več kot 10 sekund ne dobi nobenega odgovora;
- kadar od računalnika dobi odgovor na izziv, ki je bil poslan pred več kot 10 sekundami.

Če tvoj postopek pripelje do stanja, v katerem se ugasnejo vsi trije računalniki, naj te to ne moti. S tem se bo ukvarjal naslednji pogodbeni programer.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše. Programi naj berejo vhodne podatke s standardnega vhoda in izpisujejo svoje rezultate na standardni izhod. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih podatkov. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih podatkov, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih podatkov.

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.

Tvoji programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali python, mi pa jih bomo preverili s prevajalniki FreePascal, GNUjevima `gcc` in `g++` 5.4.0 (ta verzija podpira C++14, novejše različice standarda C++ pa le delno), prevajalnikom za javo iz JDK 8, s prevajalnikom Mono 4.2 za C# in z interpreterjema za python 2.7 in 3.6. Za delo lahko uporabiš CodeBlocks, Visual Studio Code, Eclipse, IntelliJ IDEA, PyCharm, prevajalnike v ukazni vrstici in druga orodja, ki so na voljo na namizju oz. v meniju.

Na spletni strani `https://putka-rtk.acm.si/competitions/rtk_2019_3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Pred začetkom tekmovanja lahko poskusiš oddati katero od nalog iz arhiva `https://putka-rtk.acm.si/tasks/test_sistema/`. Uporabniška imena in gesla za Putko so enaka kot za računalnike.

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih (praviloma desetih). Za vsak testni primer se bo izpisalo, 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 preveč pomnilnika (več kot 250 MB), ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

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

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal

pravilen odgovor, sicer pa 0 točk. Izjema je druga naloga, kjer je testnih primerov 20 in za pravilen odgovor pri posameznem testnem primeru dobiš 5 točk.

Nato se točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N-1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

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

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

Primer naloge (ne šteje k tekmovanju)

Napiši program, ki s standardnega vhoda prebere dve celi števili (obe sta v prvi vrstici, ločeni z enim presledkom) in izpiše desetkratnik njune vsote na standardni izhod.

Primer vhoda:

```
123 456
```

Ustrezen izhod:

```
5790
```

Primeri rešitev:

- V pascalu:

```
program PoskusnaNaloga;
var i, j: integer;
begin
  ReadLn(i, j);
  WriteLn(10 * (i + j));
end. {PoskusnaNaloga}
```

- V C++:

```
#include <iostream>
using namespace std;

int main()
{
  int i, j; cin >> i >> j;
  cout << 10 * (i + j) << '\n';
}
```

- V C-ju:

```
#include <stdio.h>

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

- V pythonu:

```
import sys
L = sys.stdin.readline().split()
i = int(L[0]); j = int(L[1])
print("%d" % (10 * (i + j)))
```

(*Opomba:* namesto `'\n'` lahko uporabimo `endl`, vendar je slednje ponavadi počasneje.)

(*Primeri rešitev se nadaljujejo na naslednji strani.*)

- V javi:

```
import java.io.*;
import java.util.Scanner;

public class Poskus
{
   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(10 * (i + j));
   }
}
```

- V C#:

```
using System;

class Program
{
   static void Main(string[] args)
   {
      string[] t = Console.In.ReadLine().Split(' ');
      int i = int.Parse(t[0]), j = int.Parse(t[1]);
      Console.Out.WriteLine("{0}", 10 * (i + j));
   }
}
```

NALOGE ZA TRETJO SKUPINO

1. Fitnes

Milan bi rad odprl svoj fitnes, vendar pri tem potrebuje tvojo pomoč. Sam namreč ne ve, koliko opreme mora kupiti. Ne bi rad kupil preveč opreme, saj bi s tem porabil preveč denarja, prav tako pa ne bi rad kupil premalo opreme, saj bi to pomenilo, da bodo njegove stranke nezadovoljne. Milanov fitnes bodo obiskovali sami zavzeti fitneserji, ki bodo prihajali v fitnes vsak dan. Milan je izvedel anketo med njimi in izvedel, ob katerem času v dnevu bodo v fitnesu in katero napravo bodo uporabljali.

Če bo obiskovalec prišel v fitnes in njegova želena naprava ne bo na voljo, bo jezen zapustil fitnes in se ne bo nikoli več vrnil. Milan si tega ne želi in bo, če bo treba, kupil po več naprav enakega tipa, da jih bo lahko hkrati uporabljalo več obiskovalcev. **Napiši program**, ki obdela rezultate ankete in izpiše minimalno število naprav vsakega tipa, ki jih mora kupiti.

Predpostavi, da se lahko obiskovalca na posamezni napravi izmenjata v trenutku (če torej drugi pride ob istem času, ob katerem prvi odide, lahko oba uporabljata isto napravo).

Vhodni podatki: v prvi vrstici bo celo število k ($1 \leq k \leq 10^5$) — število obiskovalcev fitnesa. Sledi k vrstic, ki opisujejo vsaka po enega obiskovalca. Vsaka od teh vrstic vsebuje po 3 cela števila, ločena s po enim presledkom; za i-tega obiskovalca so ta števila po vrsti naslednja: čas prihoda p_i, čas odhoda o_i in število h_i, ki predstavlja identifikacijsko številko tipa naprave, ki jo bo uporabljal (velja $0 \leq h_i \leq 10^9$). Časi so podani v desetinkah sekunde od polnoči; velja $0 \leq p_i < o_i < 24 \cdot 60 \cdot 60 \cdot 10$. Vsi ti podatki so zbrani znotraj enega dneva; če bo obiskovalec na neki dan prišel v fitnes, ga bo še isti dan tudi zapustil. Pri 50 % testnih primerov bo $k \leq 1000$.

Izhodni podatki: izpiši po eno vrstico za vsak tip naprave, ki se pojavlja v vhodnih podatkih, ta vrstica pa naj vsebuje dve celi števili, ločeni s po enim presledkom: najprej identifikacijsko številko tipa naprave, nato pa minimalno število naprav tega tipa, ki jih mora Milan kupiti. V izpisu naj bodo naprave urejene naraščajoče po identifikacijski številki.

Primer vhoda:

```
4
333 433 12345
500 777 998877
400 420 998877
350 440 12345
```

Pripadajoči izhod:

```
12345 2
998877 1
```

Razlaga gornjega primera (besede „prvi“, „drugi“ itd. v spodnji razlagi se nanašajo na vrstni red, v katerem obiskovalci pridejo v fitnes, ne na vrstni red v vhodnih podatkih):

- Prvi obiskovalec pride v fitnes ob času 333 in zasede eno napravo tipa 12345.
- Drugi obiskovalec pride v fitnes ob času 350 in zasede drugo napravo tipa 12345.
- Tretji prispe ob času 400, telovadi na napravi tipa 998877 in ob času 420 odide.
- Prvi in drugi obiskovalec končata (prvi ob času 433, drugi ob 440) in odideta.

- Četrti obiskovalec prispe ob času 500 in uporablja napravo tipa 998877 do časa 777.

Da ne bi razjezili nobenega obiskovalca, bi potrebovali dve napravi tipa 12345 in eno napravo tipa 9988877.

2. Telefonsko omrežje

V nekem podjetju so želeli postaviti novo telefonsko omrežje in zato so po zemljevidu na nekaterih lokacijah postavili različno močne oddajnike signala. Omrežje so vzpostavili, sedaj pa jih zanima, koliko polj je pokritih s telefonskim signalom.

Zemljevid je ogromna celoštevilska kvadratna mreža, ki jo tvorijo celice (x, y) za $-10^8 < x < 10^8$ in $-10^8 < y < 10^8$. Oddajnik z močjo r, ki je postavljen na polju (a, b), bo oddajal signal do vseh polj (x, y), za katera velja

$$|x - a| + |y - b| \leq r.$$

Napiši program, ki bo iz podatkov o položaju in moči oddajnikov izračunal, koliko polj v mreži ima telefonski signal. Pri tem seveda vsako pokrito polje šteje le enkrat, četudi ga pokriva signal več oddajnikov.

Vhodni podatki: v prvi vrstici je število oddajnikov k. Sledi k vrstic, ki opisujejo vsaka po en oddajnik. Vsaka od teh vrstic vsebuje tri cela števila, a_i (x-koordinata i-tega oddajnika), b_i (y-koordinata i-tega oddajnika) in r_i (moč i-tega oddajnika), ločena s po enim presledkom.

Omejitve: pri tej nalogi je 20 testnih primerov.

- Pri prvih 7 primerih je $1 \leq k \leq 100$, $1 \leq r_i \leq 100$, $|a_i| < 10^5$, $|b_i| < 10^5$.
- Pri naslednjih 6 primerih je $1 \leq k \leq 300$, $1 \leq r_i \leq 1000$, $|a_i| < 10^5$, $|b_i| < 10^5$.
- Pri zadnjih 7 primerih je $1 \leq k \leq 1000$, $1 \leq r_i \leq 1000$, $|a_i| < 10^7$, $|b_i| < 10^7$.

Izhodni podatki: izpiši število polj, ki jih pokriva signal vsaj enega oddajnika.

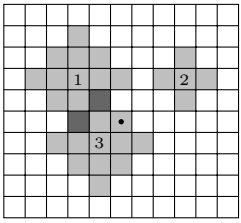
Primer vhoda:

```
3
-2 2 2
3 2 1
-1 -1 2
```

Pripadajoči izhod:

```
29
```

Ilustracija tega primera:



Komentar: pri gornjem primeru imamo tri oddajnike, dva z močjo 2 in enega z močjo 1. Temnejši odtenek sive pri oddajnikih 1 in 3 označuje polji, kjer se signala obeh oddajnikov prekrivata. Črna pika na sliki označuje koordinatno izhodišče, torej polje $(0, 0)$. Če je celotno naše telefonsko omrežje sestavljeno samo iz teh treh oddajnikov, ima telefonski signal 29 polj.

3. Transakcijski računi

Že dlje časa nadzoruješ aktivnosti lokalne kriminalne organizacije in imaš bazo številk bančnih računov, s katerimi pogosto poslujejo. Večinoma so to računi članov

organizacije, občasno pa za izboljšanje javnega ugleda kaj denarja nakažejo tudi določeni dobrodelni ustanovi. Prestregel si seznam nakazil, ki jih bodo izvedli naslednji dan. Vsako nakazilo je sestavljeno iz številke računa r (to je vedno eden od računov iz baze) in zneska z ter pomeni, da bodo na račun r plačali z enot denarja.

Vsak račun se začne s črko A, ki ji sledi devetmestna identifikacijska številka. Tej številki sledi še kontrolna številka, ki jo dobimo tako, da vzamemo zadnjo številko vsote številk identifikacijske številke. V seznamu nakazil bi rad zamenjal čim več prejemnikov denarja, tako da bi bil namesto njim denar nakazan dobrodelni ustanovi. Toda ne moreš kar prosto zamenjati računov v seznamu, saj je tudi na podlagi celega seznama izračunana kontrolna številka. Ta je izračunana tako, da vzamemo kontrolne številke vseh računov, jih seštejemo in zadnjo številko vsote proglasimo za kontrolno številko celega seznama. Prejemnike lahko zamenjaš samo z drugimi računi iz baze, pa še to samo tako, da kontrolna številka celega seznama po menjavah ostane nespremenjena. **Napiši program**, ki izračuna največjo možno skupno vsoto denarja, ki jo lahko pri tako spremenjenem seznamu nakazil prejme dobrodelna ustanova.

Vhodni podatki: v prvi vrstici sta s presledkom ločeni celi števili n in m. Sledi n vrstic s številkami računov (baza računov, povezanih z organizacijo). Prvi izmed teh n računov je račun dobrodelne ustanove. Nato sledi še m vrstic s seznamom nakazil. Vsako nakazilo je sestavljeno iz številke računa in celoštevilskega zneska, ki sta ločena s presledkom.

Izhodni podatki: izpiši eno samo celo število, namreč največji možni znesek, ki ga lahko dobi dobrodelna ustanova, če račune iz seznama nakazil zamenjaš s poljubnimi drugimi računi iz baze tako, da je kontrolna številka seznama še vedno taka kot pred spremembami.

Omejitve. Vedno velja $1 \leq n \leq 10^5$, $1 \leq m \leq 10^5$, $1 \leq z \leq 10^6$, pri nekaterih testnih primerih pa veljajo še dodatne omejitve:

- Pri prvih 10 % testnih primerov je $n \leq 10$, $m \leq 10$ in $z \leq 10$.
- Pri naslednjih 20 % testnih primerov je $n \leq 1000$, $m \leq 1000$ in $z \leq 1000$.
- Pri naslednjih 20 % testnih primerov je $z \leq 10^4$.

Primer vhoda:

```
5 5
A6666666664
A2140319954
A1000000056
A0050050000
A0050050088
A1000000056 100000
A2140319954 750000
A6666666664 1001
A1000000056 60000
A0050050088 250000
```

Pripadajoči izhod:

```
1100000
```

Komentar: račun dobrodelne ustanove je A6666666664. Na začetku je kontrolna številka seznama nakazil enaka 8. Ena izmed optimalnih rešitev je, da nakazilom za 2 500 000, 750 000 in 100 000 enot denarja spremenimo račun na A6666666664, nakazilo za 1 001 preusmerimo na A0050050000, nakazilo za 60 000 pa pustimo pri miru. Tako je kontrolna vsota seznama zopet enaka 8.

Če bi katerokoli kombinacijo štirih nakazil poskusili preusmeriti na A6666666664,

bi ugotovili, da prejemnika preostalega nakazila ne moremo zamenjati tako, da bi se kontrolna številka celotnega seznama ujemala s prvotno.

4. Nadzor

Na domači ulici je prišlo do porasta kriminalnih aktivnosti. Zato ste se s sosedi odločili za investicijo v kamere, s katerimi boste nadzorovali dogajanje vzdolž celotne ulice. Analizirali ste možne lokacije kamer in ponudbo izdelkov na trgu ter prišli do seznama možnih postavitev. Sedaj pa se morate odločiti za cenovno najugodnejši izbor, s katerim boste lahko nadzorovali celotno ulico.

Ulico lahko predstavimo z daljico dolžine d. Položaj poljubne točke na daljici lahko potem opišemo z njeno x-koordinato, ki pove oddaljenost te točke od levega krajišča daljice. Izbiramo med n kamerami, med katerimi i-ta kamera s ceno c_i pokriva na daljici interval točk z x-koordinatami od vključno a_i do vključno b_i. **Napiši program**, ki izračuna najnižjo ceno takega izbora kamer, pri katerem bo unija pripadajočih intervalov vsebovala celotno ulico.

Vhodni podatki: v prvi vrstici sta dve celi števili, d (dolžina ulice, $1 \leq d \leq 10^9$) in n (število kamer, $1 \leq n \leq 10^6$), ločeni z enim presledkom. V naslednjih n vrsticah so opisane posamezne kamere; i-ta od teh vrstic vsebuje cela števila a_i, b_i in c_i, ločena s po enim presledkom (velja $0 \leq a_i < b_i \leq d$ in $1 \leq c_i \leq 10^9$).

Naloga vsebuje deset testnih primerov. Pri prvih dveh bo veljalo $n \leq 20$. Pri prvih petih bo veljalo $n \leq 10\,000$. Pri šestem in sedmem testnem primeru bo veljalo $c_i = 1$ (za vse kamere).

Izhodni podatki: izpiši eno samo celo število, in sicer iskano skupno ceno kamer. Testni primeri bodo sestavljeni tako, da primeren nabor kamer zagotovo obstaja.

Primer vhoda:

```
6 7
3 5 3
1 2 2
0 3 2
4 6 3
0 6 10
5 6 4
0 3 1
```

Pripadajoči izhod:

```
7
```

Komentar: pri tem primeru lahko z izbiro sedme, prve in četrte kamere pokrijemo območja $[0, 3]$, $[3, 5]$ in $[4, 6]$ za ceno $1+3+3 = 7$.

5. Detektorji

Štef je varnostnik v banki, ki je sestavljena iz n trezorjev, ki so povezani z m hodniki (vsak hodnik neposredno povezuje natanko dva trezorja). Banka je opremljena z d detektorji, vsak izmed njih pokriva nekaj trezorjev in (mogoče) zazna, če je kak človek prisoten v kakšnem izmed teh trezorjev.

Neko noč je Štef delal v nočni izmeni, vendar je zaradi napornega dne zaspal. Ko se je zjutraj zbudil, je ugotovil, da so banko oropali. Preden Štef slabo novico sporoči nadrejenim, ga zanima, največ koliko denarja je lopov lahko ukradel.

Noč je razdeljena na q časovnih intervalov. Lopov se v vsakem intervalu nahaja v natanko enem izmed trezorjev (lahko je v istem trezorju več intervalov in ti intervali tudi niso nujno zaporedni). Na prehodu iz enega intervala v naslednjega lahko lopov bodisi ostane v dotedanjem trezorju ali pa se iz njega hipoma premakne v neki drug trezor po eni od povezav, ki njegov dotedanji trezor neposredno povezujejo z drugimi

trezorji. Lopov je v banko lahko prišel in odšel iz poljubnega trezorja. Če lopov neki časovni interval prebije v trezorju številka i, ukrade iz njega v tem intervalu w_i enot denarja. Lopov je v banki prisoten vso noč, od prvega do zadnjega intervala. V vsakem trezorju je toliko denarja, da ga lopov ne more izprazniti niti, če v njem preživi vso noč.

Štef za vsak časovni interval dobi podatke o tem, kateri detektorji so bili v tem intervalu v drugačnem stanju kot v prejšnjem intervalu (znotraj posameznega intervala pa se stanje detektorjev ne spreminja). S tem, da se je stanje detektorja spremenilo, hočemo reči, da je bodisi v prejšnjem časovnem intervalu zaznaval prisotnost lopova v trezorjih, ki jih pokriva, v trenutnem intervalu pa ne, ali pa v prejšnjem intervalu ni zaznaval prisotnosti, v trenutnem pa jo. Na začetku (pred prvim intervalom) ni noben detektor zaznaval prisotnosti.

Detektorji ne delujejo povsem zanesljivo, zato se Štef drži naslednjega pravila: če vsaj polovica detektorjev, ki pokrivajo določen trezor, zaznava prisotnost, potem Štef sklepa, da je lopov mogoče bil prisoten v tem trezorju, sicer pa, da ga v njem gotovo ni bilo.

Iz podatkov o stanju detektorjev skozi čas ni nujno mogoče enolično določiti, kako se je lopov premikal med trezorji oz. kdaj se je zadrževal kje. Lahko se zgodi, da obstaja več možnih poti lopova, ki so skladne z vhodnimi podatki (o hodnikih in o stanju detektorjev), se pa mogoče razlikujejo po skupni vsoti nakradenega denarja. Med temi možnimi vsotami Štefa zanima največja; **napiši program**, ki jo izračuna.

Vhodni podatki: sestavljajo jih sama cela števila in kjer jih je po več v eni vrstici, so ločena s po enim presledkom. V prvi vrstici so n (število trezorjev), m (število hodnikov), d (število detektorjev) in q (število časovnih intervalov).

Sledi m vrstic, ki opisujejo hodnike; i-ta od teh vrstic vsebuje števili a_i in b_i, ki povesta, da i-ti hodnik povezuje trezorja a_i in b_i (veljalo bo $1 \leq a_i < b_i \leq n$). Po takem hodniku gre lahko lopov tako iz a_i v b_i kot tudi iz b_i v a_i. Vsi hodniki so med seboj različni (torej je posamezni par trezorjev lahko neposredno povezan z največ enim hodnikom).

Sledi n vrstic, ki opisujejo trezorje; i-ta od teh vrstic vsebuje števila w_i, s_i in e_i. Ta povedo, da i-ti trezor pokrivajo detektorji od vključno s_i do vključno e_i (veljalo bo $1 \leq s_i \leq e_i \leq d$) in da lopovi v vsakem časovnem intervalu, ki ga prebijejo v tem trezorju, nakradejo w_i enot denarja. Posamezni trezor torej vedno pokrivajo zaporedni detektorji.

Sledi še q vrstic, ki po vrsti opisujejo časovne intervale; i-ta od teh vrstic vsebuje najprej število t_i, ki pove, koliko detektorjem se je v tem intervalu spremenilo stanje v primerjavi s prejšnjim časovnim intervalom; nato pa sledijo še številke teh detektorjev, podane v strogo naraščajočem vrstnem redu.

Omejitve: povsod bo veljalo $1 \leq n \leq 1000$, $1 \leq m \leq 10^5$, $1 \leq d \leq 10^4$, $1 \leq q \leq 100$ in (za vsak i) $1 \leq w_i \leq 10^5$. Pri prvih 20 % testnih primerov bo $n \leq 10$, $d \leq 100$ in $q \leq 10$. Pri naslednjih 30 % testnih primerov bo $n \leq 100$ in $d \leq 1000$.

Izhodni podatki: izpiši eno samo celo število, in sicer največjo možno vsoto ukradenega denarja, ki je še skladna z vhodnimi podatki. Testni primeri so sestavljeni tako, da rešitev zagotovo obstaja.

Primer vhoda:

```
4 3 3 3
1 2
2 3
2 4
8 1 1
12 1 3
10 2 3
15 3 3
2 1 2
2 1 3
1 3
```

Pripadajoči izhod:

```
34
```

Komentar: do 34 enot plena lahko lopov pride tako, da prva dva časovna intervala prebije v trezorju številka 2, tretjega pa v trezorju številka 3. Več kot toliko plena ne more dobiti na način, ki bi bil skladen z vhodnimi podatki.

NALOGE ZA ŠOLSKO TEKMOVANJE

18. januarja 2019

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. e-knjiga

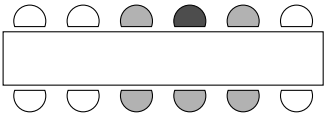
Napiši program, ki prebere besedilo in prešteje, koliko znakov angleške abecede vsebuje besedilo in koliko samoglasnikov. Program lahko bere s standardnega vhoda ali pa iz datoteke `besedilo.txt` (karkoli ti je lažje).

Program naj torej izpiše število znakov med `'a'` in `'z'` ter `'A'` in `'Z'` ter število samoglasnikov (za potrebe te naloge bomo za samoglasnike šteli znake `'A'`, `'a'`, `'E'`, `'e'`, `'I'`, `'i'`, `'O'`, `'o'`, `'U'` in `'u'`). Poleg teh znakov se lahko v vhodnem besedilu pojavljajo še poljubni drugi znaki, ki pa naj jih tvoj program ne šteje.

2. Večerja Franca Jožefa

Na dvoru pri Francu Jožefu so imeli n miz različnih dolžin; te dolžine so podane in jih označimo z $d_1, d_2, \ldots, d_n$. Vsaka miza ima obliko podolgovatega pravokotnika (velikosti $2 \times d_i$), tako da ima vsak obiskovalec sosede levo, desno, nasproti in po diagonalah (razen če sedi pri enem od vogalov — takrat nekaterih sosedov pač nima).

Primer: naslednja slika kaže mizo dolžine $d_i = 6$. Na vsaki strani mize torej sedi šest gostov. Gost, ki je na sliki pobarvan temno sivo, ima pet sosedov, in sicer tiste goste, ki so pobarvani svetlo sivo.



Na pomembni večerji so vse mize polne gostov. Stara natakarica je zelo počasna (in je zgolj ena) in pogosto se zgodi, da so na eni strani omizja hrano že pojedli, na drugi strani pa je sploh še niso dobili. Zato naredi načrt deljenja hrane tako, da bodo ljudje začeli jesti svoj obrok čim kasneje. Vsi gostje se držijo bontona, ki veli, da je nevljudno jesti, dokler nimajo vsi okrog tebe (tudi tisti po diagonalah) svojega krožnika s hrano.

Opiši postopek (ali napiši program ali podprogram, če ti je lažje), ki ugotovi, koliko krožnikov s hrano lahko natakarica razdeli, ne da bi katerikoli od gostov začel jesti, če jih deli optimalno. **Dobro utemelji**, zakaj so rezultati, ki jih vrača tvoj postopek, pravilni. Kot vhodne podatke tvoj postopek dobi števila $n, d_1, d_2, \ldots, d_{n-1}$ in d_n.

3. Robot

V podjetju, ki izdeluje igračke, so se odločili, da bodo pričeli izdelovati novo serijo robotov, ki bodo lahko peli, govorili, jokali, reševali različne naloge ter počeli še vrsto drugih zanimivih reči. Vsekakor najpomembnejša značilnost teh robotov pa je njihovo premikanje. Roboti nove serije se bodo lahko premikali le naprej ter se sukali okrog svoje osi za 90°. Zaradi boljše orientacije v prostoru je za vsakega robota pomembno, da pozna svoj trenutni položaj — koordinate v prostoru. Kot član projektne ekipe v podjetju, ki se ukvarja s pisanjem programske opreme za robote, si dobil nalogo, da **napišeš program** za robota, ki bo lahko na podlagi njegovega premikanja in obračanja ugotovil robotove koordinate v prostoru.

Vhodni podatki: v prvi vrstici vhoda se nahajajo 3 naravna števila n, x in y (vsa med 1 in 1000). Pri tem n predstavlja število ukazov (premikov ali zasukov), ki jih robot izvrši; x in y pa sta začetni koordinati robota. V drugi vrstici vhoda se nahaja n znakov, ki predstavljajo dejanske ukaze, ki jih robot izvrši. Vsak znak je ena od črk `L`, `D` ali `N`, ki imajo naslednji pomen: `L` označuje robotov zasuk za 90° v levo (nasprotna smer urinega kazalca), `D` predstavlja 90° zasuk v desno (smer urinega kazalca), `N` pa premik naprej. Robot se premika po ravni površini, ki jo predstavimo s kartezičnim koordinatnim sistemom z vodoravno osjo x in navpično osjo y ter enoto, ki predstavlja natanko en premik robota. Robot je na svojih začetnih koordinatah x in y vedno obrnjen v smeri x-osi.

Pričakovan izhod programa: vrstica, ki vsebuje dve naravni števili x in y, ki predstavljata končni koordinati robota.

Tvoj program lahko bere s standardnega vhoda in piše na standardni izhod ali pa uporablja datoteke (npr. `vhod.txt` ali `izhod.txt`), karkoli ti je lažje.

Primer vhoda:

```
7 1 3
NNNLNND
```

Pripadajoči izhod:

```
4 5
```

4. Čokolada

V zgodbi o Janku in Metki je Metka ravno potisnila zlobno čarovnico v peč in osvobodila brata Janka, ki je zaklenjen v kletki čakal in se redil, da postane večerja. V silnem navdušenju nad koncem nevarnosti in svežem duhu svobode sta se brat in sestra lotila raziskovanja posesti dobrot. V sladkorni hišici sta našla cekine in dragulje, skrivni prehod pa ju je vodil v podzemno jamo, kjer je bila skrita ogromna čokolada, ki se je raztezala globoko v temno pozemlje. Čokolado sta si ogledovala vrstico po vrstico in kaj kmalu ugotovila, da ne gre za navadno čokolado, temveč za čokolado z lešniki in rozinami!

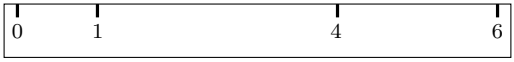
Sestradana Metka pa nad rozinami ni bila najbolj navdušena, zato je z bratom sklenila dogovor, da lahko prva izbere en strnjen odsek čokolade (eno ali več zaporednih vrstic skupaj), ki ji je najbolj všeč, on pa bo vzel preostanek. Pri tem lahko

Metka največ dvakrat razlomi čokolado in vzame kos med dvema lomoma oziroma začetni/končni odsek, če je čokolado prelomila enkrat.

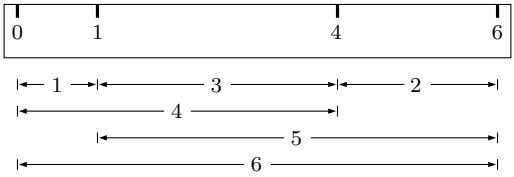
Ker je čokolada zelo velika, te prosita za pomoč. Iz jame ti bosta sporočila število vrstic čokolade, recimo mu n, in nato za vsako vrstico še to, koliko rozin in koliko lešnikov je v njej (recimo, da je v i-ti vrstici r_i rozin in ℓ_i lešnikov). Metka dobroto vrstice meri kot razliko v številu lešnikov in rozin $(\ell_i - r_i)$, dobroto večjega kosa čokolade (takega, ki obsega več zaporednih vrstic) pa kot vsoto dobrot posameznih vrstic v njem. **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki iz dobljenih podatkov (torej števil $n, r_1, \ldots, r_n, \ell_1, \ldots, \ell_n$) izračuna največjo vrednost dobrote, ki jo lahko doseže strnjen odsek čokolade (torej tak, ki ga sestavlja ena ali več zaporednih vrstic). Zaželeno je, da je tvoj postopek čim bolj učinkovit, tako da bo deloval dovolj hitro tudi za velike n.

5. Golombovo ravnilo

Ravni palici z označenimi in oštevilčenimi centimetri pravimo *ravnilo*. Pri tej nalogi imamo opravka z ravnilom, ki je varčno, a pomanjkljivo: označeni so le centimetri na nakaterih mestih, ne pa nujno na razdaljah vsakega centimetra. Primer takega ravnila je palica z oznakami na centimetrih $\{0, 1, 4, 6\}$:



Kljub pomanjkljivim oznakam lahko s tem ravnilom odmerimo vse celoštevilske razdalje med 0 in 6 cm, če le izberemo ustrezni dve oznaki na ravnilu. Hkrati se pri tem ravnilu nobena razdalja med poljubnima dvema oznakama ne ponovi. Na primer:



Enako lastnost ima na primer tudi ravnilo z oznakami $\{0, 2, 7, 8, 11\}$.

Napiši program, ki bo najprej prebral število oznak na ravnilu, potem pa še toliko celih števil, ki predstavljajo lego vsake oznake. Lege so podane v naraščajočem vrstnem redu (vsaka je strogo večja od prejšnje); prva lega je vedno 0, zadnja lega pa ustreza dolžini ravnila. Dolžina ravnila je največ 100000. Primer podatkov, ki opisujejo gornje ravnilo:

```
4
0
1
4
6
```

Program naj ugotovi in izpiše, ali za ravnilo, ki ga predstavljajo prebrani podatki, veljata naslednji dve lastnosti:

- vsak celoštevilski interval med poljubnima dvema oznakama se ponovi[1] kvečjemu enkrat, lahko pa kakšen interval manjka (mimogrede: takemu ravnilu pravimo *Golombovo ravnilo*);
- prisotni so vsi celoštevilski intervali od 1 do dolžine ravnila (mimogrede: takemu ravnilu pravimo *popolno* ravnilo).

Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `ravnilo.txt` (karkoli ti je lažje).

[1] Takšno besedilo smo uporabili na šolskem tekmovanju, vendar je ta formulacija napačna; mišljeno je, da se vsak interval *pojavi* kvečjemu enkrat (ponovi pa se sploh nikoli), npr. tako kot pri ravnilu $\{0, 1, 4, 6\}$ s primera na začetku.

NEUPORABLJENE NALOGE IZ LETA 2017

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 12. tekmovanjem ACM v znanju računalništva (leta 2017), 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. 85–141) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Dvojno zaokrožanje

Zavarovalnice zaokrožajo na goljufiv način: znesek 7,49 najprej zaokrožijo na 7,5 in to potem na 8, namesto da bi 7,49 že na začetku zaokrožile na 8. **Napiši program** ali podprogram, ki pregleda seznam števil in izračuna, koliko si na ta način prigoljufajo. Predpostavi, da so vsa števila večja od 0 in da so podana na dve decimalki natančno.

Težja različica: reši nalogo še za primer, ko so števila lahko podana tudi na več decimalk in jih smemo zaokrožiti ne le dvakrat, ampak poljubno mnogokrat in si pri tem vsakič sami izbrati, na koliko decimalk jih bomo zaokrožili. (Takrat lahko na primer 7,448 zaokrožimo najprej na 7,45, to potem na 7,5 in to na 8.)

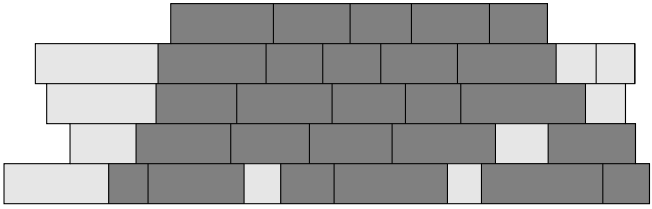
2. Volilni kolegij

Recimo, da ima neka država volilni sistem, podoben tistemu, s katerim v ZDA volijo predsednika. Imajo n regij, pri čemer ima i-ta regija p_i prebivalcev in r_i elektorjev. Na volitvah nastopita dva kandidata in vsak prebivalec voli za enega od njiju. Tisti kandidat, ki v neki regiji dobi večino glasov njenih prebivalcev, dobi potem vse elektorske glasove tiste regije; zmagovalec je kandidat, ki ima največ elektorskih glasov. (Recimo, da je vsak p_i lih in da je vsota $\sum_{i=1}^{n} r_i$ tudi liha, tako da ne more priti do izenačenih izidov.)

Izvedli so volitve in kandidat A je (za vsak i) v regiji i dobil a_i glasov, kandidat B pa preostalih $p_i - a_i$ glasov. Iz tega ni težko izračunati, kateri kandidat je zmagal. Tvoja naloga pa je, da **opišeš postopek**, ki ugotovi, kolikšno je najmanjše število volilcev, ki bi morali spremeniti svoj glas, da bi zmagal drugi kandidat.

3. Zid

Zid sestavlja več vrstic, vsako vrstico pa različno široki (vendar enako visoki) pravokotniki. Pravokotnik je *stabilen*, če leži v najnižji vrstici ali pa se z obema svojima spodnjima ogliščema dotika enega ali dveh pravokotnikov v vrstici pod svojo, vendar tako, da se njegovi oglišči ne dotikata njunih oglišč (ampak notranjosti njunih stranic). Radi bi odstranili čim več pravokotnikov, vendar moramo to narediti tako, da bodo potem vsi preostali pravokotniki stabilni, poleg tega pa ne smemo odstraniti nobenega pravokotnika iz najvišje vrstice. **Opiši postopek**, ki kot vhodne podatke dobi širine vseh pravokotnikov in ugotovi, koliko največ pravokotnikov lahko odstranimo (in katere).



Zgornja slika kaže primer zidu s 36 pravokotniki v 5 vrsticah. Pravokotniki, ki jih moramo obdržati, so pobarvani temno sivo, tisti, ki jih smemo odstraniti, pa svetlo sivo.

4. Kolovodja

Dan je usmerjen graf, v katerem točke predstavljajo mafijce, povezave pa povedo, kdo je čigav šef (vsaka povezava gre od šefa do podrejenega). **Opiši postopek**, ki ugotovi, ali obstaja „kolovodja“, torej tak mafijec, ki je (posredno ali neposredno) šef vseh drugih mafijcev. Če obstaja, naj zanj tudi izračuna najmanjši k, pri katerem velja, da je mogoče od njega v največ k korakih priti do kateregakoli drugega mafijca.

Lažja različica: predpostavi, da v grafu ni ciklov.

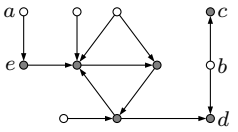
5. Podjetniške hobotnice

V podjetništvu je danes zelo moderno tako imenovano verižništvo podjetij, kjer je lastniška struktura med podjetji prepletena do nerazpoznavnosti. Skupine podjetij, ki jih prek vmesnih podjetij oziroma cele verige podjetij obvladujejo isti posamezniki, tako sestavljajo nepregledno hobotnico prepletenosti, kjer ni jasno, „kdo pije in kdo plača“.

Običajno ima podjetje več lastnikov, to so lahko posamezniki ali pa druga podjetja, vsak lastnik ima določen delež, praviloma izražen v odstotkih, in vsi deleži sestavljajo 100 %. Včasih je neko podjetje tudi lastnik (seveda samo delni) samega sebe ali pa imata dve podjetji v lastništvu določen delež drugo drugega. Recimo, da nas ne zanima velikost deležev, ampak samo obstoj posameznih deležev oziroma lastniških povezav.

Lastniška razmerja v podjetjih predstavimo s tabelo deležev, kjer so na navpični osi lastniki (to so podjetja in posamezniki kot imetniki deležev), na vodoravni osi pa podjetja. Kot rečeno, nas velikost deležev ne zanima, torej v poljih tabele ne bodo vpisane velikosti deležev v odstotkih, ampak samo *da* ali *ne* oziroma 1 ali 0.

Za boljšo predstavo lahko tabelo predstavimo z usmerjenim grafom, kjer so točke podjetja in posamezniki, povezave v grafu pa predstavljajo deleže v drugih podjetjih (kot rečeno, nas ne zanima velikost deleža, kar bi sicer lahko predstavili na primer z debelino povezav; v našem grafu so vse povezave enako debele). Če bi analizirali veliko število podjetij, bi dobili manjše in večje zaključene dele grafa, nekakšne „hobotnice“:



Primer hobotnice s 5 posamezniki (beli krožci) in 6 podjetji (temno sivi krožci).

Rekli bomo, da posameznik *vpliva* na podjetje, če ima lastniški delež v njem, lahko neposredno, lahko pa tudi posredno prek deležev v drugih podjetjih. *Primer:* na hobotnici z gornje slike vpliva posameznik a na vsa podjetja razen c; posameznik b vpliva le na podjetji c in d; ostali posamezniki vplivajo na vsa podjetja razen c in e.

Opiši postopke, ki odgovorijo na naslednja vprašanja:

(a) Kateri posameznik vpliva na največje število podjetij?

(b) Kateri so posamezniki, ki vplivajo na neko konkretno podjetje?

(c) Ali obstaja kakšno tako podjetje, na katero ne vpliva noben posameznik? (V praksi bi si tak primer razlagali kot znak, da so naši podatki o lastništvu podjetij pomanjkljivi.)

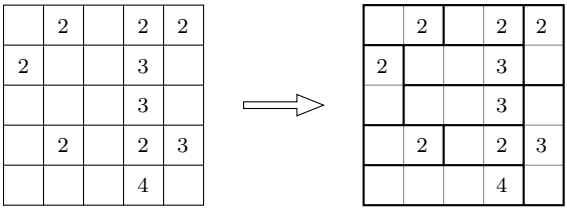
(d) Katere so „hobotnice“ v danih podatkih? S hobotnico mislimo tukaj na neprazno množico posameznikov in podjetij, za katero velja: za vsako podjetje v hobotnici so v njej tudi vsi posamezniki, ki vplivajo na to podjetje; za vsakega posameznika v hobotnici so v njej tudi vsa podjetja, na katera ta posameznik vpliva; in hobotnice ne moremo zmanjšati (odstraniti iz nje enega ali več posameznikov in/ali podjetij), ne da bi vsaj eden od prejšnjih dveh pogojev prenehal veljati.

6. Šikaku

Na igralni plošči 5×5 polj so v nekaterih poljih zapisana števila, ki imajo lahko vrednost od 1 do 4. Ostala polja so prazna. Cilj igre je, da vso ploščo pokrijemo s pravokotniki po naslednjih pravilih:

- Pravokotniki so lahko postavljeni vodoravno ali navpično.
- Vsak pravokotnik mora vsebovati natanko eno polje z vpisanim številom, to število pa mora biti enaka ploščini pravokotnika (torej številu polj v njem).
- Pravokotnik ne sme segati čez rob igralne plošče.

Naslednja slika kaže primer igralne plošče in enega od možnih načinov, kako jo pokriti s pravokotniki v skladu z gornjimi pravili:



Opiši postopek, ki na dano igralno ploščo postavi ustrezne pravokotnike v skladu s pravili igre.

7. Pretakanje tekočine

Celotno vsebino polne cisterne s prostornino P litrov želimo pretočiti v manjše posode. V trgovini je na voljo n različnih tipov posod. Pri tem imajo posode tipa i (za $i = 1, \ldots, n$) prostornino V_i litrov in ceno C_i evrov. Posod vsakega tipa je v

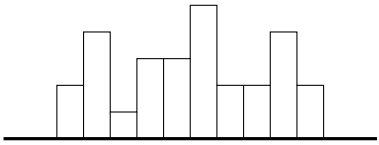
trgovini na voljo neomejeno število. Cena tipov posod narašča glede na prostornino posode, vendar ni premosorazmerna prostornini.

(*a*) **Opiši postopek**, ki izračuna, koliko posod katerega tipa potrebujemo, da bomo lahko vanje pretočili celotno vsebino cisterne in pri tem za nakup posod porabili najmanj denarja.

(*b*) **Opiši postopek**, ki izračuna, koliko največ vode lahko izpraznimo v kupljene posode, če imamo na voljo D evrov.

8. Barvanje ograje

Janezek mora za kazen prebarvati vrtno ograjo. Ta je sestavljena iz deščic širine 1 dm, ki so položene tesno skupaj ena ob drugi. Deščice so različnih višin, saj je ograjo izdelal Janezek sam iz odvečnih kosov, vsaka deščica pa se dotika tal, ki so ravna. Ograja je videti nekako takole:



Janezek pozna višino vsake deščice, zato lahko ograjo opiše s seznamom, ki za vsako deščico vsebuje po eno število (višina v dm), na primer:

$$[2, 4, 1, 3, 3, 5, 2, 2, 4, 2].$$

Janezek ima čopič, ki je širok ravno 1 dm. Ograjo bo barval z navpičnimi in vodoravnimi potegi čopiča (nikoli ne vleče čopiča diagonalno). To počne tako, da se ob vsakem trenutku vse ščetine čopiča dotikajo ograje. Ponovno lahko potegne čopič tudi preko že pobarvanega dela ograje. Janezek bi rad prebarval ograjo z minimalnim številom potez, da bi lahko čimprej odšel na obisk k Pepci. **Napiši podprogram** SteviloPotez(ograja), ki dobi seznam z opisom ograje in vrne minimalno število potez.

9. Krojač

Ste že kdaj gledali krojača, kako meri blago, ki je zvito na rolah? Seveda ga zanimajo le „tekoči metri“. Vsakič odmeri bodisi dolžino njegove merske palice (torej 1 m) ali pa odvije toliko blaga, da podvoji dolžino že odvitega blaga. Denimo, da je že odvitih x metrov blaga, potrebujemo pa y metrov ($y > x$). **Napiši funkcijo**, ki določi minimalno število potez, da bo krojač opravil svoje delo. Pri tem je poteza odmerek 1 m ($+1$) ali pa podvojena dolžina ($\times 2$).

Primeri: da pridemo od $x = 10$ do $y = 17$, potrebujemo 7 potez (7 operacij $+1$); od $x = 10$ do $y = 21$ potrebujemo 2 potezi (ena operacija $\times 2$ in ena operacija $+1$); od $x = 10$ do $y = 50$ pa 5 potez ($+1$, $+1$, $\times 2$, $+1$, $\times 2$).

Težja različica: reši nalogo še za primer, ko je poleg potez $+1$ in $\times 2$ dovoljena tudi poteza -1.

10. Najkrajše poti

Pri tej nalogi bomo delali z usmerjenimi *multigrafi* — to so grafi, v katerih sme za posamezni par točk (u, v) obstajati tudi po več različnih povezav $u \to v$ (čeprav imajo torej te povezave vse isto začetno krajišče u in tudi vse isto končno krajišče v, jih vendarle štejemo za različne; takšnim povezavam pravimo, da so *vzporedne*).

To, da imamo več vzporednih povezav, sicer nič ne vpliva na to, ali je ena točka dosegljiva iz druge in kako dolga je najkrajša pot od ene do druge, pač pa lahko vplivajo na to, *koliko* različnih poti od ene do druge točke obstaja, kajti če na primer dve poti naredita korak od u do v po dveh različnih vzporednih povezavah $u \to v$, štejemo tidve poti za različni.

Dan je multigraf G in točki s in t v njem. Vanj bi radi dodali eno povezavo $x \to y$ in to tako, da bo v dopolnjenem grafu (po dodajanju nove povezave) število različnih najkrajših poti od s do t čim večje. **Opiši postopek**, ki ugotovi, katero povezavo $x \to y$ bi bilo treba dodati in koliko bo potem teh najkrajših poti od točke s do točke t.

Težja različica: reši nalogo še za primer, ko delamo z navadnimi grafi namesto z multigrafi. To pomeni, da vhodni graf nima vzporednih povezav in da tudi mi ne smemo predlagati take nove povezave, ki bi bila vzporedna kakšni od že obstoječih.

11. Planinci

Dan je graf planinskih poti; v vsaki točki grafa se nahaja koča. Povezave so neusmerjene (po njih gremo lahko v obe smeri), ne tvorijo ciklov in po njih se dá priti od vsake točke do vsake druge točke (graf je torej povezan). Planinec gre na pot od koče s do koče t in se ustavi v vsaki vmesni koči. Ko pride do nezgode, mora reševalna služba ugotoviti, na kateri povezavi v drevesu se planinec nahaja. To storijo tako, da pokličejo v kočo in vprašajo, ali jo je iskani planinec že obiskal. Kakšno je minimalno število klicev, s katerim lahko zagotovo locirajo planinca? Takih planincev, ki se izgubljajo na danem drevesu gorskih poti, je več.

(a) **Opiši postopek**, ki bo učinkovito odgovarjal na takšne poizvedbe: kot vhodna podatka bo torej dobil s in t, izračunal pa bo najmanjše potrebno število klicev, s katerimi lahko lociramo planinca, ki je šel na pot od tega s do tega t. Graf se med poizvedbami ne spreminja, zato si lahko vnaprej pripraviš kakšne podatke, ki ti bodo prišli prav za hitrejše odgovarjanje na poizvedbe. Graf ima do 10^5 točk, pa tudi poizvedb je lahko do 10^5, zato naj bo postopek čim bolj učinkovit.

(b) **Opiši postopek**, ki planinca tudi dejansko poišče in pri tem izvede največ minimalno potrebno število klicev v planinske koče. Kot vhodna podatka torej dobi s in t, vrniti pa mora povezavo, na kateri se je planinec ponesrečil. Predpostavi, da je na voljo funkcija `JeObiskal(u)`, ki vrne logično vrednost (**true** ali **false**), ki pove, ali je planinec že obiskal kočo u ali ne.

12. Kontrolne vsote

Recimo, da imamo neko naravno število n. Izračunajmo vsoto njegovih števk (v desetiškem zapisu). Če ta vsota ni enomestno število, izračunajmo vsoto *njenih* števk; in tako naprej, dokler ne dobimo enomestne vsote (od 1 do 9). Tej zadnji vsoti bomo rekli *enomestna kontrolna vsota* števila n.

Če pogoj za ustavitev tega postopka spremenimo tako, da se ustavimo, čim dobimo vsoto, ki je dvomestno ali enomestno število, pa bomo dobljeni vsoti rekli *dvomestna kontrolna vsota* števila n. Če je bil n že sam po sebi eno- ali dvomesten, je on kar sam svoja dvomestna kontrolna vsota.

Primeri: iz $n = 899\,798\,998\,999\,989$ dobimo najprej vsoto števk 129, iz te 12, iz te pa 3; enomestna kontrolna vsota tega n je torej 3, dvomestna vsota pa 12. Pri $n = 100\,202$ sta tako eno- kot dvomestna kontrolna vsota enaki 5. Pri $n = 987$ je dvomestna kontrolna vsota enaka 24, enomestna pa 6. Pri $n = 65$ je dvomestna kontrolna vsota enaka 65, enomestna pa 2.

(a) **Napiši podprogram**, ki za dani n izračuna njegovo eno- in dvomestno kontrolno vsoto. Pravzaprav ga napiši v dveh različicah: eni, ki dobi n kot celoštevilsko vrednost (npr. tipa **int**), in eni, ki ga dobi kot niz (npr. tipa string ali **char ***; predpostavi, da nima vodilnih ničel); slednje lahko pride prav, če nas zanimajo zelo velika števila n.

(b) **Opiši postopek**, ki za dano enomestno število c (od 1 do 9) in za dani $k \geq 1$ izpiše poljubno tako k-mestno naravno število, v katerem so vse števke različne od 0 in ki ima enomestno kontrolno vsoto c.

(c) Če nas zanimajo malo večji k, recimo do 10^{18}, je lahko izpis takega k-mestnega števila neugodno počasen. **Napiši podprogram**, ki poleg c in k dobi kot parameter še celo število i z območja $1 \leq i \leq k$ ter izračuna le i-to števko tistega števila, ki bi ga pri teh c in k izpisal tvoj postopek iz podnaloge (b).

(d) **Opiši postopek**, ki za dano naravno število s izračuna najmanjše tako naravno število n, ki ima vsoto števk enako s. Koliko števk ima ta n (v odvisnosti od s)?

(e) **Opiši postopek**, ki za dani naravni števili a in b, pri čemer velja $a \leq b$, izračuna množico vseh možnih vsot števk, ki jih imajo števila od a do b. Zanima nas torej množica $\{V(n) : a \leq n \leq b\}$. Predpostavi, da sta števili a in b podani kot niza in da sta lahko precej dolgi; če je število b v desetiškem zapisu dolgo t števk, naj bo časovna zahtevnost tvoje rešitve $O(t)$. *Primer:* pri $a = 898$ in $b = 902$ gledamo torej števila 898, 899, 900, 901 in 902, njihove vsote števk pa so po vrsti 25, 26, 10, 11 in 12. Tvoj postopek bi moral torej pri teh a in b vrniti množico $\{10, 11, 12, 25, 26\}$.

(f) Kako bi bilo treba postopek iz podnaloge (e) spremeniti, da bi namesto množice vsot števk vračal množico dvomestnih kontrolnih vsot, torej $\{K_2(n) : a \leq n \leq b\}$?

(g) Recimo, da nas pri prejšnji podnalogi zanimajo primeri, ko je $b = 9a$. Zanima nas torej množica $M(a) := \{K_2(n) : a \leq n \leq 9a\}$. Kakšna je ta množica v odvisnosti od a? Pri katerih a vsebuje ta množica vse možne dvomestne kontrolne vsote od 1 do 99?

(h) Reši podnalogi (b) in (c) še za dvomestne kontrolne vsote namesto enomestnih; sestaviti moraš torej k-mestno število n s samimi neničelnimi števkami in z dvomestno kontrolno vsoto $K_2(n) = c$ (parameter c gre lahko od 1 do 99). Če primerno število n ne obstaja, naj tvoja rešitev javi napako.

13. Marsovska števila

Marsovci pri zapisu števil namesto števk uporabljajo nize, ki so sestavljeni iz malih črk angleške abecede in (pravilno gnezdenih) oklepajev. Pri tem veljajo naslednja

pravila:

- Vsaka mala črka je sama zase marsovsko število z neko znano konstantno vrednostjo.
- Če so nizi $s_1, s_2, \ldots, s_k$ (za $k > 1$) marsovska števila z vrednostmi $x_1, \ldots, x_k$, potem je tudi niz `(`$s_1 s_2 \ldots s_k$`)` marsovsko število, njegova vrednost pa se izračuna po naslednjem postopku:

  inicializiraj v na 0 in i na 1;
  ponavljaj, dokler je $i \leq n$:
    če je $i < k$ in $x_i < x_{i+1}$, potem
      povečaj v za $x_{i+1} - x_i$ in povečaj i za 2;
    sicer
      povečaj v za x_i in povečaj i za 1;

  Na koncu tega postopka predstavlja v ravno vrednost niza `(`$s_1 s_2 \ldots s_k$`)`. (Vidimo torej, da so marsovska števila neke vrste posplošitev rimskih števil, kjer se v primerih, kjer črka z manjšo vrednostjo stoji tik pred črko z večjo vrednostjo — na primer IV ali XC — ravno tako odšteje manjšo od večje.)
- Nizi, ki jih po gornjih dveh pravilih ne moremo sestaviti, pa niso marsovska števila.

Primer: recimo, da so dovoljene črke `p` (z vrednostjo 5), `d` (z vrednostjo 10) in `e` (z vrednostjo 1). Potem ima niz `(epd)` vrednost 14, niz `((ep)d)` ima vrednost 6, niz `(e(pd))` ima vrednost 4, niz `((ee)(eee))` ima vrednost 1 itd.

(*a*) Napiši podprogram, ki preveri, ali je dani niz veljavno marsovsko število.

(*b*) Napiši podprogram, ki za dani niz (in tabelo vrednosti posameznih črk abecede) izračuna njegovo vrednost (ali pa ugotovi, da niz ni veljavno marsovsko število).

(*c*) Opiši postopek, ki za dani n izračuna, na koliko načinov je mogoče v niz n črk dodati oklepaje tako, da nastane veljavno marsovsko število. Na primer, pri nizu dolžine $n = 4$ je takšnih možnosti 11: `(cccc)`, `(c(ccc))`, `(c((cc)c))`, `(c(c(cc)))`, `(c(cc)c)`, `(cc(cc))`, `((cc)cc)`, `((cc)(cc))`, `((ccc)c)`, `((c(cc))c)` in `(((cc)c)c)`.

(*d*) Opiši postopek, ki dobi niz črk (in tabelo vrednosti posameznih črk abecede) in ugotovi, kako je treba vanj dodati oklepaje, da nastane marsovsko število z največjo možno vrednostjo.

(*e*) Opiši postopek, ki dobi marsovsko število (in tabelo vrednosti posameznih črk abecede) in ugotovi, koliko največ oklepajev je mogoče pobrisati iz njega (in katere), če hočemo, da ima po brisanju število enako vrednost kot pred njim. Pri tem smemo oklepaje brisati le tako, da hkrati pobrišemo oklepaj in njemu pripadajoči zaklepaj; tako na primer iz marsovskega števila `(c(cc)(cc)c)` ne smemo narediti `(c(cccc)c)`, pač pa le `(ccc(cc)c)`, `(c(cc)ccc)` in `(cccccc)`.

Definirajmo še *alternirajoča marsovska števila*, ki so definirana z enakimi pravili kot običajna marsovska števila, le s to razliko, da se vrednost števila oblike $s =$ `(`$s_1 s_2 \ldots s_k$`)` računa po preprostejši formuli: če ima posamezni s_i vrednost x_i, je zdaj vrednost števila s definirana kot $x_1 - x_2 + x_3 - \ldots + (-1)^{k-1} x_k$.

(f) Reši podnalogo (d) še za alternirajoča marsovska števila.

(g) Opiši postopek, ki poišče največjo vrednost, ki jo je mogoče dobiti iz danega alternirajočega marsovskega števila z brisanjem oklepajev. (Podobno kot pri (e) smemo tudi tu oklepaje brisati le v parih: oklepaj in njemu pripadajoči zaklepaj.)

(h) Reši podnalogo (e) še za alternirajoča marsovska števila.

REŠITVE NALOG ZA PRVO SKUPINO

1. Smučarski užitki

Ocene vseh prog je koristno hraniti v tabeli (v naši spodnji rešitvi je to vektor ocene). Najprej preberemo število prog, nato gremo v zanki po vseh progah in preberemo njihove začetne ocene; nato pa do konca vhodnih podatkov beremo številke prevoženih prog in seštevamo njihove ocene, ki jih dobimo iz tabele ocene. Po vsaki vožnji pomnožimo oceno prevožene proge v tabeli z 0,9, da dobimo pravo oceno za naslednjo vožnjo po tej progi. Vsoto doslej prevoženih prog hranimo v spremenljivki vsota, ki jo na koncu tudi izpišemo.

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

int main()
{
  // Preberimo število prog.
  int p; cin >> p;
  // Preberimo začetne ocene vseh prog.
  vector<double> ocena(p);
  for (int i = 0; i < p; i++) cin >> ocena[i];
  // Preberimo vožnje in izračunajmo vsoto ocen.
  double vsota = 0;
  while (true)
  {
    int proga; cin >> proga;        // Preberimo naslednjo prevoženo progo.
    if (! cin.good()) break;        // Ali smo bili že na koncu vhodnih podatkov?
    vsota += ocena[--proga];        // Prištejmo njeno oceno k vsoti.
    ocena[proga] *= 0.9;            // V prihodnje bo njena ocena za 10 % manjša.
  }
  // Izpišimo rezultat.
  cout << vsota << endl; return 0;
}
```

Paziti moramo še na to podrobnost, da so številke prog v vhodnih podatkih od 1 do p, indeksi v tabeli oz. vektorju ocena pa gredo od 0 naprej. Naša gornja rešitev zato številke prevoženih prog zmanjša za 1, preden jih uporabi kot indekse v vektor ocena.

Še primer rešitve v pythonu:

```
import sys
# Preberimo število prog in njihove začetne ocene.
p = int(sys.stdin.readline())
ocene = [float(sys.stdin.readline()) for i in range(p)]
# Berimo prevožene proge in računajmo vsoto ocen.
vsota = 0
for vrstica in sys.stdin:
  # Prebrano vrstico pretvorimo v številko od 0 do p − 1.
  proga = int(vrstica) − 1
  vsota += ocene[proga]      # Prištejmo oceno te proge k vsoti.
  ocene[proga] *= 0.9        # V bodoče bo njena ocena za 10 % manjša.
# Izpišimo rezultat.
print(vsota)
```

2. Razmazani seznam

Kot namiguje že besedilo naloge, bomo elemente vhodne množice A brali v zanki in po vsakem prebranem elementu takoj izpisali vsa števila, za katera lahko že sklepamo, da pripadajo množici B. Na primer, če je naslednji element množice A enak x, lahko iz tega sklepamo, da B vsebuje med drugim vsa števila od $x - m$ do $x + v$. Tega, ali vsebuje B tudi kakšna večja števila, takrat še ne moremo vedeti, zato teh še ne bomo izpisovali. Glede števil, manjših od $x - m$, pa je tako: zaradi x in zaradi morebitnih kasnejših elementov A-ja (ki so vsi večji od x, saj naloga pravi, da nam funkcija Naslednji vrača elemente množice A v naraščajočem vrstnem redu) ne more priti v B nobeno število, manjše od $x - m$. Kar je torej takih števil v B, smo jih morali izpisati že prej, preden smo pri branju množice A sploh prišli do x, v bodoče pa se nam z njimi ne bo treba ukvarjati.

Paziti moramo še na to, da ne izpišemo istega elementa množice B po večkrat. Imejmo na primer spremenljivko z, ki hrani zadnji doslej izpisani element množice B. Ko preberemo x kot naslednji element A-ja, vemo, da moramo z izpisom nadaljevati pri $z + 1$ ali pa pri $x - m$, karkoli od tega dvojega je večje. Števila od tam naprej izpisujemo v vgnezdeni zanki, dokler ne pridemo do $x + v$.

Na začetku, ko nismo izpisali še ničesar, moramo z inicializirati na neko vrednost, ki je zagotovo manjša od vseh elementov množice B — to bo zagotovilo, da pri izpisu ne bomo nobenega pomotoma preskočili. Ker je najmanjša možna vrednost v množici A enaka 1, je najmanjša možna vrednost v množici B enaka $1 - m$, torej lahko z inicializiramo na $-m$, pa bo gotovo manjši od vseh elementov B-ja.

```
#include <iostream>
using namespace std;

void RazmazaniSeznam(int m, int v)
{
  int z = -m;
  while (true)
  {
    int x = Naslednji();    // Preberimo naslednji element množice A.
    if (x < 0) break;       // Ali smo že na koncu?
    // Doslej smo izpisali že vse tiste elemente B-ja, ki so ≤ z, in to so tudi vsi
    // elementi B-ja, ki so za največ v večji od doslej prebranih elementov A-ja.
    // Če z izpisom še nismo prišli do x - m, skočimo na to vrednost.
    if (z < x - m) z = x - m - 1;
    // Izpišimo števila do vključno x + v.
    while (z < x + v) cout << ++z << endl;
  }
}
```

Zapišimo takšno rešitev še v pythonu:

```
def RazmazaniSeznam(m, v):
  z = -m
  while True:
    x = Naslednji()                  # Preberimo naslednji element množice A.
    if x < 0: break                  # Ali smo že na koncu?
    z = max(z, x - m - 1)            # Preskočimo števila, manjša od x - m.
    while z < x + v: z += 1; print(z)  # Izpišimo števila do vključno x + v.
```

3. Veriga

Ko beremo vhodne podatke po vrsticah, si je koristno poleg trenutne vrstice zapomniti še prejšnjo (v spodnjem programu jo hranimo v spremenljivki prejsnja). Ko preberemo novo vrstico, se v zanki zapeljemo po njej in primerjamo njene znake z istoležnimi znaki prejšnje vrstice; če se znaka ujemata, vemo, da se tu nadaljuje veriga iz prejšnje vrstice, sicer pa se začenja nova veriga (ki je zaenkrat dolga le 1 znak). Pri primerjanju pazimo še na to, da je lahko prejšnja vrstica krajša od trenutne. Dolžino dosedanje verige za vsak stolpec (teh je največ 80, saj naloga pravi, da so vrstice dolge največ 80 znakov) hranimo v tabeli v.

Poleg tega hranimo tudi podatke o najdaljši verigi doslej, ne le o njeni dolžini, ampak tudi o tem, v katerem stolpcu leži in v kateri vrstici se začne, saj bomo morali to na koncu izpisati. Vsakič ko izračunamo novo dolžino verige v trenutnem stolpcu (pri trenutni vrstici), pogledamo, če je daljša od najdaljše doslej, in če je, si jo zapomnimo.

```
#include <cstdio>
#include <string>
using namespace std;

int main()
{
  enum { MaxDolz = 80 };
  int najVeriga = 0, najStolpec = −1, najVrstica = −1;
  int v[MaxDolz]; for (int x = 0; x < MaxDolz; x++) v[x] = 0;
  string vrstica;
  for (int stVrstice = 1; ! NaKoncu(); stVrstice++)
  {
    // Preberimo naslednjo vrstico, prejšnjo pa si začasno še zapomnimo.
    string prejsnja = vrstica; vrstica = Vrstica();
    // Primerjajmo istoležne znake obeh vrstic in popravljajmo dolžine verig.
    for (int x = 0; x < vrstica.length(); x++)
    {
      // Ali lahko podaljšamo verigo iz prejšnje vrstice?
      if (x < prejsnja.length() && vrstica[x] == prejsnja[x]) v[x]++;
      else v[x] = 1;    // Če ne, začnimo novo verigo dolžine 1.
      // Če je to najdaljša veriga doslej, si jo zapomnimo.
      if (v[x] > najVeriga)
        najVeriga = v[x], najStolpec = x,
        najVrstica = stVrstice − v[x] + 1;
    }
  }
  // Izpišimo rezultat.
  printf("Najdaljša veriga je dolga %d znakov, leži v stolpcu %d in "
       "se začne v vrstici %d.\n", najVeriga, najStolpec + 1, najVrstica);
  return 0;
}
```

Iz primera v besedilu naloge je razvidno, da naj bi se stolpce štelo od 1 naprej, na kar moramo paziti pri izpisu, saj drugod kot številke stolpcev in indekse v tabelo v uporabljamo števila od 0 naprej.

Zapišimo to rešitev še v pythonu:

```
MaxDolz = 80; v = [0] * MaxDolz
najVeriga = 0; najStolpec = −1; najVrstica = −1; vrstica = ""; stVrstice = 0

while not NaKoncu():
  # Preberimo naslednjo vrstico, prejšnjo pa si začasno še zapomnimo.
  prejsnja = vrstica; vrstica = Vrstica(); stVrstice += 1

  # Primerjajmo istoležne znake obeh vrstic in popravljajmo dolžine verig.
  for x in range(len(vrstica)):
    # Ali lahko podaljšamo verigo iz prejšnje vrstice?
    if x < len(prejsnja) and vrstica[x] == prejsnja[x]: v[x] += 1
    else: v[x] = 1 # Če ne, začnimo novo verigo dolžine 1.

    # Če je to najdaljša veriga doslej, si jo zapomnimo.
    if v[x] > najVeriga: najVeriga = v[x]; najStolpec = x; najVrstica = stVrstice − v[x] + 1

# Izpišimo rezultat.
print("Najdaljša veriga je dolga %d znakov, leži v stolpcu %d in "
    "se začne v vrstici %d." % (najVeriga, najStolpec + 1, najVrstica))
```

4. Jezero

Kot pove že besedilo naloge, bo naša rešitev tekla v neskončni zanki. Ko preberemo novo gladino vode, jo lahko primerjamo s prejšnjo meritvijo in tako določimo smer gibanja (ali gladina raste, pada ali ostaja enaka); spodnji program predstavi smer s celoštevilsko spremenljivko, ki ima lahko vrednost +1, 0 ali −1.

Novo smer gibanja gladine nato primerjajmo s tisto iz prejšnje iteracije naše zanke in tako ugotovimo, ali se nadaljuje dosedanja smer ali ne. V spremenljivki trajanje hranimo podatek o tem, kako dolgo se gladina že giblje v dosedanjo smer. Če je nova smer enaka prejšnji, moramo le povečati števec trajanje, sicer pa ga postavimo na 1 in si novo smer zapomnimo v spremenljivki smer.

Zdaj imamo vse, kar potrebujemo za odločitev o tem, kdaj odpreti ali zapreti zapornico. Če je na primer nova globina pod 33, jo zapremo. Zapreti jo moramo načeloma tudi, če je smer gibanja rastoča (smer == 1) in to traja že 11 korakov (to pomeni, da zadnjih 12 meritev tvori strogo naraščajoče zaporedje), vendar moramo v tem primeru preveriti še, da gladina ni nad 66 (saj naloga pravi, da imata prvi dve pravili prednost pred drugima dvema, torej pri gladini nad 66 zapornice ne smemo zapreti). Podobno razmišljamo tudi pri pravilih za odpiranje zapornice.

```
void Jezero()
{
  int globina = −1, smer = 0, trajanje = 0;
  while (true)
  {
    // Preberimo novo gladino vode.
    int novaGlobina = GlobinaVode();
    if (globina < 0) globina = novaGlobina;

    // Določimo smer gibanja (ali globina raste, pada ali ostaja enaka).
    int novaSmer = (novaGlobina > globina) ? 1 : (novaGlobina < globina) ? −1 : 0;

    // Kako dolgo se že giblje v to smer?
    if (smer == novaSmer) trajanje++;
    else smer = novaSmer, trajanje = 1;

    // Zapomnimo si globino za naslednjo iteracijo.
    globina = novaGlobina;
```

```
    // Spremenimo stanje zapornice, če je treba.
    if (globina < 33 || (globina <= 66 && smer < 0 && trajanje >= 11))
      PremakniZapornico(false);
    else if (globina > 66 || (globina >= 33 && smer > 0 && trajanje >= 11))
      PremakniZapornico(true);
  }
}
```

Še rešitev v pythonu:

```
globina = −1; smer = 0; trajanje = 0
while True:
  # Preberimo novo gladino vode.
  novaGlobina = GlobinaVode();
  if globina < 0: globina = novaGlobina;

  # Določimo smer gibanja (ali globina raste, pada ali ostaja enaka).
  novaSmer = 1 if novaGlobina > globina else −1 if novaGlobina < globina else 0

  # Kako dolgo se že giblje v to smer?
  if smer == novaSmer: trajanje += 1
  else: smer = novaSmer; trajanje = 1;

  # Zapomnimo si globino za naslednjo iteracijo.
  globina = novaGlobina;

  # Spremenimo stanje zapornice, če je treba.
  if globina < 33 or globina <= 66 and smer < 0 and trajanje >= 11:
    PremakniZapornico(False)
  elif globina > 66 or globina >= 33 and smer > 0 and trajanje >= 11:
    PremakniZapornico(True)
```

Nalogo bi se dalo seveda rešiti tudi drugače; lahko bi na primer hranili zadnjih 12 meritev v tabeli ali seznamu ter šli po vsakem branju nove meritve z zanko po tem seznamu in preverjali, ali gladina v njem ves čas raste ali ves čas pada (ali nič od tega).

5. Stolpci in vrstice

Vhodni niz sestavljajo izmenično skupine črk in skupine števk. Ko ga beremo znak po znak, je koristno pri tem hraniti podatek o tem, ali smo bili doslej pri črkah ali pri števkah; spodnja rešitev ima v ta namen logično spremenljivko `crke`. Koordinati trenutne celice hranimo (oz. počasi računamo) v spremenljivkah `stolpec` in `vrstica`. Ko se premaknemo iz skupine števk v skupino črk (ali pa dosežemo konec niza), vemo, da je opisa trenutne celice konec, torej moramo njeni koordinati izpisati, spremenljivki `stolpec` in `vrstica` pa postaviti na 0, da bomo pripravljeni na naslednjo celico.

Med branjem črk moramo postopoma računati številko stolpca, med branjem števk pa številko vrstice. Slednje je preprosto: ko preberemo naslednjo števko, moramo vrednost, ki smo jo dobili iz predhodnih števk, pomnožiti z 10 in ji prišteti vrednost nove števke. Na primer: če smo doslej prebrali števke `123`, smo imeli v spremenljivki `vrstica` vrednost 123; in če je naslednji znak potem števka `4`, bomo spremenljivko `vrstica` pomnožili z 10, ji prišteli 4 in tako dobili želeno vrednost 1234.

Pri pretvorbi zaporedja črk v številko stolpca pa je stvar malo bolj zapletena. Začnemo lahko s podobnim razmislekom kot pri števkah, pri čemer črke od `A` do

`Z` obravnavamo tako, kot da bi bile to števke od 0 do 25. Tako lahko na primer enomestne nize od `A` do `Z` predelamo v števila od 0 do 25; podobno lahko dvomestne nize od `AA` do `ZZ` predelamo v števila od 0 do $26 \cdot 26 - 1 = 675$; tromestne nize od `AAA` do `ZZZ` predelamo v števila od 0 do $26 \cdot 26 \cdot 26 - 1 = 17\,575$; in tako naprej. Vidimo pa, da to še ni dobro, saj se nam tako dobljena števila pri vsaki dolžini niza začnejo od 0, namesto da bi se nadaljevala tam, kjer so se pri prejšnji dolžini končala. Številu, ki smo ga dobili na ta način, moramo torej prišteti število vseh krajših črkovnih nizov. Na primer, če imamo niz treh črk, moramo prišteti število eno- in dvočrkovnih nizov, torej $1 + 26 + 26^2$ (1 na začetku potrebujemo zato, ker hočemo številke stolpcev od 1 naprej namesto od 0 naprej). Spodnji program računa to vrednost v spremenljivki prej in jo na koncu (pri izpisu) prišteje k vrednosti spremenljivke stolpec (ki je nastala pri pretvorbi prebranih črk v števke). Z drugimi besedami lahko tudi rečemo, da prej pove, koliko stolpcev ima krajši niz od našega, stolpec pa pove, koliko stolpcev ima enako dolg niz kot naš, vendar je njihov niz po abecedi pred našim.

```
#include <stdio.h>

int main()
{
  int stolpec = 0, vrstica = 0, c, prej = 0;
  bool crke = true;
  do {
    c = fgetc(stdin); // Preberimo naslednji znak.
    if (c >= '0' && c <= '9')
    {
      // c je števka; ali smo prečkali mejo med črkami in števkami?
      if (crke) crke = false, vrstica = 0;
      // Popravimo spremenljivko „vrstica“, da bo upoštevala števko c.
      vrstica = 10 * vrstica + (c - '0');
    }
    else
    {
      // c je črka ali pa konec vhodnih podatkov (EOF).
      // Ali se je pravkar končala skupina števk (in s tem opis ene celice)?
      if (! crke) {
        // Izpišimo koordinate dosedanje celice in se pripravimo na novo.
        printf("%d, %d\n", stolpec + prej, vrstica);
        stolpec = 0; prej = 0; crke = true; }
      // Popravimo spremenljivko „stolpec“, da bo upoštevala števko c.
      stolpec = 26 * stolpec + (c - 'A');
      // Popravimo „prej“, da bo spet vsebovala število vseh krajših nizov.
      prej = 26 * prej + 1;
    }
  } while (c != EOF);
  return 0;
}
```

Oglejmo si še primer rešitve v pythonu. Ta bo za spremembo prebrala celoten vhodni niz naenkrat v spremenljivko. Opise celic v njem potem beremo v zanki, znotraj nje pa imamo najprej vgnezdeno zanko, ki bere črke (in računa vrednosti stolpec in prej po enakem postopku kot zgoraj), nato pa še zanko, ki bere števke (in računa številko vrstice):

```
import sys
s = sys.stdin.readline().strip()
i = 0
while i < len(s):
    stolpec = 0; prej = 0; vrstica = 0

    # Preberimo opis stolpca.
    while s[i].isalpha():
        stolpec = 26 * stolpec + ord(s[i]) - ord('A')
        prej = 26 * prej + 1
        i += 1

    # Preberimo opis vrstice.
    while i < len(s) and s[i].isdigit():
        vrstica = 10 * vrstica + ord(s[i]) - ord('0')
        i += 1

    print("%d, %d" % (stolpec + prej, vrstica)) # Izpišimo trenutno celico.
```

Rešitev lahko še malo poenostavimo z naslednjim opažanjem: ker nas na koncu zanima le vsota vrednosti stolpec in prej, ne pa vsaka od teh dveh spremenljivk sama zase, in ker obe popravljamo (po vsaki prebrani črki) na zelo podoben način, lahko namesto dveh ločenih spremenljivk uporabimo eno samo, ki bo hranila vsoto obeh. Če tej novi spremenljivki rečemo kar stolpec, jo moramo inicializirati na 0 (tako kot doslej) in jo popravljati po naslednji formuli:

```
stolpec = 26 * stolpec + (c - 'A') + 1;
```

Spremenljivke prej pa zdaj sploh ne potrebujemo več.

REŠITVE NALOG ZA DRUGO SKUPINO

1. Anagramska razdalja

Ker nam je načeloma vseeno, kateri anagram t-ja dobimo (da bo le število uporabljenih operacij čim manjše), nas niti pri s-ju niti pri t-ju ne bo zanimal vrstni red znakov, pač pa le to, kolikokrat se katera črka pojavi v nizu. Recimo, da se nek znak c pojavi $\#_c(s)$-krat v nizu s in $\#_c(t)$-krat v nizu t. Minimum od tega dvojega, $m_c := \min\{\#_c(s), \#_c(t)\}$, nam pove, koliko c-jev v nizu s moramo pustiti pri miru, ker jih bomo potrebovali tudi še pri t. Nobene koristi ni od tega, da kakšnega od tistih c-jev brišemo ali spreminjamo v kakšen drug znak, saj bi morali potem nekoč kasneje kakšen c spet vriniti ali spremeniti kakšen drug znak vanj, s tem pa bi le po nepotrebnem povečevali število operacij.

Tako bo torej vsega skupaj $m := \sum_c m_c$ znakov s-ja prišlo prav tudi pri t-ju in se nam z njimi ni treba ukvarjati. Ostane še $|s| - m$ znakov, ki so odveč (nobeden od njih se ne pojavlja v t-ju), in po drugi strani manjka $|t| - m$ znakov (ki jih bomo potrebovali v t-ju, pa se ne pojavljajo v s-ju). Minimum od tega dvojega, torej $\min\{|s|, |t|\} - m$, nam pove, koliko odvečnih znakov s-ja lahko spremenimo v manjkajoče znake; potem pa, če je $|s| > |t|$, nam ostane še $|s| - |t|$ znakov, ki jih je treba pobrisati, če pa je $|t| > |s|$, nam manjka še $|t| - |s|$ znakov, ki jih je treba vriniti. Skupno število brisanj ali vrivanj je torej v vsakem primeru $\max\{|s|, |t|\} - \min\{|s|, |t|\}$. Če k temu prištejemo še skupno število spreminjanj znakov od prej, torej $\min\{|s|, |t|\} - m$, imamo vsega skupaj $\max\{|s|, |t|\} - m$ operacij. To je rezultat, po katerem sprašuje naloga.

Opisanega razmisleka ni težko zapisati kot podprogram v C++:

```
int AnagramskaRazdalja(const char *s, const char *t)
{
  // Preštejmo pojavitve vsake črke v s in v t.
  int ns[26] = { }, nt[26] = { };
  while (*s) ns[*s++ - 'a']++;
  while (*t) nt[*t++ - 'a']++;
  // Izračunajmo dolžino obeh nizov (ds, dt) in število skupnih črk (m).
  int ds = 0, dt = 0, m = 0;
  for (int c = 0; c < 26; c++) {
    ds += ns[c]; dt += nt[c];
    m += (ns[c] < nt[c]) ? ns[c] : nt[c]; }
  // Izračunajmo potrebno število operacij.
  return (ds > dt ? ds : dt) - m;
}
```

Oglejmo si še en način, kako priti do enakega rezultata. Namesto dveh tabel, ki štejeta pojavitve vsake črke v s in v t, imejmo eno samo tabelo, kjer za vsako črko štejemo razlike med številom njenih pojavitev v s in v t. Na koncu bodo torej tu vrednosti $\#_s(c) - \#_t(c)$ za vse c. Pozitivna števila v tej tabeli predstavljajo črke, ki so v s odveč (ker jih v t ni), negativna pa črke, ki v s manjkajo (v t pa so prisotne). Zdaj lahko v zanki seštejemo pozitivna posebej in negativna posebej (pri slednjih pravzaprav vzemimo njihove absolutne vrednosti), pa dobimo skupno število odvečnih in skupno število manjkajočih črk — to sta ravno vrednosti $|s| - m$

in $|t| - m$, o katerih smo govorili že zgoraj. Kot smo že videli, je iskano število operacij ravno večja izmed teh dveh vrednosti.

```
int AnagramskaRazdalja2(const char *s, const char *t)
{
  // Izračunajmo razliko v številu pojavitev posamezne črke v s in v t.
  int razlike[26] = {};
  while (*s) razlike[*s++ - 'a']++;
  while (*t) razlike[*t++ - 'a']--;

  // Izračunajmo število odvečnih in manjkajočih znakov (v s glede na t).
  int odvec = 0, manjka = 0;
  for (int c = 0; c < 26; c++)
    if (razlike[c] > 0) odvec += razlike[c];
    else manjka -= razlike[c];

  // Vrnimo potrebno število operacij.
  return (odvec > manjka) ? odvec : manjka;
}
```

2. Zaboji

Vrstni red, v katerem moramo pobirati zaboje iz skladišča, je točno določen; vprašanje je le to, kako zamikati zaboje, da spravimo na zadnje mesto tistega, ki ga moramo naslednjega pobrati. Nobene koristi ni od tega, da bi jih med dvema pobiranjema zamikali malo v levo in malo v desno, saj bi s tem le zapravljali čas. Preden začnemo zamikati zaboje, moramo torej pogledati, ali bomo do naslednjega zaboja, ki ga moramo pobrati, hitreje prišli tako, da bomo ves čas zamikali v levo, ali pa tako, da bomo ves čas zamikali v desno. Mogoče je tudi, da sta obe možnosti enakovredni (npr. če je trenutno stanje $3, 2, 5, 4$ in bi kot naslednjega radi pobrali zaboj 2, potrebujemo ali dva zamika v levo ali pa dva v desno).

Nalogo lahko torej rešujemo z neke vrste simulacijo, pri kateri v seznamu vzdržujemo stanje zabojev in na vsakem koraku pogledamo, kje je naslednji zaboj, ki ga moramo pobrati, in koliko zamikov bo treba izvesti, da pride na konec:

vhod: naj bo s seznam, ki vsebuje začetno stanje zabojev v skladišču;

$r := 0$; (* *v r bomo računali skupno število operacij* *)
for $z := 1$ **to** n:
  $k := n - z + 1$; (* *število preostalih zabojev v skladišču* *)
  $i :=$ položaj zaboja z v seznamu s (od 1 do k);
  $D := k - i$; (* *potrebno število zamikov v desno* *)
  $L := i$; (* *potrebno število zamikov v levo* *)
  $r := r + \min\{L, D\} + 1$; (* $+1$ *je za pobiranje zaboja* z *)
  if $L < D$ **then** zamakni s ciklično za L mest v levo
  else zamakni s ciklično za D mest v desno;
  pobriši z iz s-ja (kjer se zdaj nahaja na koncu seznama);
return r;

Časovna zahtevnost tega postopka je precej odvisna od tega, kako predstavimo seznam s in izvajamo operacije na njem. Preprosta rešitev je na primer s tabelo; iskanje zaboja s vzame tedaj $O(n)$ časa, prav tako tudi ciklični zamik za poljubno število mest v levo ali desno (to je najlažje izvesti tako, da zaboje začasno skopiramo

v pomožno tabelo), brisanje z-ja s konca tabele pa vzame le $O(1)$ časa, saj ni treba drugega, kot da si zapomnimo, da je seznam zdaj za en element krajši. Ker moramo naštete reči izvesti po enkrat v vsaki iteraciji glavne zanke, ta pa se izvede n-krat, je časovna zahtevnost celotnega postopka tako $O(n^2)$.

Na misel nam lahko pridejo razne ideje, kako to ali ono izmed naštetih operacij poceniti. Namesto da ciklično zamikamo celo tabelo, si lahko le zapomnimo indeks zadnjega zaboja v njej; zamikanje lahko potem izvedemo v $O(1)$ časa, vendar pa zato element z, ki ga moramo nato pobrisati, ne bo več nujno na koncu tabele, zato bo brisanje tega elementa zdaj vzelo $O(n)$ časa, ker bomo morali elemente, ki so v tabeli za z-jem, zamakniti za eno mesto v levo. Kaj pa, če tega slednjega ne bi naredili in bi pustili, da v tabeli ostane luknja? Brisanje je zdaj spet v $O(1)$ časa, poleg tega pa je vsak zaboj ves čas na istem mestu v tabeli, torej postane iskanje zaboja v seznamu trivialno in nam tudi vzame le $O(1)$ časa. Težava pa je zdaj v tem, da ni več očitno, koliko zabojev je med z in tistim na koncu skladišča, ker ne vemo, koliko je med njima lukenj v tabeli. Lahko se seveda zapeljemo po tabeli in zaboje preštejemo, vendar nam bo to spet vzelo $O(n)$ časa, temu pa bi se radi izognili. Vseeno zapišimo to rešitev malo podrobneje:

vhod: tabela $s[1..n]$ z začetnim stanjem zabojev v skladišču;

(* *Pripravimo si tabelo, ki pove, kje v s je kakšen zaboj.* *)
for $i := 1$ **to** n **do** $kje[s[i]] := i$;
(* *Pripravimo si tabelo, ki pove, kje so v s luknje.* *)
for $i := 1$ **to** n **do** $L[i] := 0$;

$zadnji := n$; (* *indeks (v s) najbolj desnega zaboja v skladišču; lahko je tudi indeks neke luknje desno od tistega zaboja* *)
$r := 0$; (* *skupno število operacij* *)
for $z := 1$ **to** n:
  $k := n - z + 1$; (* *število preostalih zabojev v skladišču* *)
  $i := kje[z]$; (* *položaj naslednjega zaboja, ki ga pobiramo* *)
  $od := \min\{i, zadnji\}$; $do := \max\{i, zadnji\}$;
  $Z_1 :=$ $do - od -$ vsota $L[od, \ldots, do]$;
  (* Z_1 *je število zamikov v eno smer — desno, če je* $i < zadnji$, *sicer pa levo.* *)
  $Z_2 := k - 1 - Z_1$; (* *število zamikov v drugo smer* *)
  $r := r + \min\{Z_1, Z_2\} + 1$;
  $zadnji := i$; (* *v mislih ciklično zamaknimo seznam* *)
  $L[i] := 1$; (* *pobrišimo z iz seznama, tam je zdaj luknja* *)
return r;

Vidimo lahko, da so skoraj vse operacije zdaj zelo poceni, težava je le štetje lukenj med zabojem z in koncem skladišča. V ta namen moramo sešteti več zaporednih elementov tabele L in če bomo to počeli z zanko po vseh teh elementih, bo imela naša rešitev na koncu še vedno zahtevnost $O(n^2)$. Do boljše rešitve pridemo, če nad tabelo L vzdržujemo kakšno primerno drevesasto strukturo, ki vsebuje vsote po več zaporednih elementov L-ja.

Ena možnost je na primer polno binarno drevo; nad tabelo L postavimo še eno, polovico manjšo tabelo L_1, v kateri vsak element vsebuje vsoto dveh zaporednih elementov tabele L; nad L_1 naredimo še pol manjšo tabelo L_2, v kateri vsak element

vsebuje vsoto dveh zaporednih elementov tabele L_1 (in s tem štirih zaporednih elementov tabele L_2); in tako naprej. Tako imamo $O(\log n)$ nivojev tabel in da izračunamo vsoto poljubnega zaporedja elementov tabele L, moramo na vsakem nivoju pogledati največ dve števili. Tudi ko povečamo nek element L-ja z 0 na 1, moramo na vsakem nivoju popraviti največ eno vrednost. Tako imamo v vsaki iteraciji glavne zanke (po z) zdaj $O(\log n)$ dela in časovna zahtevnost celotne rešitve je zdaj $O(n \log n)$.

Še bolj elegantna podatkovna struktura za naš namen je Fenwickovo drevo, pri katerem originalno tabelo L kar povozimo z malo drugačno tabelo L', v kateri $L'[i]$ vsebuje vsoto členov $L[f(i)+1, \ldots, i]$ prvotne tabele, pri čemer je $f(i)$ število, ki ga dobimo, če v dvojiškem zapisu i-ja ugasnemo najnižji prižgani bit. Tudi pri takšnem drevesu nam računanje poljubne vsote več zaporednih elementov L-ja in sprememba posameznega elementa L-ja vzameta po $O(\log n)$ časa.

Do rešitve s časovno zahtevnostjo $O(n \log n)$ lahko pridemo tudi tako, da zaporedje zabojev namesto s tabelo $s[1..n]$ predstavimo z neko primerno uravnoteženo drevesasto strukturo, na primer rdeče-črnim drevesom. V takih drevesih so elementi običajno urejeni po nekem vrstnem redu; v našem primeru bo vrstni red tak, kot je vrstni red zabojev v skladišču. Za vsako številko zaboja od 1 do n hranimo kazalec na tisto vozlišče drevesa, ki vsebuje ta zaboj. V vsakem vozlišču hranimo tudi število vseh zabojev v poddrevesu, ki se začne pri tem vozlišču. S pomočjo tega podatka lahko v $O(\log n)$ časa preštejemo, koliko je zabojev desno od z. V $O(\log n)$ časa lahko izvedemo tudi brisanje, ciklični zamik (bodisi s prestavljanjem celih poddreves bodisi tako, da vzdržujemo kazalec na najbolj desni zaboj in ga tik pred brisanjem z-ja popravimo tako, da bo kazal na z-jevega predhodnika v seznamu).

3. Ograje

Vpeljimo v našo mrežo koordinatni sistem: koordinate naj gredo od $x = 0$ na levem robu do $x = w$ na desnem in od $y = 0$ na zgornjem robu do $y = h$ na spodnjem. Mrežo lahko zdaj opišemo z nekaj dvodimenzionalnimi tabelami:

- `polje[y][x]` naj bo število na tistem polju, ki ima zgornji levi kot v točki (x, y); če je polje prazno, imejmo tam vrednost -1. Pri tem gre lahko x od 0 do $w - 1$ in y od 0 do $h - 1$.

- `vod[y][x]` naj bo logična vrednost, ki pove, ali obstaja vodoravna ograja od točke (x, y) do $(x + 1, y)$. Tu gre lahko x od 0 do $w - 1$, koordinata y pa 0 do h, ne le do $h - 1$.

- `nav[y][x]` naj bo logična vrednost, ki pove, ali obstaja navpična ograja od (x, y) do $(x, y + 1)$. Zdaj gre lahko x od 0 do w (ne le do $w - 1$), y pa od 0 do $h - 1$.

Razmislimo zdaj o tem, kako bi preverili pogoje iz besedila naloge. Glede števila ograj, ki obdajajo posamezno polje, je stvar preprosta; z dvema gnezdenima zankama (po x in y) preglejmo vsa polja, za vsako neprazno polje pa poglejmo njegove štiri stranice in preštejmo ograje na njih. Če se to število ne ujema s tistim, ki je vpisano v polju, je mreža neveljavna.

Podobno lahko z dvema gnezdenima zankama tudi pregledamo vse elemente tabel `vod` in `nav` in preštejemo vse ograje. S tem bomo lahko preverili, če je na

mreži sploh prisotna kakšna ograja. Če je, si tudi zapomnimo kakšno točko, ki leži na ograji (vseeno je, katero); na primer, če opazimo vodoravno ograjo od (x, y) do $(x + 1, y)$, si zapomnimo točko (x, y) in smer desno; če pa opazimo navpično ograjo od (x, y) do $(x, y + 1)$, si zapomnimo točko (x, y) in smer dol.

Zdaj se postavimo v točko na ograji, ki smo si jo zapomnili v prejšnjem odstavku, in sledimo ograji v smeri, ki smo si jo zapomnili. Po vsakem koraku razmislimo, v katero smer se ograja nadaljuje; pri tem moramo paziti, da ne gremo nazaj v smer, iz katere smo ravnokar prišli. Če opazimo več možnih nadaljevanj, to pomeni, da je ograja razvejena in mreža neveljavna; če ni nobenega primernega nadaljevanja, pa to pomeni, da ograje ne tvorijo cikla in je mreža tudi neveljavna. Če se ne zgodi nič od tega, bomo sčasoma neizogibno prišli nazaj v točko, kjer smo naš obhod začeli. Takrat lahko pogledamo, če je število korakov, ki smo jih na obhodu naredili, enako skupnemu številu ograj; mreža je veljavna, če se števili ujemata (kajti če se ne, to pomeni, da ograje ne tvorijo enega samega cikla).

```
#include <vector>
using namespace std;

struct Mreza
{
  // m[y][x] = število na polju (x, y); −1, če je prazno
  vector<vector<int>> polja;
  // vod[y][x] = stanje ograje na zgornjem robu polja (x, y); tudi za y = h
  // nav[y][x] = stanje ograje na levem robu polja (x, y); tudi za x = w
  vector<vector<bool>> vod, nav;
  bool JeVeljavna() const;
};

bool Mreza::JeVeljavna() const
{
  // Prazna mreža ne more vsebovati ciklične ograje in je gotovo neveljavna.
  int h = polja.size(); if (h == 0) return false;
  int w = polja[0].size(); if (w == 0) return false;

  // Preverimo, če je okrog vsakega polja pravo število ograj.
  for (int y = 0; y < h; y++) for (int x = 0; x < w; x++)
  {
    if (polja[y][x] < 0) continue; // prazno polje
    int stOgraj = (vod[y][x] ? 1 : 0) + (vod[y + 1][x] ? 1 : 0) +
                  (nav[y][x] ? 1 : 0) + (nav[y][x + 1] ? 1 : 0);
    if (polja[y][x] != stOgraj) return false;
  }

  // Preštejmo ograje. Eno od točk na ograji (in smer ograje iz te točke)
  // si zapomnimo v spremenljivkah x0, y0 in smer.
  enum { Desno = 0, Dol = 1, Levo = 2, Gor = 3 };
  const int DX[] = { 1, 0, −1, 0 }, DY[] = { 0, 1, 0, −1 };
  int x0, y0, smer, stOgraj = 0;
  for (int y = 0; y <= h; y++) for (int x = 0; x <= w; x++) {
    if (x < w && vod[y][x]) stOgraj++, x0 = x, y0 = y, smer = Desno;
    if (y < h && nav[y][x]) stOgraj++, x0 = x, y0 = y, smer = Dol; }
  if (stOgraj <= 0) return false; // Vsaj ena ograja mora biti prisotna.

  // Sledimo zdaj ograji od točke (x0, y0) naprej.
  int x = x0, y = y0, stKorakov = 0;
  while (true)
```

```
  {
    // Premaknimo se v trenutno smer.
    x += DX[smer]; y += DY[smer]; stKorakov++;
    // Če pridemo nazaj v začetno točko (x0, y0), končajmo.
    if (x == x0 && y == y0) break;
    // Določimo smer nadaljevanja.
    int smerIz = (smer + 2) % 4; smer = -1;
    for (int s = 0; s < 4; s++)
    {
      if (s == smerIz) continue; // Iz te smeri smo prišli.
      // Preverimo, ali obstaja ograja v smeri „s“.
      int x2 = x + DX[s], y2 = y + DY[s];
      if (x2 < 0 || x2 > w || y2 < 0 || y2 > h) continue;
      if (x < x2) x2 = x; if (y < y2) y2 = y;
      if (! (s % 2 ? nav[y2][x2] : vod[y2][x2])) continue;
      // Če je možno nadaljevanje v več kot eni smeri, so ograje razvejene.
      if (smer >= 0) return false;
      smer = s;
    }
    if (smer < 0) return false; // Smo v slepi ulici.
  }
  // Zdaj smo prehodili en cikel. Če smo s tem uporabili vse ograje,
  // je mreža veljavna, sicer očitno obstaja še nek ločen niz ograj.
  return stKorakov == stOgraj;
}
```

4. Past za žvižgače

Najprej preberimo število n (od 1 do 32), ki pove, katero različico vhodnega besedila moramo izpisati. Najenostavnejši način, kako v odvisnosti od tega števila pripraviti 32 različnih kopij vhodnega besedila, je ta, da prvih pet besed „in“ ali „ter“ spremenimo glede na spodnjih pet bitov števila n. Števila od 1 do 32 se namreč razlikujejo v svojih spodnjih petih bitih (natančneje povedano: nobeni dve od teh števil se ne ujemata popolnoma v teh petih bitih). V neki spremenljivki — v spodnjem programu je to bit — si zapomnimo, pri katerem bitu smo zdaj (oz. z drugimi besedami: koliko besed „in“ ali „ter“ smo v vhodnem besedilu doslej že videli); ko naletimo na naslednjo besedo „in“ ali „ter“, pogledamo ustrezni bit števila n in se glede na njegovo vrednost odločimo, ali bi izpisali „in“ ali „ter“. Vse drugo v vhodnem besedilu pa lahko izpisujemo brez sprememb.

Naša spodnja rešitev bere vhodno besedilo znak po znak. Ko začnemo brati neko besedo, vnaprej še ne moremo vedeti, ali jo bomo morali izpisati nespremenjeno ali ne, zato si njene znake sprva shranjujmo v tabelo s, zapomnimo pa si tudi njeno dolžino d. Če se izkaže, da je beseda daljša od treh znakov, lahko takoj zaključimo, da to ni niti „in“ niti „ter“, torej jo bomo morali izpisati nespremenjeno; zato takoj izpišimo tisto, kar že imamo v s, dolžino d pa postavimo na -1 kot znak, da bomo preostanek te besede izpisovali kar sproti. Če pa pridemo do konca besede (torej do znaka, ki ni črka, ali pa do konca vhodnih podatkov) in opazimo, da je bila dolga 3 znake ali manj, lahko s pomočjo tabele s preverimo, če je bila to ena od besed „in“ in „ter“. Če ni bila ali pa če smo videli že vsaj pet takih besed, jo lahko izpišemo nespremenjeno; sicer pa se za to, ali izpisati „in“ ali „ter“, odločimo na podlagi

naslednjega bita števila n.

```
#include <stdio.h>

int main()
{
  int n, c, d = 0, bit = 0; scanf("%d\n", &n); n--;
  char s[3]; // trenutna beseda
  do
  {
    c = fgetc(stdin);
    if (('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z'))
    {
      // Če trenutna beseda še ni predolga, dodajmo novo črko v „s“ in je ne izpišimo.
      if (0 <= d && d <= 2) { s[d++] = c; continue; }

      // Če je ravnokar postala predolga, izpišimo doslej zadržane črke iz „s“.
      for (int i = 0; i < d; i++) fputc(s[i], stdout);

      // Odtlej črke predolge besede izpisujemo sproti.
      d = -1; fputc(c, stdout); continue;
    }

    // Smo na koncu besede; preverimo, če je to ena od prvih petih „in“ in „ter“.
    if (bit < 5 && ((d == 2 && s[0] == 'i' && s[1] == 'n') ||
                    (d == 3 && s[0] == 't' && s[1] == 'e' && s[2] == 'r')))

      // Če je, izpišimo „in“ ali „ter“ glede na trenutni bit n-ja.
      printf(((n >> bit++) & 1) ? "in" : "ter");

    // Sicer izpišimo vhodno besedo, če je še nismo.
    else for (int i = 0; i < d; i++) fputc(s[i], stdout);

    // Izpišimo še pravkar prebrani ne-črkovni znak.
    d = 0; if (c != EOF) fputc(c, stdout);
  }
  while (c != EOF);
  return 0;
}
```

Oglejmo si še malo drugačno rešitev, tokrat za spremembo v pythonu. Ta bo brala vhodno besedilo po vrsticah in vsako s pomočjo regularnih izrazov (modul `re` iz pythonove knjižnice) razbila na skupine črkovnih (`\w*`) in nečrkovnih (`\W*`) znakov, torej na besede in presledke med njimi. Za vsako besedo potem preverimo, če je to ena od „in“ in „ter“ in če takih še nismo videli vsaj pet; v tem primeru pogledamo trenutni bit vrednosti n, da se odločimo, kaj izpisati; sicer pa izpišemo vhodno besedo brez sprememb. Nato pa v vsakem primeru izpišemo še presledke.

```
import sys, re
n = int(sys.stdin.readline()); bit = 0; inTer = ("in", "ter")

# Berimo vhodno besedilo po vrsticah.
for s in sys.stdin:

  # Razbijmo vrstico na besede in presledke med njimi.
  for m in re.finditer(r"(\w*)(\W*)", s):

    # Ali je to ena od prvih petih besed „in“ ali „ter“?
    if bit < 5 and m[1] in inTer:

      # Če da, izpišimo tisto, kar zahteva trenutni bit n-ja.
      sys.stdout.write(inTer[(n >> bit) & 1]); bit += 1
```

```
    # Sicer izpišimo vhodno besedo nespremenjeno.
    else: sys.stdout.write(m[1])

    # Izpišimo še presledke za besedo.
    sys.stdout.write(m[2])
```

5. Pekarna

Naloga pravi, da je Vahtarjev odgovor na prejeti izziv vedno ravno dvakratnik števila, ki ga je dobil kot izziv. Ko torej dobimo odgovor, ga moramo le deliti z 2, pa bomo videli, na kateri izziv se nanaša. Znati pa moramo tudi nekako ugotoviti, kdaj smo tisti izziv poslali, da bomo lahko preverili, ali se odgovor nanaša na več kot 10 sekund star izziv. To med drugim pomeni, da se nam izzivi ne smejo (prepogosto) ponavljati (to je koristno tudi zaradi primera, ko Vahtar obvisi in v nedogled ponavlja zadnji poslani odgovor).

Lahko bi si v neki tabeli ali seznamu zapisovali poslane izzive in pri vsakem še čas, ko smo ga poslali; še lažje pa je, če kot izziv pošljemo kar trenutni čas v sekundah, torej vrednost, ki jo vrača funkcija Pocakaj. Za vsak primer ji bomo prišteli 1, da izziv gotovo ne bo enak 0 (kajti v tem primeru bi bil tudi odgovor 0 in potem ne bi mogli vedeti, ali nam funkcija Odgovor sporoča, da odgovora sploh ni, ali da je prišel odgovor z vrednostjo 0). Tega, da bi prišlo do prekoračitve obsega celih števil, se nam ni treba bati, saj naloga pravi, da vse računalnike tako ali tako enkrat na leto ustavijo in ponovno zaženejo, torej časi v sekundah ne bodo šli čez nekaj deset milijonov (v 365 dneh je 31 536 000 sekund).

V glavni zanki naše rešitve najprej pokličemo Pocakaj, da počakamo na začetek nove sekunde. Za vsakega od ostalih dveh računalnikov potem najprej preverimo, če nam že predolgo dolguje odgovor; če da, mu ugasnemo napajalnik, sicer pa mu pošljemo nov izziv. S tem poskrbimo, da vsakemu računalniku (dokler deluje normalno) enkrat na sekundo pošljemo nov izziv, kot zahteva besedilo naloge. Nato še obdelamo odgovore, ki smo jih dobili od tega računalnika; pri vsakem izračunamo, kdaj je bil poslan izziv, na katerega se nanaša, in če je ta izziv starejši od 10 sekund, pošiljatelju odgovora ugasnemo napajalnik. Čas zadnjega prejetega odgovora od vsakega računalnika hranimo v tabeli zadnji.

```
int main()
{
  const int jaz = KdoSem();
  int zadnji[4] = { -1, -1, -1, -1 };

  while (true)
  {
    // Počakajmo na začetek naslednje sekunde.
    int zdaj = Pocakaj();

    // V zanki obdelajmo ostala dva računalnika.
    for (int x = 1; x <= 3; x++) if (x != jaz)
    {
      // Če že preveč zamuja z odgovorom, mu ugasnimo napajalnik.
      if (zadnji[x] > 0 && zdaj - zadnji[x] > 10) UgasniNapajalnik(x);
      // Sicer mu pošljimo nov izziv.
      else Vprasanje(x, zdaj + 1);

      // Obdelajmo prispele odgovore.
      while (true)
```

```
        {
          // Preberimo naslednji odgovor od računalnika x.
          int odg = Odgovor(x); if (odg == 0) break;

          // Zapišimo si čas, ko smo ta odgovor prejeli.
          zadnji[x] = zdaj;

          // Če se odgovor nanaša na prestaro vprašanje, mu ugasnimo napajalnik.
          int vprasanje = odg / 2 - 1;
          if (zdaj - vprasanje > 10) UgasniNapajalnik(x);
        }
      }
    }
    return 0;
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Fitnes

Za začetek lahko podatke o obiskovalcih uredimo po številki naprave, saj bomo morali kasneje tako ali tako obravnavati vsako napravo posebej, da bomo izračunali, koliko izvodov te naprave potrebujemo (pa tudi rezultate moramo izpisati v naraščajočem vrstnem redu številk naprave). Po takšnem urejanju pridejo zapisi, ki se nanašajo na posamezno napravo, skupaj v seznamu in vprašanje je le še, kako za takšno skupino obiskovalcev določiti, koliko jih je največ hkrati v fitnesu — to nam pove, koliko naprav tega tipa potrebujemo. V nadaljevanju našega razmisleka se torej lahko ukvarjamo z vsako tako skupino obiskovalcev posebej (in tudi sproti izpisujemo rezultate).

Preprosta (vendar neučinkovita) možnost je, da vzdržujemo tabelo z $24 \cdot 60 \cdot 60 \cdot 10 = 864\,000$ elementi; za vsako desetinko sekunde v dnevu imamo en element, ki pove, koliko obiskovalcev trenutne naprave je takrat prisotnih v fitnesu. Na začetku inicializiramo vse na 0, nato pa gremo za vsakega obiskovalca v zanki od s_i do vključno $e_i - 1$ in povečujemo števce na tistih mestih v tabeli. Na koncu se še enkrat sprehodimo po tabeli in določimo največjo vrednost v njej; to je iskano število naprav tega tipa. Slabost te rešitve je, da če je obiskovalcev veliko in je vsak v fitnesu skoraj cel dan, bo naša rešitev porabila preveč časa za povečevanje števcev v tabeli, zato bi pri večjih testnih primerih prekoračila časovno omejitev. Pri testnih primerih z našega tekmovanja bi ta rešitev dobila 50 % točk.

Boljša rešitev je, da opazimo, da lahko do spremembe v številu obiskovalcev pride le takrat, ko kak obiskovalec pride v fitnes ali pa ga zapusti — torej le ob časih s_i in e_i za obiskovalce iz trenutne skupine (torej tiste, ki jih zanima trenutna naprava). Pripravimo si torej seznam parov $(s_i, +1)$ in $(e_i, -1)$, ki nam povedo, kdaj se število obiskovalcev spremeni in za koliko. Te pare uredimo po času, če pa jih je več ob istem času, jih uredimo še po drugi komponenti, tako da odhajajoče obiskovalce obdelamo prej kot prihajajoče (saj naloga pravi, da če v istem trenutku en obiskovalec odide, drugi pa pride, bosta lahko uporabila isto napravo). Nato se moramo le sprehoditi po parih v tem vrstnem redu in sproti popravljati število obiskovalcev; največje število obiskovalcev, ki ga na ta način dobimo, si zapomnimo (po vsakem povečanju preverimo, če smo presegli dosedanji maksimum) in ga na koncu izpišimo. Časovna zahtevnost te rešitve je $O(k \log k)$.

Namesto da najprej urejamo obiskovalce po številki naprave in nato pri vsaki skupini posebej pripravljamo in urejamo pare $(s_i, +1)$ in $(e_i, -1)$, lahko oboje skupaj naredimo v enem zamahu: za vsakega obiskovalca pripravimo dve trojici $(h_i, s_i, +1)$ in $(h_i, e_i, -1)$ in uredimo seznam takšnih trojic.

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  // Preberimo vhodne podatke in pripravimo seznam prihodov
  // in odhodov vseh obiskovalcev.
```

```
  int k; scanf("%d", &k);
  struct Krajisce { int cas, delta, naprava; };
  vector<Krajisce> v; v.reserve(2 * k);
  for (int i = 0; i < k; i++)
  {
    int prihod, odhod, naprava; scanf("%d %d %d", &prihod, &odhod, &naprava);
    v.push_back({prihod, 1, naprava}); v.push_back({odhod, -1, naprava});
  }
  // Seznam uredimo po napravi, zapise z isto napravo po času,
  // tiste z istim časom pa tako, da pridejo odhodi pred prihodi.
  sort(v.begin(), v.end(), [] (const auto & x, const auto & y) {
    int r = x.naprava - y.naprava; if (r != 0) return r < 0;
    r = x.cas - y.cas; if (r != 0) return r < 0; else return x.delta < y.delta; });
  // Preglejmo seznam in izpišimo rezultate.
  for (int i = 0; i < v.size(); )
  {
    int naprava = v[i].naprava, maxHkrati = 0;
    // Preglejmo vse zapise za trenutno napravo in štejmo,
    // koliko jih je hkrati zasedenih. Maksimum tega si zapomnimo.
    for (int hkrati = 0; i < v.size() && v[i].naprava == naprava; i++) {
      hkrati += v[i].delta; maxHkrati = max(maxHkrati, hkrati); }
    printf("%d %d\n", naprava, maxHkrati);
  }
  return 0;
}
```

2. Telefonsko omrežje

Preprosta rešitev je, da gremo v zanki po vseh oddajnikih in pri vsakem oddajniku še z dvema gnezdenima zankama po vseh poljih, ki jih pokriva. Vsa tako pokrita polja dodajajmo v razpršeno tabelo ali kakšno podobno podatkovno strukturo, v kateri je posamezno polje lahko prisotno največ enkrat. Na koncu nam število elementov v tej strukturi pove, koliko polj je pokritih, ravno to pa nas zanima. Slabost te rešitve je, da oddajnik z močjo r pokrije $r^2+(r+1)^2$ polj, torej je skupno število pokritih polj v najslabšem primeru $O(k \cdot r^2)$. Takšni sta zato tudi časovna in prostorska zahtevnost te rešitve. Z njo bi uspešno rešili manjše testne primere (na našem tekmovanju bi dobili 35 % točk), pri večjih pa bi ji zmanjkalo pomnilnika za shranjevanje vseh pokritih polj (prej ali slej bi z naštevanjem vseh pokritih polj prekoračili tudi časovno omejitev).

Boljša rešitev je na primer ta, da območje v obliki kara ($\diamond$), ki ga pokriva posamezni oddajnik, v mislih razrežemo po vrsticah. Če stoji oddajnik v točki (a_i, b_i) in ima moč r_i, nam tako nastane $2r_i + 1$ vodoravnih blokov (v vrsticah od $b_i - r_i$ do $b_i + r_i$). Dolžine teh blokov počasi naraščajo od 1 do $2r_i + 1$ in nato spet padajo proti 1. V isti vrstici mreže so seveda lahko prisotni bloki različnih oddajnikov in lahko se med seboj tudi prekrivajo; poskrbeti moramo, da območij, kjer se bloki prekrivajo, ne bomo šteli po večkrat.

Uporabimo lahko zelo podoben prijem kot pri prejšnji nalogi; recimo, da nam posamezni oddajnik i v trenutni vrstici prispeva blok od $x = \ell_i$ do $x = r_i$. Če se v mislih premikamo po trenutni vrstici od leve proti desni, bo do spremembe v številu oddajnikov, ki pokrivajo neko polje, prišlo (zaradi bloka i) pri $x = \ell_i$

(kjer se pokritost poveča za 1) in pri $x = r_i + 1$ (kjer se pokritost zmanjša za 1). Pripravimo si torej seznam parov oblike $(\ell_i, +1)$ in $(r_i, -1)$, jih uredimo in jih v tem vrstnem redu preglejmo. Pri vsaki spremembi poglejmo, če je bila dosedanja pokritost (pred spremembo) večja od 0; če je bila, potem vemo, da je območje od prejšnje do trenutne spremembe res pokrito in da moramo polja na tem območju prišteti k rezultatu, po katerem sprašuje naloga.

Vprašanje je še, kako vedeti, s katerimi vrsticami se moramo ukvarjati in kateri oddajniki pridejo v poštev pri posamezni vrstici. Spet lahko uporabimo podoben razmislek kot v prešnjem odstavku: če se počasi premikamo po mreži od manjših y proti večjim, lahko do spremembe pri tem, kateri oddajniki so prisotni v trenutni vrstici, pride le pri y-koordinatah oblike $b_i - r_i$ (kjer se začne oddajnik i) in $b_i + r_i + 1$ (kjer oddajnika i ni več). Lahko si torej spet pripravimo pare $(b_i - r_i, +1)$ in $(b_i + r_i + 1, -1)$ in jih uredimo. Pri vsaki y-koordinati, kjer pride do spremembe, bi lahko preprosto ponovno pregledali vse oddajnike, da vidimo, kateri so prisotni v trenutni vrstici; lahko pa vzdržujemo seznam aktivnih oddajnikov in vanj dodamo oz. iz njega pobrišemo le tistega, zaradi katerega se s trenutno spremembo sploh ukvarjamo.[2]

Kakšna je cena tega postopka? Recimo, da je v vrstici y prisotnih n_y oddajnikov; pri tej vrstici imamo torej $O(n_y \log n_y)$ dela (ker moramo urediti $2n_y$ parov). Ker imamo k oddajnikov in je vsak prisoten v $O(r)$ vrsticah, je $\sum_y n_y = O(kr)$. Skupni čas obdelave vseh vrstic je potem reda $\sum_y n_y \log n_y \le \sum_y n_y \log k = O(kr \log k)$. S tem lahko dovolj hitro rešimo vse testne primere na našem tekmovanju.[3]

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

struct Krajisce { int koord, delta, i; };
bool operator < (const Krajisce &u, const Krajisce &v) {
  return (u.koord == v.koord) ? u.delta < v.delta : u.koord < v.koord; }

int main()
{
  int k; scanf("%d", &k);
  // Preberimo podatke o oddajnikih in pripravimo v yKraj seznam
  // njihovih začetnih in končnih y-koordinat.
  struct Oddajnik { int x, y, r, kje = -1; };
  vector<Oddajnik> odd(k);
  vector<Krajisce> xKraj(2 * k), yKraj; yKraj.reserve(2 * k);
  for (int i = 0; i < k; i++)
  {
    auto &O = odd[i]; scanf("%d %d %d", &O.x, &O.y, &O.r);
    yKraj.push_back({O.y - O.r, 1, i}); yKraj.push_back({O.y + O.r + 1, -1, i});
```

[2] Prva od teh dveh možnosti nam vzame skupno $O(k^2)$ časa (ker pride do sprememb pri $O(k)$ vrsticah in moramo pri vsaki na novo pregledati vseh k oddajnikov), druga pa le $O(k \log k)$ (za urejanje parov, nato pa še $O(1)$ za vsako dodajanje in brisanje v seznamu aktivnih oddajnikov, kar bo dalo $O(k)$ za vse spremembe skupaj). Kot bomo videli v naslednjem odstavku, se cena tega tako ali tako utopi v ceni glavnega dela postopka, torej pregledovanju vsake vrstice.

[3] Za srednje velike testne primere bi bila dovolj dobra tudi malo preprostejša različica te rešitve, ki bi šla pri vsaki vrstici po vseh oddajnikih in ugotavljala, kateri so prisotni v njej. Taka rešitev s časovno zahtevnostjo $O(k^2 r)$ bi dobila na našem tekmovanju 65 % točk.

```
  }

  // Pregledujmo ravnino po naraščajočih y-koordinatah in vzdržujmo
  // seznam trenutno odprtih oddajnikov (to so tisti, ki so prisotni na
  // trenutni y-koordinati). Velja odd[odprti[j]].kje = j.
  vector<int> odprti(k, -1); int stOdprtih = 0;
  long long stPokritih = 0;
  sort(yKraj.begin(), yKraj.end());
  for (int iy = 0; iy < yKraj.size(); iy++)
  {
    const auto &yk = yKraj[iy];

    if (yk.delta > 0) {
      // Pri y-koordinati yk.koord se začne območje, ki ga pokriva oddajnik yk.i;
      // dodajmo ga na konec seznama odprtih oddajnikov.
      odd[yk.i].kje = stOdprtih;
      odprti[stOdprtih++] = yk.i; }

    else {
      // Pri y-koordinati yk.koord se začne območje, ki ga ne oddajnik yk.i ne pokriva več;
      // pobrišimo ga iz seznama odprtih oddajnikov, na njegovo mesto pa dodajmo tistega
      // s konca seznama (recimo mu z).
      int kje = odd[yk.i].kje, z = odprti[--stOdprtih];
      odprti[kje] = z; odd[z].kje = kje; odd[yk.i].kje = -1; }

    if (stOdprtih == 0 || yk.koord == yKraj[iy + 1].koord) continue;

    // Če so med trenutno y-koordinato in tisto, pri kateri pride do naslednje spremembe,
    // odprti kakšni oddajniki, preglejmo vse vrstice na tem območju in štejmo pokrita polja.
    for (int y = yk.koord; y < yKraj[iy + 1].koord; y++)
    {
      // Za vsak odprti oddajnik poglejmo, pri katerem x se začne in konča območje,
      // ki ga on pokriva pri trenutni y-koordinati.
      for (int i = 0; i < stOdprtih; i++)
      {
        const auto &O = odd[odprti[i]];
        int dx = O.r - abs(O.y - y);
        xKraj[2 * i] = { O.x - dx, 1, -1 };
        xKraj[2 * i + 1] = { O.x + dx + 1, -1, -1 };
      }

      // Preglejmo te x-koordinate naraščajoče in štejmo pokrita polja.
      sort(xKraj.begin(), xKraj.begin() + 2 * stOdprtih);
      int xPrej, pokritost = 0;
      for (int i = 0; i < 2 * stOdprtih; i++)
      {
        const auto &K = xKraj[i];

        // Polja od xPrej do K.koord - 1 pokriva „pokritost“ oddajnikov.
        if (pokritost > 0) stPokritih += K.koord - xPrej;

        // Pri K.koord se pokritost spremeni.
        pokritost += K.delta; xPrej = K.koord;
      }
    }
  }

  // Izpišimo rezultat.
  printf("%lld\n", stPokritih); return 0;
}
```

Še boljšo rešitev pa dobimo, če mrežo v mislih pobarvamo kot šahovnico in jo zasu-

kamo za 45 stopinj. Območje, ki ga pokriva oddajnik z močjo r, je zdaj pravzaprav videti kot kvadrat — če gledamo na naši šahovnici črna polja posebej in bela posebej, vidimo, da naš oddajnik pokriva en bel kvadrat in en črn kvadrat; eden od teh dveh kvadratov je velik $r \times r$ polj, drugi pa $(r+1) \times (r+1)$ polj (manjši je tisti, čigar barva je enaka barvi polja, na katerem stoji oddajnik). Naloga zdaj pravzaprav sprašuje, kakšna je ploščina unije vseh teh kvadratov, to pa je znan problem s področja računske geometrije, ki ga lahko rešimo s preletom ravnine (*plane sweep*) v $O(k \log k)$ časa (posebej za črne in posebej za bele kvadrate, nato pa obe ploščini seštejmo).[4] Lepo pri tej rešitvi je, da je odvisna le še od števila oddajnikov k, ne pa tudi od njihove moči r.

3. Transakcijski računi

Če premešamo vrstni red računov v seznamu nakazil, ostane kontrolna številka seznama nespremenjena. Zato je za naš namen pomembno predvsem, da se odločimo, pri koliko nakazilih bi vpisali račun dobrodelne organizacije; vprašanje, katera nakazila točno bi to bila, je potem trivialno — k dobrodelni organizaciji usmerimo pač nakazila z največjimi zneski, manjša nakazila pa k drugim računom, ki jih potrebujemo, da dosežemo želeno kontrolno številko.

Opazimo lahko, da na kontrolno številko seznama vplivajo le kontrolne številke računov v njem — vse, kar v številki računa stoji pred kontrolno številko, lahko ignoriramo. Naj bo g kontrolna številka prvotnega seznama nakazil, kot smo ga dobili v vhodnih podatkih. Recimo, da ima račun dobrodelne organizacije kontrolno številko c; če ga vpišemo v k nakazil, bo to h kontrolni vsoti seznama prispevalo $(k \cdot c) \bmod 10$, do g pa nam torej manjka še $(g - k \cdot c) \bmod 10$.[5] Vprašanje je torej, ali lahko v preostalih $m - k$ nakazil vpišemo druge račune iz baze tako, da bodo njihove številke skupaj dale vsoto $(g - k \cdot c) \bmod 10$. Poiskati moramo največji k, pri katerem je to mogoče.

Kot smo že videli, je dovolj, če od vsakega računa obdržimo le njegovo kontrolno številko; potem tudi ni koristi od tega, da bi imeli več različnih računov z enako kontrolno številko; vse, kar si moramo torej od prvotne baze računov (če odmislimo račun dobrodelne organizacije) zapomniti, je to, katere številke od 0 do 9 se pojavljajo kot kontrolne številke kakšnega računa v bazi. Tej množici recimo D; množici kontrolnih vsot, ki jih je mogoče dobiti, če zapolnimo t nakazil z računi iz baze (brez računa dobrodelne organizacije), pa recimo D_t. Pri $t = 0$ sploh ni nobenega nakazila in je kontrolna vsota lahko le 0, torej $D_0 = \{0\}$. Pri večjih t pa lahko vsako kontrolno vsoto, ki je dosegljiva v $t-1$ korakih, dopolnimo z vsako kontrolno številko iz D; tako lahko zapišemo zvezo $D_t = \{(a+b) \bmod 10 : a \in D_{t-1}, b \in D\}$.

Tako smo prišli do naslednje rešitve: v zanki računajmo D_t za vse večje t, pri vsakem pa nato poglejmo, če je $(g-(m-t)\cdot c) \bmod 10 \in D_t$. Čim je ta pogoj izpolnjen, vemo, da smo našli najboljšo rešitev: račun dobrodelne organizacije moramo vpisati v $m - t$ največjih nakazil na seznamu. Seznam lahko preprosto uredimo po zneskih

[4] S tem problemom smo se na naših tekmovanjih že srečali; gl. npr. nalogo 1998.2.3 (na str. 345 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004*, rešitev pa je na str. 355–61).

[5] Da ne bo težav z negativnimi števili, je to v praksi varneje računati kot $(g + 10 - (k \cdot c) \bmod 10) \bmod 10$.

in seštejemo $m-t$ največjih; taka rešitev bi imela časovno zahtevnost $O(m \log m)$ in bi bila za naše namene čisto dovolj dobra. Še bolje pa je, če uporabimo katerega od postopkov, ki poiščejo največjih $m - t$ elementov v $O(m)$ časa (npr. quickselect ali pa mediana median; v C++ tako načeloma deluje funkcija nth_element iz standardne knjižnice).

Oglejmo si implementacijo tega postopka v C++. Za predstavitev množic D in D_t lahko uporabimo kar celoštevilske spremenljivke, v katerih spodnjih 10 bitov pove, katere števke od 0 do 9 so prisotne v množici.

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  int n, m, nas, drugi1 = 0;
  scanf("%d %d\n", &n, &m);

  // Preberimo seznam računov.
  for (int i = 0; i < n; i++) {
    char s[12]; scanf("%11s\n", s);
    if (i == 0) nas = s[10] - '0';
    else drugi1 |= 1 << (s[10] - '0'); }

  // Preberimo seznam nakazil.
  vector<int> zneski(m);
  int vsota = 0; // To bo kontrolna vsota celega seznama.
  for (int i = 0; i < m; i++) {
    char s[12]; scanf("%11s %d\n", s, &zneski[i]);
    vsota += s[10] - '0'; }
  vsota %= 10;

  // Poglejmo, kolikokrat lahko uporabimo naš račun, da dosežemo želeno vsoto.
  int stNasih = m;    // Število nakazil na naš račun.
  int drugi = 1;      // Vsote, dosegljive z (m - stNasih) uporabami drugih računov.
  while (true)
  {
    // Koliko moramo sestaviti z ostalimi računi, če naš račun uporabimo stNasih-krat?
    int razlika = (vsota + 10 - (nas * stNasih) % 10) % 10;

    // Ali to razliko lahko sestavimo z ostalimi računi?
    if ((drugi >> razlika) & 1) break;

    // Če ne, poskusimo število nakazil na naš račun zmanjšati.
    if (--stNasih < 0) break;

    // Katere vsote lahko sestavimo z eno dodatno uporabo drugih računov?
    int noviDrugi = 0;
    for (int a = 0; a < 10; a++) if ((drugi >> a) & 1)
      noviDrugi |= (drugi1 << a);
    drugi = (noviDrugi & 1023) | (noviDrugi >> 10);
  }

  // Uporabili bomo račune z največjim zneskom.
  nth_element(zneski.begin(), zneski.begin() + (m - stNasih), zneski.end()); // O(m)
  long long rezultat = 0;
  for (int i = m - stNasih; i < m; i++) rezultat += zneski[i];
  printf("%lld\n", rezultat); return 0;
}
```

4. Nadzor

Nalogo lahko rešujemo z dinamičnim programiranjem. Uredimo kamere naraščajoče po b_i, torej po desni koordinati intervala, ki ga pokrivajo. Kot zadnjo kamero (tisto z največjim indeksom) v našem izboru bomo morali uporabiti eno od tistih, ki imajo desno krajišče pri d (torej $b_i = d$). Ta kamera nam torej pokrije interval $[a_i, b_i]$ oz. $[a_i, d]$, ostane pa nam vprašanje, kako pokriti preostanek ulice, torej interval $[0, a_i]$. Poiščimo prvo tako kamero j, ki ima $b_j \geq a_i$. Če bi poleg kamere i uporabili v našem izboru le kamere $1, \ldots, j-1$, ulica ne bi bila v celoti pokrita, ker se vse tiste kamere končajo prej, preden se kamera i začne. Nujno moramo torej poleg kamere i uporabiti še eno od kamer $j, \ldots, i-1$ (med vsemi temi možnostmi bomo seveda uporabili tisto, ki nam bo dala najmanjšo skupno ceno). Pri tisti kameri lahko potem na podoben način razmišljamo o tem, kako pokriti območje levo od nje.

Naš razmislek lahko povzamemo takole: naj bo $f(i)$ cena najcenejšega takega nabora kamer, ki vsebuje kamero i, ne vsebuje kamer $i+1, \ldots, n$ in v celoti pokrije območje $[0, b_i]$. Kot smo videli v prejšnjem odstavku, je

$$f(i) = c_i + \max_j\{f(j) \ : \ 1 \leq j < i \text{ in } b_j \geq a_i\}.$$

Rezultat, po katerem sprašuje naloga, pa je potem $\max_i\{f(i) \ : \ b_i = d\}$.

Vrednosti funkcije f je koristno računati po naraščajočih i in si jih sproti shranjevati v tabelo, kjer nam bodo pri roki, ko jih bomo kasneje spet potrebovali. Tega postopka ne bi bilo težko zapisati z dvema gnezdenima zankama, zunanji po i in notranji po j:

```
for (int i = 0; i < n; i++) {
  int m = ∞;
  for (int j = i − 1; j >= i && b[j] >= a[i]; j−−) m = min(m, f[j]);
  f[i] = c[i] + m; }
```

V najslabšem primeru ima ta postopek časovno zahtevnost $O(n^2)$, kar bi bilo za večje testne primere že prepočasi.

Najmanjši primerni j lahko določimo tako, da v zaporedju $b_1, \ldots, b_n$ z bisekcijo (binarnim iskanjem) poiščemo najmanjši člen, ki je $\geq a_i$. Za to porabimo $O(\log n)$ časa; še vedno pa ostane vprašanje, kako učinkovito poiskati minimum vrednosti $f[j], f[j+1], \ldots, f[i-1]$. V ta namen moramo tako ali drugače nekje vzdrževati minimume po več zaporednih elementov tabele f, da nam kasneje ne bo treba iti v zanki od j do $i-1$ in pregledovati vsakega posebej. Preden se začnemo ukvarjati s podrobnostmi tega, zapišimo glavni del našega programa:

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

const long long int Inf = 1000000000000001LL;
struct Rezultati { ... }; // To si bomo ogledali malo kasneje.

int main()
{
  // Preberimo vhodne podatke.
```

```
  int n, d; scanf("%d %d", &d, &n);
  struct Kamera { int a, b, c; };
  vector<Kamera> K(n); for (auto &k : K) scanf("%d %d %d", &k.a, &k.b, &k.c);

  // Kamere uredimo po desnem krajišču.
  sort(K.begin(), K.end(), [] (const auto &x, const auto &y) { return x.b < y.b; });

  Rezultati rez {n}; // Podatkovna struktura, ki bo hranila že izračunane vrednosti f(i).
  long long najboljsa = Inf; // Najboljša rešitev, ki pokrije celo daljico.

  for (int i = 0; i < n; i++)
  {
    const int a = K[i].a, b = K[i].b;

    // Kakšna je najnižja cena, da pokrijemo inteval [0, b],
    // če uporabimo kamero i in nobene od kamer i + 1, ..., n - 1?
    // Poiščimo prvo kamero, ki ima desno krajišče ≥ a.
    auto p = upper_bound(K.begin(), K.end(), a - 1,
                         [] (int x, const auto &y) { return x < y.b; });
    int j = p - K.begin();

    // Da pokrijemo območje levo od a, nas stane min { f[j], ..., f[i - 1] }.
    long long cena = K[i].c + (a == 0 ? 0 : rez.Minimum(j, i));

    // Shranimo vrednost f(i).
    rez.Vpisi(i, cena);
    if (b == d) najboljsa = min(najboljsa, cena);
  }

  printf("%lld\n", najboljsa); return 0;
}
```

Razmislimo zdaj o tem, kako implementirati strukturo Rezultati z operacijama Minimum(j, i), ki mora vrniti $\min\{f(j), \ldots, f(i-1)\}$, in Vpisi(i, x), ki mora shraniti podatek, da je $f(i) = x$.

Preprosta možnost je na primer ta, da tabelo f v mislih razdelimo na „bloke“ s po B elementi in v neki ločeni tabeli hranimo minimum vsakega bloka (te minimume po potrebi tudi sproti popravljamo, ko računamo nove vrednosti $f[i]$). Ko nas potem zanima minimum f za območje od j do $i-1$, lahko uporabimo shranjene minimume tistih blokov, ki v celoti ležijo na tem območju; posamezne elemente tabele f pa moramo pregledati le na začetku in na koncu območja, kjer po en blok na vsaki strani leži na našem območju le delno. Če za B vzamemo približno $\sqrt{n}$, imamo tudi približno $\sqrt{n}$ blokov in časovna zahtevnost takšnega računanja minimuma od j do $i-1$ je $O(\sqrt{n})$, časovna zahtevnost celotne rešitve pa $O(n\sqrt{n})$. Za naše testne primere je bilo to že dovolj hitro.

```
struct Rezultati
{
  int n, B;
  // fb[i] = minimum vrednosti f[j] za B * i ≤ j < B * (i + 1).
  vector<long long> f, fb;

  Rezultati(int N)
  {
    n = N; B = 1; while (B * B < n) B++;
    f.resize(n, Inf); fb.resize(n / B + 1, Inf);
  }

  long long Minimum(int od, int doPred)
```

```
  {
    long long r = Inf;
    // Preglejmo delno pokriti blok na začetku območja [od, doPred).
    while (od < doPred && (od % B) != 0) r = min(r, f[od++]);
    // Preglejmo delno pokriti blok na koncu območja [od, doPred).
    while (od < doPred && (doPred % B) != 0) r = min(r, f[--doPred]);
    // Preglejmo minimume vmesnih blokov, ki so pokriti v celoti.
    od /= B; doPred /= B;
    while (od < doPred) r = min(r, fb[od++]);
    return r;
  }

  void Vpisi(int i, long long x)
  {
    f[i] = min(x, f[i]);                // vpišimo x kot novo vrednost f[i]
    fb[i / B] = min(x, fb[i / B]);      // minimum bloka, v katerem leži i
  }
};
```

Še boljša rešitev je binarno drevo. Nad prvotno tabelo f vzdržujmo še več manjših tabel f_k za $k = 1, \ldots, \lfloor \log_2 n \rfloor$, pri čemer je tabela f_k dolga $\lfloor n/2^k \rfloor$ elementov in v njej $f_k[i]$ vsebuje minimum vrednosti $f(t)$ za $i \cdot 2^k \leq t < (i+1) \cdot 2^k$. Z drugimi besedami je torej $f_k[i] = \min\{f_{k-1}[2i], f_{k-1}[2i+1]\}$. Ko izračunamo vrednost $f(i)$, moramo na vsakem nivoju (pri vsakem k) na novo izračunati en element (in sicer $f_k[\lfloor i/2^k \rfloor]$). Ko pa iščemo minimum $f(j), \ldots, f(i-1)$, moramo na vsakem nivoju pogledati največ dva elementa (kajti če bi hoteli pogledati tri zaporedne, bi lahko namesto dveh izmed njih že uporabili enega na naslednjem višjem nivoju, ki vsebuje ravno njun minimum). Tako za računanje minimuma kot za vpis nove vrednosti $f(i)$ zdaj porabimo $O(\log n)$ časa, časovna zahtevnost celotne rešitve pa je zato $O(n \log n)$.

Pri implementaciji takšne rešitve niti ni nujno imeti več tabel; lahko jih vse zložimo eno za drugo v isti vektor, ki že hrani tabelo f. Vse f_k skupaj nimajo več kot n elementov, zato bo dovolj, če pri vektorju f zdaj alociramo za $2n$ elementov prostora.

```
struct Rezultati
{
  int n; vector<long long> f;
  Rezultati(int N) : n(N), f(2 * N, Inf) { }

  long long Minimum(int od, int doPred)
  {
    long long r = Inf;
    auto nivo = f.begin(); int dNivoja = n;
    while (od < doPred)
    {
      // „nivo“ kaže na začetek trenutnega nivoja (ene od tabel f_k).
      // Na tem nivoju je „dNivoja“ elementov, nas pa zanima minimum tistih
      // na indeksih od „od“ do „doPred − 1“. Toda če taki indeksi nastopajo
      // v parih oblike 2t in 2t + 1, lahko primeren minimum dobimo na naslednjem
      // nivoju, zato bomo zdaj pogledali le prvi in zadnji indeks na trenutnem
      // nivoju, ki mogoče v takšnih parih ne nastopata.
      if (od & 1) r = min(r, nivo[od++]);
```

```
      if (doPred & 1) r = min(r, nivo[--doPred]);
      nivo += dNivoja; dNivoja /= 2; od /= 2; doPred /= 2;
    }
    return r;
  }

  void Vpisi(int i, long long x)
  {
    auto nivo = f.begin(); int dNivoja = n;
    while (i < dNivoja)
    {
      nivo[i] = min(nivo[i], x);
      nivo += dNivoja; dNivoja /= 2; i /= 2;
    }
  }
};
```

Zelo elegantna možnost pa je tudi Fenwickovo drevo. Običajna različica tega drevesa je namenjena temu, da lahko nad tabelo $u[1..n]$ učinkovito (v logaritemskem času) računamo delne vsote oblike $u[1] + \ldots + u[k]$, pri čemer lahko elemente tabele u tudi spreminjamo. To dosežemo tako, da namesto prvotne tabele u vzdržujemo (enako veliko) tabelo U, v kateri $U[i]$ vsebuje vsoto členov $u[j]$ za $g(i) < j \leq i$, pri čemer je $g(i)$ število, ki ga dobimo, če v dvojiškem zapisu števila i ugasnemo najnižji prižgani bit. Ko nas zanima vsota $u[1] + \ldots + u[k]$, se lahko hitro prepričamo, da jo lahko izračunamo kot $U[k] + U[g(k)] + U[g(g(k))] + \ldots + U[0]$ (na koncu si mislimo $U[0] = 0$). Ko pa se neka vrednost $u[k]$ spremeni (recimo na $u[k] + \triangle$), moramo za enako $\triangle$ spremeniti tudi vse tiste elemente $U[i]$, pri katerih k leži na območju $g(i) < k \leq i$. Obe operaciji je mogoče izvesti v $O(\log n)$ časa.

Takšno drevo lahko čisto dobro uporabimo tudi za delo z minimumi namesto z vsotami, pomembno je le, da se elementi osnovne tabele u le zmanjšujejo, ne pa povečujejo. Tako bi lahko poceni izračunali minimum prvih nekaj elementov tabele u, od indeksa 1 naprej. Toda spomnimo se, da bodo nas v resnici zanimali minimumi oblike $\min\{f(j), \ldots, f(i-1)\}$, torej se ne začnejo na začetku tabele, ampak pri j, ki je lahko tudi večji od 1. Tej težavi se lahko izognemo tako, da osnovno tabelo obrnemo: vrednost $f(i)$ bomo hranili v $u[n+1-i]$; na začetku inicializirajmo vse elemente tabele u na $+\infty$, ko pa izračunamo pravo vrednost $f(i)$, jo vpišimo v $u[n+1-i]$ in ustrezno popravimo tabelo U. Če zdaj izračunamo $\min\{u[1], \ldots, u[n+1-j]\}$, je to isto kot $\min\{f(j), \ldots, f(n)\}$, kar pa je enako $\min\{f(j), \ldots, f(i-1)\}$, saj vrednosti $f(i), \ldots, f(n)$ takrat še nismo izračunali in so pripadajoči elementi tabele u takrat še enaki $+\infty$ in na minimum nič ne vplivajo.

V spodnji implementaciji sicer namesto $n+1-i$ ali $n+1-j$ uporabljamo $n-i$ oz. $n-j$, ker gredo indeksi, ki jih metodi Minimum in Vpisi dobivata kot parametre, od 0 do $n-1$ namesto od 1 do n.

```
struct Rezultati
{
  int n; vector<long long> U;
  Rezultati(int N) : n(N), U(N + 1, Inf) { }

  long long Minimum(int od, int doPred)
  {
    long long r = Inf;
```

```
    for (int k = n − od; k > 0; k &= k − 1) r = min(r, U[k]);
    return r;
  }

  void Vpisi(int i, long long x)
  {
    for (int k = n − i; k <= n; k += k & ∼(k − 1)) U[k] = min(U[k], x);
  }
};
```

5. Detektorji

Možne poti lopova in skupno vrednost nakradenega plena je koristno pregledovati po naraščajoči dolžini (torej po naraščajočem številu časovnih intervalov). Pri vsakem trezorju v si zapomnimo še največji znesek, ki ga lahko lopov nakrade od začetka noči do trenutnega časovnega intervala t, če se trenutno nahaja v tistem trezorju; temu znesku recimo $f_t(v)$.

Lopov se lahko ob času t nahaja v trezorju v, če sta izpolnjena naslednja dva pogoja:

1. da vsaj polovica detektorjev od s_v do e_v v trenutnem časovnem intervalu zaznava prisotnost lopova; in
2. da se je ob času $t-1$ nahajal v v ali pa v nekem takem trezorju u, za katerega obstaja hodnik med u in v.

Na začetku noči, v prvem časovnem intervalu, drugi od teh dveh pogojev ne pride v poštev, saj naloga pravi, da lahko lopov svojo pot začne v kateremkoli trezorju. Kasneje pa, če je mogoče v v priti v enem koraku iz več sosednjih trezorjev u, nas bo med njimi zanimal seveda tisti, ki dá največji skupni nakradeni znesek. Tako dobimo

$$f_t(v) = \max_u\{f_{t-1}(u) \ : \ \text{obstaja hodnik med } u \text{ in } v\} + w_v,$$

če je izpolnjen prvi pogoj (da dovolj detektorjev zaznava prisotnost); če pa ni izpolnjen, si mislimo $f_t(v) = -\infty$.

Funkcijo f torej lahko računamo po naraščajočih t, pri vsakem t gremo z zanko po v, pri vsakem v pa z zanko po njegovih sosedih u (torej tistih trezorjih, ki jih hodniki neposredno povezujejo z v). Drobna izboljšava pa je, da upoštevamo, da če je bil nek v-jev sosed u ob času $t-1$ nedosegljiv (torej če je $f_{t-1}(u) = -\infty$), potem tudi k dosegljivosti v-ja ob času t ne bo mogel prispevati. Zato je bolje imeti zanko po u; pri vsakem najprej preverimo, če je $f_{t-1}(u) \neq -\infty$, in šele če je ta pogoj izpolnjen, pojdimo z zanko po njegovih sosedih v in poglejmo, če lahko vrednost $f_{t-1}(u)$ kaj pripomore k maksimumu, ki ga računamo za $f_t(v)$. Pri obeh postopkih pa imamo s tem pri vsakem časovnem intervalu $O(n + m)$ dela, ker moramo v najslabšem primeru pregledati vse trezorje in vse hodnike.

Ostane še vprašanje, kako bi učinkovito preverili prvi pogoj, torej ali v trenutnem časovnem intervalu zaznava prisotnost vsaj polovica izmed detektorjev $s_v, \ldots, e_v$. Vhodni podatki nam povedo spremembe v stanju detektorjev; s pomočjo teh sprememb seveda ni težko vzdrževati tabele, ki nam bo za vsak detektor povedala njegovo trenutno stanje. Naj bo recimo $a[i] = 1$, če detektor i trenutno zaznava prisotnost, in $a[i] = 0$, če je ne zaznava. Zdaj nas torej pravzaprav zanima, ali je

$a[s_v] + \ldots + a[e_v] \geq (e_v - s_v + 1)/2$. Če bomo to vsoto računali z zanko po vseh detektorjih od s_v do e_v, nam bo to vzelo $O(d)$ časa, in ker moramo to narediti za vsak trezor pri vsakem časovnem koraku, bomo za to skupaj porabili $O(ndq)$ časa. To bi bilo dovolj dobro za manjše testne primere (na našem tekmovanju bi takšna rešitev dobila polovico točk), za večje pa je že prepočasi.

Boljša rešitev je, da si pripravimo tabelo delnih vsot: $b[0] = 0$ in (za $i \geq 1$) $b[i] = b[i-1] + a[i]$. Tako nam torej $b[i]$ pove, koliko izmed prvih i detektorjev trenutno zaznava prisotnost; razlika $b[e_v] - b[s_v - 1]$ pa nam pove, koliko izmed detektorjev od s_v do e_v zaznava prisotnost — prav to pa nas zanima, ko preverjamo, ali bi lopov lahko trenutno bil v trezorju v. Pri vsakem časovnem intervalu torej porabimo $O(d)$ časa za izračun tabele delnih vsot, nato pa imamo pri vsakem trezorju le $O(1)$ dela s preverjanjem, ali dovolj njegovih detektorjev zaznava prisotnost.

Vsega skupaj je torej časovna zahtevnost naše rešitve $O(q \cdot (n + m + d))$, kar je za naše potrebe že dovolj dobro. Oglejmo si implementacijo takšne rešitve v C++:

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

struct Trezor
{
  int s, e;      // prvi in zadnji detektor, ki ga pokrivata
  int w;         // znesek, ki ga lopov nakrade tu v enem intervalu
  int f = 0;     // vrednost najboljše poti s koncem v tem trezorju
  int fp;        // f iz prejšnjega časovnega intervala
  vector<int> sosedje;
};

int main()
{
  int n, m, d, q; scanf("%d %d %d %d", &n, &m, &d, &q);
  vector<Trezor> trezorji(n);

  // Preberimo podatke o hodnikih in pripravimo za vsak trezor seznam sosedov.
  for (int i = 0; i < m; i++)
  {
    int ai, bi; scanf("%d %d", &ai, &bi); ai--; bi--;
    auto &sa = trezorji[ai].sosedje, &sb = trezorji[bi].sosedje;
    trezorji[ai].sosedje.push_back(bi); trezorji[bi].sosedje.push_back(ai);
  }

  // Preberimo podatke o trezorjih.
  for (auto &V : trezorji) scanf("%d %d %d", &V.w, &V.s, &V.e);

  // Preberimo podatke o detektorjih in izračunajmo rezultat.
  // stanje[i] je stanje i-tega detektorja (0 ali 1);
  // vsote[i] je vsota stanj detektorjev od 0 do i - 1.
  vector<int> stanje(d, 0), vsote(d + 1, 0);
  for (int cas = 0; cas < q; cas++)
  {
    // Preberimo spremembe in popravimo stanja detektorjev.
    int nSprememb; scanf("%d", &nSprememb);
    while (nSprememb-- > 0) {
      int i; scanf("%d", &i); i--;
      stanje[i] = 1 - stanje[i]; }
```

```
    // Na novo izračunajmo delne vsote.
    for (int i = 0; i < d; i++) vsote[i + 1] = vsote[i] + stanje[i];
    // Rezultate iz prejšnjega časovnega intervala preselimo iz f v fp.
    for (auto &t : trezorji) t.fp = t.f;
    // Poti iz prejšnjega intervala podaljšajmo za en korak.
    for (const auto &U : trezorji) if (U.fp >= 0)
      for (int v : U.sosedje) { auto &V = trezorji[v]; V.f = max(V.f, U.fp); }
    // Za vsak trezor poglejmo, če v njem vsaj polovica trezorjev zaznava gibanje.
    for (auto &V : trezorji)
      if (2 * (vsote[V.e] - vsote[V.s - 1]) < V.e - V.s + 1) V.f = -1;
      else if (V.f >= 0) V.f += V.w;
  }
  // Izpišimo največji možni nakradeni znesek.
  int naj = -1; for (const auto &V : trezorji) naj = max(naj, V.f);
  printf("%d\n", naj); return 0;
}
```

To rešitev bi se dalo na razne načine še izboljšati. Lahko bi na primer vzdrževali množico trezorjev, ki so v danem trenutku sploh dosegljivi, torej $D_t := \{v : f_t(v) \neq -\infty\}$. Pri računanju D_t in f_t potem ni treba iti z u po vseh trezorjih, ampak je dovolj že, če gremo po vseh trezorjih iz D_{t-1}.

Namesto da v vsakem časovnem intervalu znova računamo vse delne vsote, lahko vzdržujemo Fenwickovo drevo. Pri njem porabimo $O(\log d)$ časa za vsak spremenjeni detektor, medtem ko smo za izračun vseh delnih vsot prej porabili $O(d)$ časa; tu je torej drevo v prednosti, če je število spremenjenih detektorjev majhno. Slabost pa je, da pri drevesu porabimo tudi $O(\log d)$ časa za vsak trezor, pri katerem moramo preveriti, ali v njem dovolj detektorjev zaznava prisotnost lopova, medtem ko smo s tabelo delnih vsot porabili le $O(1)$ časa za vsak tak trezor. Drevo je torej v prednosti, če je teh trezorjev (to so vsi trezorji, ki so sosedje kakšnega trezorja iz D_{t-1}) malo.

Obe možnosti, tabelo delnih vsot in Fenwickovo drevo, lahko tudi skombiniramo: načeloma vzdržujemo Fenwickovo drevo, če pa pri kakšnem časovnem intervalu opazimo, da je spremenjenih detektorjev veliko ali pa da imajo trezorji iz D_{t-1} veliko sosedov, lahko takrat vendarle zgradimo tabelo delnih vsot in jo uporabljamo pri tem časovnem intervalu, na koncu pa jo predelamo v Fenwickovo drevo, kar vzame še $O(d)$ časa. Tako nam obdelava posameznega časovnega intervala vzame $O(\min\{d + n', (d' + n') \log d\})$ časa, če imamo d' spremenjenih detektorjev in n' trezorjev, ki so sosedje tistih iz D_{t-1}.

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. e-knjiga

Vhodno besedilo lahko beremo znak po znak; pri vsakem branju moramo še preveriti, če smo že prišli do konca (EOF). Med branjem vzdržujemo dve celoštevilski spremenljivki, ki štejeta doslej prebrane črke in samoglasnike. Po branju novega znaka preverimo, če je ta znak črka oz. samoglasnik, in po potrebi povečamo enega ali oba števca. Ko pridemo do konca vhodnih podatkov, moramo le še izpisati vrednosti obeh števcev. Oglejmo si primer takšne rešitve v C/C++:

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

int main()
{
  int c, stCrk = 0, stSamoglasnikov = 0;
  while ((c = fgetc(stdin)) != EOF)
  {
    c = toupper(c);
    if (c < 'A' || c > 'Z')
      continue;
    stCrk++;
    if (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U')
      stSamoglasnikov++;
  }
  printf("%d črk, %d samoglasnikov.\n", stCrk, stSamoglasnikov);
  return 0;
}
```

Funkcijo toupper iz C-jeve standardne knjižnice smo uporabili, da nam pretvori male črke v velike, tako da nam kasneje pri preverjanju, ali je znak črka in ali je samoglasnik, ni treba posebej preverjati še malih črk. Šlo pa bi seveda tudi brez tega, le pogoji v stavkih **if** bi bili malo daljši.

Oglejmo si še primer rešitve v pythonu. Ta bo za spremembo brala vhodne podatke po vrsticah:

```
import sys
stCrk = 0; stSamoglasnikov = 0
for s in sys.stdin:
  for c in s:
    if 'A' <= c <= 'Z' or 'a' <= c <= 'z':
      stCrk += 1
      if c in "AEIOUaeiou":
        stSamoglasnikov += 1
print("%d črk, %d samoglasnikov." % (stCrk, stSamoglasnikov))
```

2. Večerja Franca Jožefa

Dogajanje na eni mizi nič ne vpliva na dogajanje na drugih mizah (ker sosedje posameznega človeka sedijo za isto mizo kot on), zato lahko naš postopek obravnava vsako mizo posebej. Recimo torej, da gledamo neko mizo dolžine d; vprašanje je

zdaj, koliko različnim ljudem za to mizo lahko prinesemo večerjo, ne da bi kdo od njih začel jesti.

Mizo si predstavljajmo kot razdeljeno na d stolpcev, oštevilčenih od 1 do d, pri čemer vsak stolpec pokriva dva gosta (enega na vsaki strani mize). Če sta v nekem stolpcu oba gosta že dobila večerjo, bomo rekli, da je *poln*, sicer pa, da je *prazen*. Vidimo lahko, da ne smemo imeti treh polnih stolpcev skupaj, saj bi tedaj gosta v srednjem od njih že lahko začela jesti. Med dvema zaporednima praznima stolpcema smeta biti torej največ dva polna, ne pa trije ali več.

Levo od prvega (najbolj levega) praznega stolpca pa sme biti največ en poln; če bi bila dva, bi gosta v levem od teh dveh že lahko začela jesti; več kot dva pa tudi ne smeta biti, saj smo že v prejšnjem odstavku videli, da ne smemo imeti treh polnih stolpcev skupaj. Podoben razmislek nam tudi pove, da sme biti desno od zadnjega (najbolj desnega) praznega stolpca največ en poln.

Če imamo torej k praznih stolpcev, je lahko celotna miza dolga največ $3k$ (en poln stolpec levo od prvega praznega; eden desno od zadnjega praznega; po dva med vsakima dvema zaporednima praznima; to je skupaj $1 + 1 + 2 \cdot (k - 1)$ polnih stolpcev; ko prištejemo še k praznih, imamo skupaj $3k$ stolpcev). Pri mizi dolžine d mora torej veljati $d \leq 3k$, kar lahko predelamo v $k \geq d/3$. Ker mora biti k celo število, je najmanjši primerni k enak $\lceil d/3 \rceil$ (to je vrednost, ki jo dobimo, če $d/3$ zaokrožimo navzgor na najbližje celo število).

Imeti moramo torej vsaj $\lceil d/3 \rceil$ praznih stolpcev in v vsakem praznem stolpcu mora biti vsaj en gost brez večerje; večerjo lahko torej prinesemo kvečjemu $2d - \lceil d/3 \rceil$ gostom, to pa je isto kot $\lfloor 5d/3 \rfloor$ (torej vrednost $5d/3$, zaokrožena navzdol na najbližje celo število). To moramo le še sešteti po vseh mizah. Zapišimo dobljeno rešitev v C/C++:

```
int Vecerja(int n, int dolzine[])
{
  int r = 0;
  for (int i = 0; i < n; i++)
    r += (5 * dolzine[i]) / 3;
  return r;
}
```

Ali v pythonu:

```
def Vecerja(dolzine):
  return sum((5 * d) // 3 for d in dolzine)
```

3. Robot

Niz, ki opisuje gibanje robota, bomo v zanki obdelovali znak po znak in po vsakem koraku ustrezno popravili podatke o položaju robota. Poleg njegove x- in y-koordinate moramo hraniti še smer, v katero je obrnjen, saj je od smeri odvisno, kako se njegove koordinate spremenijo pri premiku naprej.

Smer bi sicer lahko predstavili s kotom v stopinjah, ker pa se robot vedno obrača za 90 stopinj, so za orientacijo robota le štiri možnosti in jih bomo predstavili kar s števili od 0 do 3 (0 = desno, 1 = gor, 2 = levo, 3 = dol). Pri zasuku v levo se torej smer poveča za 1, pri zasuku v desno pa zmanjša za 1; paziti moramo le na to, da

če bi pri tem smer padla pod 0 ali narasla nad 3, jo moramo povečati oz. zmanjšati za 4.

Sprememba koordinat pri koraku naprej je odvisna od orientacije robota. Če je obrnjen desno, se x poveča za 1, y pa ostane nespremenjen; če je obrnjen gor, je ravno obratno; itd. Te stvari je najlažje predstaviti kar s parom tabel DX in DY, ki povesta spremembo koordinate x oz. y v odvisnosti od smeri.

Oglejmo si implementacijo takšne rešitve v C++:

```
#include <iostream>
using namespace std;

int main()
{
  const int DX[] = { 1, 0, −1, 0 }, DY[] = { 0, 1, 0, −1 };
  int n, x, y, smer = 0;
  // Preberimo začetni položaj in število korakov.
  cin >> n >> x >> y;
  // Preberimo opis gibanja robota.
  while (n−− > 0)
  {
    // Preberimo naslednji korak.
    char c; cin >> c;
    // Ustrezno popravimo položaj ali orientacijo robota.
    if (c == 'L') smer = (smer + 1) % 4;
    else if (c == 'D') smer = (smer + 3) % 4;
    else x += DX[smer], y += DY[smer];
  }
  // Izpišimo končni položaj robota.
  cout << x << " " << y << endl;
  return 0;
}
```

In v pythonu:

```
import sys

DX = [1, 0, −1, 0]
DY = [0, 1, 0, −1]
# Preberimo začetni položaj in število korakov.
n, x, y = (int(s) for s in sys.stdin.readline().split())
# Preberimo opis gibanja robota.
s = sys.stdin.readline(); smer = 0
for i in range(n):
  # Obdelajmo naslednji korak robota.
  if s[i] == 'L': smer = (smer + 1) % 4
  elif s[i] == 'D': smer = (smer + 3) % 4
  else: x += DX[smer]; y += DY[smer]
# Izpišimo končni položaj robota.
print("%d %d" % (x, y))
```

4. Čokolada

Označimo dobroto i-te vrstice z d_i; torej $d_i = \ell_i - r_i$. Naloga zdaj pravzaprav sprašuje, kakšna je največja vsota oblike $s_{ij} = d_i + d_{i+1} + \ldots + d_j$ (to je dobrota

takega kosa čokolade, ki obsega vrstice od vključno i-te do vključno j-te). Pri tem morata seveda i in j ostati v okvirih $1 \le i \le j \le n$, ker ima naša čokolada n vrstic.

Preprosta rešitev je, da gremo z zanko po vseh i (od 1 do n) in nato pri vsakem i s še eno vgnezdeno zanko po vseh j (od i do n), da izračunamo vse vsote s_{ij} in si zapomnimo največjo. Vsot ni treba računati vsake posebej od začetka; ko pri fiksnem i povečamo j za 1, pridobi vsota s_{ij} na koncu še en seštevanec, vse pred njim pa ostane nespremenjeno. Tako moramo le prišteti novi seštevanec k dosedanji vsoti, pa dobimo novo vsoto za malo večji odsek čokolade. Zapišimo ta postopek s psevdokodo:

```
s* := −∞; (* Najboljši doslej najdeni rezultat. *)
for i := 1 to n:
  s := 0;
  for j := i to n:
    (* Na tem mestu je s = d_i + ... + d_{j−1}. *)
    s := s + d_j;
    (* Zdaj je s = d_i + ... + d_j. Poglejmo, če je to najboljša rešitev doslej. *)
    if s > s* then s* := s;
return s*; (* Vrnimo najboljšo rešitev. *)
```

Pregledati moramo $O(n^2)$ parov (i, j), pri vsakem pa imamo $O(1)$ dela, da popravimo vsoto in preverimo, če je najboljša doslej (in si jo zapomnimo, če je res najboljša doslej). Časovna zahtevnost te rešitve je torej $O(n^2)$.

Do boljše rešitve pridemo z naslednjim opažanjem: vsoto od i-tega do j-tega člena lahko dobimo tako, da vzamemo vsoto prvih j členov in od nje odštejemo vsoto prvih $i-1$ členov. Definirajmo torej kumulativne vsote: $c_k = d_1 + d_2 + \ldots + d_k$; potem za dobroto tistega kosa čokolade, ki obsega vrstice od i-te do j-te, velja $s_{ij} = c_j - c_{i-1}$.

Če primerjamo zdaj vrednosti s_{ij} za različne i pri fiksnem j, vidimo, da imajo vse isti c_j, vendar različne c_{i-1}. Da bo s_{ij} čim večja, mora biti c_{i-1} čim manjša. Ko smo pri nekem j, torej pravzaprav ni treba pregledati vseh možnih i, ampak le tistega, ki da najmanjšo c_{i-1} (izmed vseh doslej pregledanih). Zdaj ne potrebujemo več dveh gnezdenih zank, ampak le eno:

```
s* := −∞; (* Najboljši doslej najdeni rezultat. *)
c := 0; m := 0; (* c je trenutna kumulativna vsota, m je najmanjša doslej. *)
for j := 1 to n:
  (* Na tem mestu je c = c_{j−1} in m = min{c_0, ..., c_{j−1}}. *)
  c := c + d_j;
  (* Zdaj je c = c_j. *)
  s := c − m;
  (* Zdaj je s = max_{1≤i≤j} s_ij. Poglejmo, če je to najboljša rešitev doslej. *)
  if s > s* then s* := s;
  (* Poglejmo, če je c najmanjša kumulativna vsota doslej. *)
  if c < m then m := c;
return s*; (* Vrnimo najboljšo rešitev. *)
```

Pri vsakem j imamo zdaj le konstantno mnogo dela, tako da je časovna zahtevnost naše nove rešitve le $O(n)$.

5. Golombovo ravnilo

Označimo z n število oznak na ravnilu, njihove lege pa (od leve proti desni) z $x_1, \ldots, x_n$. Dolžino ravnila označimo z d; naloga pravi, da je $x_1 = 0$ in $x_n = d$.

Na koliko načinov si lahko izberemo dve oznaki, med katerima bomo merili razdaljo? Če si za levo oznako izberemo x_1, si lahko za desno izberemo katerokoli od ostalih $n-1$ oznak; če si za levo oznako izberemo x_2, si lahko za desno izberemo katerokoli od $n-2$ oznak, ki ležijo desno od nje; in tako naprej. Skupaj imamo torej $(n-1)+(n-2)+\ldots+2+1 = n(n-1)/2$ različnih parov oznak. Vse možne pare lahko pregledamo z dvema gnezdenima zankama, eno za levo in eno za desno oznako, in pri vsakem paru (i, j) izračunamo razdaljo med njima, torej razliko $x_j - x_i$.

Ravnilo je Golombovo, če so vse te razdalje med seboj različne, in popolno, če med njimi nastopi vsaj enkrat vsako število od 1 do d. Koristno je torej, če si nekam zapisujemo, katere razdalje smo že videli; tako lahko sproti preverjamo, če smo trenutno razdaljo videli že kdaj prej (če smo jo, lahko zaključimo, da ravnilo *ni* Golombovo), na koncu pa lahko za vse razdalje od 1 do d preverimo, če smo jih sploh kdaj videli (če kakšne nismo, lahko zaključimo, da ravnilo *ni* popolno).

Vprašanje je še, v kakšni podatkovni strukturi hraniti podatke o tem, katere razdalje smo že videli. Za našo nalogo, kjer so ravnila razmeroma kratka, je primerna kar navadna tabela ali vektor logičnih vrednosti; takšna rešitev porabi $O(d)$ pomnilnika, preverjanje tega, ali smo nek element že videli, pa je preprosto in hitro. Še ena možnost bi bila razpršena tabela oz. množica (npr. unordered_set v C++, set v pythonu ipd.), kjer bi bila poraba pomnilnika sorazmerna s tem, koliko je *različnih* razdalj, to pa je $O(\min\{n^2, d\})$. To bi bilo lahko koristno, če je število oznak n majhno v primerjavi z dolžino ravnila d. Časovna zahtevnost naše rešitve je v obeh primerih $O(n^2 + d)$.

Oglejmo si primer rešitve v C++:

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

int main()
{
  // Preberimo vhodne podatke.
  int n; cin >> n;
  vector<int> x(n);
  for (int i = 0; i < n; i++) cin >> x[i];

  // Preglejmo vse razdalje med dvema oznakama.
  int d = x[n - 1];
  vector<bool> zeVideno(d + 1, false);
  bool jeGolombovo = true;
  for (int j = 1; j < n; j++) for (int i = 0; i < j; i++)
  {
    int r = x[j] - x[i];

    // Če razdalje r ne vidimo prvič, ravnilo ni Golombovo.
    if (zeVideno[r]) jeGolombovo = false;
    else zeVideno[r] = true;
  }

  // Preverimo, če je ravnilo popolno.
  bool jePopolno = true;
```

```
  for (int r = 1; r <= d; r++)
    if (! zeVideno[r]) {
      jePopolno = false; break; }
  // Izpišimo rezultate.
  cout << "Ravnilo " << (jeGolombovo ? "je" : "ni") << " Golombovo in ";
  cout << (jePopolno ? "je" : "ni") << " popolno.";
  cout << endl;
  return 0;
}
```

Še podobna rešitev v pythonu:

```
import sys
# Preberimo vhodne podatke.
n = int(sys.stdin.readline())
x = [int(sys.stdin.readline()) for i in range(n)]
# Preglejmo vse razdalje med dvema oznakama.
d = x[-1]
zeVideno = [False] * (d + 1)
jeGolombovo = True
for j in range(1, n):
  for i in range(j):
    r = x[j] - x[i]
    if zeVideno[r]: jeGolombovo = False
    else: zeVideno[r] = True
# Preverimo, ce je ravnilo popolno.
jePopolno = all(zeVideno[1:])
# Izpišimo rezultate.
print("Ravnilo %s Golombovo in %s popolno." % (
  "je" if jeGolombovo else "ni",
  "je" if jePopolno else "ni"))
```

Mimogrede, če se ravnilo izkaže za Golombovo, bi lahko to, ali je tudi popolno, preverili še enostavneje: takrat vemo, da so vse razdalje različne, da jih je $n(n-1)/2$ in da posamezna razdalja ne more biti manjša od 1 ali večja od d; torej, če je $d > n(n-1)/2$, potem gotovo niso prisotne vse možne razdalje od 1 do d (in ravnilo ni popolno), če pa je $d = n(n-1)/2$, potem gotovo so prisotne vse (in ravnilo je popolno).

Razmislimo še o različici naloge, pri kateri ravnilo šteje za Golombovo takrat, ko se vsak interval *ponovi* kvečjemu enkrat (namesto *pojavi* kvečjemu enkrat) — torej se sme pojaviti kvečjemu dvakrat. Vse, kar moramo spremeniti v naši rešitvi, je, da namesto tabele logičnih vrednosti uporabimo tabelo celih števil, ki štejejo, kolikokrat smo posamezni interval doslej že videli. Ko opazimo, da bi se pri nekem intervalu ta števec povečal z 2 na 3, lahko takoj zaključimo, da ravnilo ni Golombovo po tej novi definiciji:

```
  vector<int> zeVideno(d + 1, 0);
  bool jeGolombovo = true;
  for (int j = 1; j < n; j++) for (int i = 0; i < j; i++)
  {
    int r = x[j] - x[i];
```

```
    // Če smo razdaljo r prej videli že dvakrat, ravnilo ni Golombovo.
    if (zeVideno[r] >= 2) jeGolombovo = false;
    else zeVideno[r]++;
  }
```

Ali v pythonu:

```
zeVideno = [False] * (d + 1)
jeGolombovo = True
for j in range(1, n):
  for i in range(j):
    r = x[j] - x[i]
    # Če smo razdaljo r prej videli že dvakrat, ravnilo ni Golombovo
    if zeVideno[r] >= 2: jeGolombovo = False
    else: zeVideno[r] = True
```

Naloge so sestavili: detektorji — Urban Duh; jezero — Primož Gabrijelčič; pekarna — Matija Grabnar in Primož Gabrijelčič; nadzor — Tomaž Hočevar; robot — Branko Kavšek; večerja Franca Jožefa — Vid Kocijan; fitnes, telefonsko omrežje — Samo Kralj; ograje — Mitja Lasič; zaboji — Mitja Lasič in Janez Brank; veriga — Mitja Lasič, Polona Novak in Primož Gabrijelčič; past za žvižgače, Golombovo ravnilo — Mark Martinec; stolpci in vrstice, e-knjiga — Polona Novak; transakcijski računi — Jure Slak; smučarski užitki, čokolada — Jasna Urbančič; razmazani seznam, anagramska razdalja — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2017

1. Dvojno zaokrožanje

Celi del števila za potrebe naše naloge ni zanimiv; važno je le tisto, kar pride za decimalno vejico. Naloga pravi, da so števila podana na dve decimalki; recimo, da sta to števki a (desetinke) in b (stotinke). Če je $a \geq 5$, bi se zaokrožalo v vsakem primeru navzgor, torej z dvojnim zaokrožanjem ne moremo prigoljufati ničesar. Če je $a \leq 3$, po prvem zaokrožanju te števke ne moremo spraviti na več kot 4, zato se v drugem zaokrožanju zagotovo zaokroži navzdol in si tudi ne moremo prigoljufati ničesar.

Dvojno zaokrožanje lahko torej pride prav le, če je $a = 4$. Če hočemo v drugem zaokrožanju zaokrožiti navzgor, moramo v prvem zaokrožanju povečati desetinke s 4 na 5, to pa je možno le, če je $b \geq 5$. Tako torej vidimo, da korist od dvojnega zaokrožanja nastopi le v primerih, ko je ne-celi del števila med 0,45 in 0,49. To je tisto, kar bo moral naš podprogram preveriti.

```
#include <cmath>
using namespace std;

bool DvojnoZaokrozanje(double x)
{
  double y = x - floor(x);
  return 0.445 <= y && y <= 0.495;
}
```

Pomagali smo si s standardno funkcijo floor, ki število zaokroži navzdol; razlika x − floor(x) je torej neceli del števila x, to pa je tisto, za kar moramo preveriti, če leži med 0,45 in 0,49. V pogoju smo meji v resnici malo razširili, da se izognemo težavam zaradi nenatančnosti pri predstavitvi necelih števil s plavajočo vejico.[6]

Če imamo število podano kot niz, nam za težave s predstavitvijo necelih števil ni treba skrbeti; poiskati moramo le decimalno piko v nizu in preveriti, če je naslednji znak `'4'`, še naslednji pa ena od števk `'5'` do `'9'`.

```
bool DvojnoZaokrozanje(const char *x)
{
  while (*x && *x != '.') x++;
  return x[0] == '.' && x[1] == '4' && ('5' <= x[2] && x[2] <= '9');
}
```

Kakorkoli že, naloga pravi, da dobimo seznam števil, tako da potrebujemo še zanko, ki pregleda vse elemente seznama in prešteje, pri koliko izmed njih z dvojnim zaokrožanjem kaj pridobimo:

[6] Na primer: zneska $x = 1{,}45$ se v dvojiškem zapisu ne dá predstaviti s končno mnogo števkami (kajti $1{,}45_{10} = 1{,}01(1100)_2$, pri čemer oklepaji povedo, da se števke 1100 ponavljajo v neskončnost), zato v spremenljivki tipa **double** namesto 1,45 dobimo najbližje število, ki ga je tam še mogoče predstaviti, to pa je $1{,}45 - 1/(5 \cdot 2^{52})$, torej je njegov neceli del (v spremenljivki y) malo manj kot 0,45 in če bi v pogoju uporabili `0.45 <= y`, ta ne bi bil izpolnjen.

```
#include <vector>

int KolikoPridobimo(const vector<double> &seznam)
{
  int vsota = 0;
  for (double x : seznam) if (DvojnoZaokrozanje(x)) vsota += 1;
  return vsota;
}
```

Razmislimo še o težji različici naloge, pri kateri vhodna števila niso nujno podana na dve decimalki. Naj bo spet a prva števka za decimalno vejico. Podobno kot pri prvotni različici naloge vidimo, da če je $a \geq 5$, bi že z enkratnim zaokrožanjem zaokrožili navzgor, zato od večkratnega zaokrožanja ni nobene koristi; in če je $a \leq 3$, bomo na koncu v vsakem primeru zaokrožili navzdol, tudi če bomo prej z enim ali več zaokrožanji na nižjih mestih spremenili a v 4.

Večkratno zaokrožanje lahko torej pride prav le pri $a = 4$. Recimo, da za to štirico pride še nekaj štiric (lahko tudi nobena), prej ali slej pa mora nastopiti neka števka, ki ni 4; recimo ji b. (Če je treba, dodajmo našemu številu na koncu še eno ničlo, tako da bo tak b zagotovo obstajal.) Podoben razmislek kot prej nam zdaj pove, da če je $b \leq 3$, ne bo od večkratnega zaokrožanja nobene koristi, kajti kjerkoli zaokrožimo, se bo zaokrožilo navzdol; če pa je $b \geq 5$, lahko z večkratnim zaokrožanjem počasi spreminjamo štirice v petice in tako dosežemo, da se število na koncu zaokroži navzgor. Primer: $7{,}4446 \to 7{,}445 \to 7{,}45 \to 7{,}5 \to 8$.

Da ne bo težav z zaokrožitvenimi napakami, predpostavimo, da imamo število podano kot niz:

```
bool VeckratnoZaokrozanje(const char *x)
{
  // Poiščimo decimalno piko.
  while (*x && *x != '.') x++;

  // Če je bilo število celo ali pa prva števka za decimalno piko
  // ni 4, potem od večkratnega zaokrožanja ni koristi.
  if (x[0] != '.' || x[1] != '4') return false;

  // Poiščimo za decimalno piko prvo števko, ki ni 4.
  x += 2; while (*x == '4') x++;

  // Večkratno zaokrožanje se splača, če je ta števka večja od 4.
  return *x > '4';
}
```

Pogoj `*x > '4'` na koncu deluje tudi v primeru, ko so za decimalno vejico same štirice; takrat se namreč zanka v predhodni vrstici ustavi pri znaku `'\0'` na koncu niza, ta pa je manjši od `'4'`, tako da funkcija takrat pravilno vrne **false**.

2. Volilni kolegij

Recimo brez izgube za splošnost, da je zmagal A. V regijah, kjer je imel večino B, nima smisla spreminjati nobenih glasov, saj je B tam že zdaj dobil vse elektorske glasove. Oglejmo si zdaj regije, v katerih je imel večino A. Za vsako od teh regij lahko izračunamo, koliko volilcev bi moralo v njej spremeniti svoj glas, da bi imel potem v regiji večino B. V regiji i s p_i prebivalci potrebuje kandidat vsaj $(p_i + 1)/2$

glasov, da ima večino (spomnimo se, da je p_i lih). Ker jih je imel B le $p_i - a_i$, je potrebna sprememba torej $d_i := (p_i + 1)/2 - (p_i - a_i) = a_i - (p_i - 1)/2$. S to spremembo pridobi B vseh r_i elektorjev te regije. Sprememba, večja od d_i, v tej regiji ni smiselna, saj enak učinek dosežemo že s spremembo zgolj d_i glasov; sprememba, manjša od d_i, pa tudi ni smiselna, saj bo B tedaj ostal poraženec v tej regiji in ne bo pridobil nobenega elektorja.

Edini dve smiselni možnosti glede regije i sta torej ta, da svoj glas spremeni d_i volilcev (in B v tej regiji zmaga in pridobi r_i elektorjev), in ta, da ga ne spremeni nihče (in B v tej regiji ostane poraženec in ne pridobi nobenega elektorja). Če na začetku izračunamo, koliko najmanj elektorjev mora poraženec B pridobiti, da postane zmagovalec, imamo zdaj pred seboj problem nahrbtnika: izbrati moramo nekaj regij i tako, da bo vsota njihovih r_i dosegla (ali presegla) nek prag, pri tem pa naj bo vsota njhovih d_i najmanjša možna.

Kot je pri problemu nahrbtnika običajno, ga lahko rešimo z dinamičnim programiranjem. Naj bo $f(i, g)$ najmanjše skupno število volilcev, ki bi morali v prvih i regijah spremeniti svoj glas, da bi poraženec volitev pridobil iz teh regij vsaj g elektorskih glasov. Če v regiji i nihče ne spremeni svojega glasu, moramo g elektorjev pridobiti v prejšnjih $i - 1$ regijah, za to pa mora spremeniti svoj glas $f(i - 1, g)$ volilcev; če pa v regiji i spremeni svoj glas d_i volilcev, je potem v prejšnjih $i - 1$ regijah dovolj že, če pridobimo le $g - r_i$ elektorjev. Tako smo dobili rekurzivno zvezo

$$f(i, g) = \min\{f(i - 1, g), f(i - 1, g - r_i) + d_i\}.$$

Robni primeri so $f(i, g) = 0$ za $g \leq 0$ in $f(0, g) = \infty$ za $g > 0$. Funkcijo f je koristno računati po naraščajočih i in si rezultate shranjevati v tabelo. Ko računamo vrednosti $f(i, \cdot)$, moramo imeti pri roki vrednosti $f(i-1, \cdot)$, starejše (torej $f(i-2, \cdot)$, $f(i-3, \cdot)$ itd.) pa lahko sproti pozabljamo, ker jih ne bomo več potrebovali. Spodnja rešitev ima v ta namen dva vektorja, f[0] in f[1], pri čemer spremenljivka c pove, kateri od njiju hrani $f(i, \cdot)$, kateri pa $f(i - 1, \cdot)$.

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

int KolikoSprememb(int n, const vector<int> &pi, const vector<int> &ri,
                   const vector<int>& ai)
{
  // Izračunajmo število elektorjev za oba kandidata.
  int R = 0, Ra = 0, P = 0;
  for (int i = 0; i < n; i++) {
    R += ri[i]; P += pi[i];
    if (ai[i] > pi[i] - ai[i]) Ra += ri[i]; }
  int Rb = R - Ra; bool zmagaB = (Rb > Ra);

  // Koliko elektorskih glasov manjka poraženci?
  int Rm = (R + 1) / 2 - (zmagaB ? Ra : Rb);

  // Rešimo problem nahrbtnika. Najprej izračunajmo f(0, g) za vse g.
  vector<int> f[2]; f[0].resize(Rm + 1); f[1].resize(Rm + 1);
  f[0][0] = 0; for (int g = 1; g <= Rm; g++) f[0][g] = P + 1;
  int c = 0; // Pove, da je f[c] trenutno aktualni vektor.

  // Izračunajmo f(i, g) za večje i.
```

```
  for (int i = 0; i < n; i++) {
    // Koliko glasov je poraženec celotnih volitev dobil v tej regiji?
    int gp = (zmagaB) ? ai[i] : pi[i] − ai[i];
    // Koliko glasov mu manjka do zmage v tej regiji?
    int di = (pi[i] + 1) / 2 − gp;
    // Če je tu že zmagal, ne more pridobiti dodatnih elektorjev.
    if (di <= 0) continue;
    // V f[c] so vrednosti f(i, .), v f[1 − c] pa bomo
    // izračunali vrednosti f(i + 1, .).
    auto &fs = f[c], &fn = f[1 − c]; c = 1 − c;
    for (int g = 0; g <= Rm; g++)
      fn[g] = min(fs[g], (g >= ri[i] ? fs[g − ri[i]] : 0) + di); }
  // Vrnimo rezultat, to je f(n, Rm).
  return f[c][Rm];
}
```

3. Zid

Če nek pravokotnik obdržimo, to pomeni, da moramo obdržati tudi tista dva pravokotnika v naslednji vrstici, na katerih ležita njegovi spodnji dve oglišči. Ker za začetek vemo, da moramo obdržati vse pravokotnike v najvišji vrstici, lahko od tam pregledujemo zid po vrsticah od zgoraj navzdol in si sproti označujemo, katere pravokotnike moramo obdržati v nižjih vrsticah.

Recimo, da ima zid h vrstic, ki jih oštevilčimo od 1 (najvišja) do h (najnižja); število pravokotnikov v vrstici y naj bo n_y, pri čemer naj se i-ti pravokotnik v tej vrstici začne pri x-koordinati z_{yi}, konča pa pri k_{yi}. Predpostavimo, da dobimo pravokotnike v vsaki vrstici že urejene po x-koordinati in da se seveda ne prekrivajo; torej je $z_{yi} < k_{yi} \leq z_{y,i+1}$ za vse y in i. (Če med pravokotniki v začetnem stanju zidu ni vrzeli, velja celo $k_{yi} = z_{y,i+1}$, vendar bo naš postopek, kot bomo videli, deloval tudi za primere, ko smejo biti med pravokotniki vrzeli.)

Ko torej za neko vrstico, recimo $y-1$, vemo, katere pravokotnike v njej hočemo obdržati, se lahko v zanki sprehajamo po tej vrstici in pri vsakem obdržanem pravokotniku pogledamo, na katerih dveh pravokotnikih vrstice y ležita njegovi spodnji dve oglišči; za tista dva pravokotnika potem označimo, da ju moramo obdržati. Pravokotnike, ki jih na ta način ne označimo, pa bomo lahko na koncu zavrgli.

for $i := 1$ **to** n_1 **do** *obdrži*$[1, i]$:= TRUE;
for $y := 2$ **to** h:
  for $j := 1$ **to** n_y **do** *obdrži*$[y, j]$:= FALSE;
  $j := 1$;
  for $i := 1$ **to** n_{y-1} **do if** *obdrži*$[y-1, i]$:
    (* *Pravokotnik i v vrstici $y-1$ bomo obdržali. Preskočimo v vrstici y tiste pravokotnike, ki se končajo levo od njegovega levega roba.* *)
    while $j \leq n_y$ **and** $k_{yj} < z_{y-1,i}$ **do** $j := j + 1$;
    (* *Pravokotnik j je zdaj tisti, ki mora podpirati levo spodnje oglišče i-ja.* *)
    if not $(j \leq n_y$ **and** $z_{yj} < z_{y-1,i} < k_{yj})$ **then**
      Napaka: problem je nerešljiv, ker spodnje levo oglišče i-tega pravokotnika v vrstici $y-1$ ni primerno podprto, odstraniti pa ga ne smemo.

else *obdrži*$[y, j]$:= TRUE;
(* *Preskočimo v vrstici y tiste pravokotnike, ki se končajo levo od desnega roba i-tega pravokotnika v vrstici y.* *)
while $j \le n_y$ **and** $k_{yj} < k_{y-1,i}$ **do** $j := j + 1$;
(* *Pravokotnik j je zdaj tisti, ki mora podpirati desno spodnje oglišče i-ja.* *)
if not $(j \le n_y$ **and** $z_{yj} < k_{y-1,i} < k_{yj})$ **then**
Napaka: problem je nerešljiv (podobno kot zgoraj).
else *obdrži*$[y, j]$:= TRUE;

Ob koncu tega postopka imamo v tabeli *obdrži* podatke o tem, katere pravokotnike moramo obdržati, vse ostale pa lahko odstranimo. Če nas zanima le število odstranjenih pravokotnikov, pa lahko prihranimo nekaj pomnilnika tako, da v vsakem trenutku vzdržujemo le dve vrstici tabele *obdrži* (za y in $y - 1$), število odstranjenih pravokotnikov pa računamo sproti — na začetku vrstice ga povečamo za n_y, nato pa ga za 1 zmanjšamo vsakič, ko se vrednost kakšnega elementa tabele *obdrži* spremeni s FALSE na TRUE.

4. Kolovodja

Razmislimo najprej o lažji različici naloge, pri kateri v grafu ni ciklov. Potem je za kolovodjo potreben pogoj ta, da ima vhodno stopnjo 0 (kajti če obstaja povezava $u \to v$, potem v ne more biti hkrati tudi (posredno ali neposredno) nadrejen u-ju, sicer bi imeli v grafu cikel; to pa pomeni, da tak v gotovo ni kolovodja). Tak kandidat ni podrejen nikomur; torej, če je takih kandidatov več, potem nobeden ni kolovodja, ker ni nadrejen ostalim kandidatom. Če pa je tak kandidat en sam — recimo mu u — je on gotovo kolovodja. Da se prepričamo o tem, moramo pokazati, da je tak u nadrejen vsem ostalim mafijcem, torej da so vse ostale točke grafa dosegljive iz u. Pa recimo, da neka točka v ni dosegljiva iz u; ker nobena točka razen u nima vhodne stopnje 0, kaže v v neka povezava $v_1 \to v$. Točka v_1 je različna od u (sicer bi bila v dosegljiva iz u), torej kaže tudi vanjo neka povezava $v_2 \to v_1$; s tem razmišljanjem lahko nadaljujemo in sestavimo neskončno verigo povezav $v \leftarrow v_1 \leftarrow v_2 \leftarrow v_3 \leftarrow \ldots$. Ker pa nimamo neskončno mnogo točk, se prej ali slej v tej verigi neka točka pojavi dvakrat, to pa pomeni, da vmesni del verige tvori cikel. Tako smo prišli v protislovje, saj je naš graf acikličen. □

S tem razmislekom smo prišli do naslednjega postopka:

(* *Določimo vhodne stopnje vseh točk.* *)
za vsako točko $u \in V$: $s[u] := 0$;
za vsako povezavo $(u \to v) \in E$: $s[v] := s[v] + 1$;
(* *Poiščimo točko z vhodno stopnjo 0 in preverimo, če je edina.* *)
$n_0 := 0$;
za vsako točko $u \in V$:
if $s[u] = 0$:
$n_0 := n_0 + 1$; $u_0 := u$;
if $n_0 = 1$ **then** u_0 je kolovodja **else** kolovodja ne obstaja;

Če smo na ta način uspešno našli kolovodjo, lahko z iskanjem v širino poiščemo najkrajše poti od njega do vseh ostalih točk v grafu in tako rešimo še drugi del naloge:

za vsako točko $u \in V$: $d[u] := -1$;
$d[u_0] := 0$; $k := 0$; $Q :=$ prazna vrsta; dodaj u_0 v Q;
while Q ni prazna:
  vzemi iz vrste Q prvo točko, recimo u; $k := d[u]$;
  za vsako povezavo $(u \to v) \in E$:
    if $d[v] \geq 0$ **then continue**;
    $d[v] := k + 1$; dodaj v na konec vrste Q;

Ta postopek pregleduje točke grafa po naraščajoči oddaljenosti od kolovodje u_0; razdaljo (dolžino najkrajše poti) od u_0 do u shrani v $d[u]$. Največja od teh razdalj se na koncu postopka nahaja v spremenljivki k, torej smo dobili prav tisti k, po katerem sprašuje naloga.

Rešimo zdaj nalogo še v splošnem, torej brez predpostavke, da je graf acikličen. Recimo, da bi v takem grafu poiskali krepko povezane komponente. Za točke iz posamezne take komponente potem velja, da kar je dosegljivo iz ene od njih, je dosegljivo tudi iz vseh ostalih (saj so vse točke v komponenti dosegljive druga iz druge). Zato za potrebe razmišljanja o tem, ali so iz neke točke dosegljive vse druge ali ne, sploh ni treba razlikovati med več točkami iz iste komponente — lahko jih vse združimo v eno samo točko. Tako iz našega prvotnega grafa nastane nov, manjši graf, v katerem vsaka točka predstavlja po eno krepko povezano komponento prvotnega grafa.

Novi graf je acikličen, zato lahko v njem preverimo obstoj kolovodje s prej omenjenim postopkom za aciklične grafe. Če kolovodje ne najdemo, to pomeni, da ga tudi v prvotnem grafu ni bilo; če pa ga najdemo, predstavlja ta kolovodja novega grafa neko krepko povezano komponento prvotnega grafa in kolovodje prvotnega grafa so vse točke tiste komponente (in nihče drug).

Iz vsakega od teh kolovodij lahko potem poženemo iskanje v širino, da določimo njegov k (število korakov, v katerem je mogoče iz njega priti do katerekoli druge točke) — različni kolovodje, čeprav so v isti krepko povezani komponenti, imajo namreč lahko različne k, nas pa najbrž zanima najmanjši med njimi.

5. Podjetniške hobotnice

Recimo, da imamo p podjetij (oštevilčenih s števili od 1 do p) in ℓ posameznikov (oštevilčenih s števili od $p+1$ do $p+\ell$). Tabela deležev (recimo ji T), ki jo dobimo kot vhodni podatek, ima torej $p + \ell$ vrstic in p stolpcev, tako da v njej element $T[u, v]$ pove, ali ima u lastniški delež v podjetju v.

V praksi je podjetij najbrž veliko, posamezni lastnik pa ima deleže le v peščici podjetij. Tabela T bo torej „redka“ — velika večina elementov v njej bo imela vrednost 0 oz. FALSE. Tak način predstavitve lastništva je zato precej potraten s prostorom, pa tudi s časom, kajti ko bomo hoteli pregledati vsa podjetja, v katerih ima nek lastnik deleže, (ali pa vse lastnike nekega podjetja), bomo morali pregledati celo vrstico (ali pa stolpec) tabele, četudi je večina elementov v njej praznih. Zato podatke najprej predelajmo v obliko, ki bo prikladnejša za nadaljnjo obdelavo. Za vsakega lastnika u bomo pripravili seznam $D[u]$ podjetij, v katerih ima deleže, za vsako podjetje v pa seznam njegovih lastnikov $L[v]$:

for $u := 1$ **to** $p + \ell$ **do** $D[u] :=$ prazen seznam;

```
for v := 1 to p do L[v] := prazen seznam;
for u := 1 to p + ℓ:
  for v := 1 to p do if T[u, v]:
    dodaj u v L[v] in dodaj v v D[u];
```

(*a*) Za lastnika u lahko s preiskovanjem v globino, širino ali sploh v poljubnem vrstnem redu pregledamo vse točke v grafu, ki so dosegljive iz u po povezavah v enem ali več korakih; pri tem lahko tudi štejemo, koliko jih je.

```
funkcija KolikoDosegljivih(u):
  for v := 1 to p do dosegljivo[v] := FALSE;
  Q := {u}; n := 0;
  while Q ni prazna:
    v := poljuben element Q-ja; pobriši v iz Q;
    for vsak w ∈ D[v] do if not dosegljivo[w]:
      dodaj w v Q; dosegljivo[w] := TRUE; n := n + 1;
  return n;
```

Če množico Q uporabljamo kot vrsto (vedno pobrišemo iz nje tisti element, ki je že najdlje v vrsti), se ta postopek obnaša kot iskanje v širino; vendar pa bo rezultat enak tudi, če Q uporabljamo kot sklad ali pa če jemljemo podjetja iz nje v poljubnem vrstnem redu.

Ta postopek moramo zdaj pognati po enkrat za vsakega možnega lastnika u, pa bomo lahko ugotovili, kdo vpliva na največ podjetij.

```
funkcija KdoNajveč:
  n* := −1; u* := −1;
  for u := p + 1 to p + ℓ:
      n := KolikoDosegljivih(u);
      if n > n* then n* := n; u* := u;
  return u*;
```

Opisano rešitev bi se dalo na razne načine še izboljšati. Pri iskanju v širino porabimo najprej $O(p)$ časa za inicializacijo tabele *dosegljivo*, kar se utegne izkazati za potratno, ker je v praksi verjetno, da je iz danega u dosegljivo le majhno število podjetij (majhno v primerjavi s p), zato bo preostanek iskanja v širino porabil razmeroma malo časa. Nekaj časa bi lahko torej prihranili, če tabele *dosegljiva* ne bi bilo treba inicializirati znova pri vsakem u. To lahko naredimo tako, da v njej namesto logičnih vrednosti hranimo številke lastnikov; namesto TRUE zdaj uporabljamo številko u, karkšno koli drugo vrednost v tabeli (ki je ostala tam od prejšnjih klicev funkcije KolikoDosegljivih) pa si razlagamo kot FALSE. Tabelo *dosegljivi* inicializiramo le enkrat samkrat, pred prvim klicem KolikoDosegljivih, ko postavimo vse njene elemente na 0.

Še ena koristna izboljšava bi bila, da bi v grafu najprej poiskali krepko povezane komponente. Zanje vemo, da če je iz u dosegljiva ena točke take komponente, so iz u dosegljive tudi vse ostale točke iz tiste komponente. Graf lahko torej predelamo tako, da vse točke posamezne komponente zlijemo skupaj v eno samo, s čimer nastane nov graf, ki ima le toliko točk, kolikor je imel prvotni graf krepko povezanih komponent. Na novem grafu lahko poganjamo enak postopek kot zgoraj, paziti moramo le na to,

da pri iskanju v širino spremenljivke n ne povečujemo vsakič za 1, ampak za število točk tiste komponente originalnega grafa, iz katere je trenutna točka w novega grafa nastala. (Takšna predelava grafa v nov, manjši graf bi prišla prav tudi pri naslednjih podnalogah.)

(b) Tudi tu lahko uporabimo iskanje v širino, le da v nasprotni smeri, od podjetij na njihove lastnike. Začnemo pa pri tistem podjetju, ki nas zanima (torej za katero bi radi našli posameznike, ki vplivajo nanj):

```
funkcija KdoVplivaNa(x):
  for u := 1 to p + ℓ do dosegljivo[u] := false;
  Q := {x}; A := {}; dosegljivo[x] := true;
  while Q ni prazna:
    v := poljuben element Q-ja; pobriši v iz Q;
    for vsak u ∈ L[v] do if not dosegljivo[u]:
      dosegljivo[u] := true;
      if u > p then dodaj u v A else dodaj u v Q;
  return A;
```

Prej smo imeli v tabeli *dosegljivo* le po en element za vsako podjetje, zdaj pa jih potrebujemo tudi za posameznike, saj lahko med pregledovanjem lastnikov naletimo na oboje. Ko naletimo na lastnika, ki je posameznik, ga dodamo v množico A in jo na koncu vrnemo. V praksi lahko to množico predstavimo z navadnim seznamom ali tabelo, saj zagotovo ne bomo poskusili dodati nobenega posameznika v A več kot enkrat.

(c) Spet lahko pregledujemo graf v smeri od lastnikov k podjetjem, podobno kot pri (a); vendar pa zdaj na začetku za dosegljivega ne vzamemo enega samega posameznika, ampak *vse* posameznike. Od tam s pregledovanjem grafa sčasoma dosežemo vsa podjetja, na katera vpliva vsaj en posameznik; nato moramo le še pregledati, ali smo dosegli vsa podjetja ali ne.

```
funkcija AliObstajaNedosegljivo:
  for v := 1 to p do dosegljivo[v] := false;
  Q := {}; for u := p + 1 to p + ℓ do dodaj u v Q;
  while Q ni prazna:
    v := poljuben element Q-ja; pobriši v iz Q;
    for vsak w ∈ D[v] do if not dosegljivo[w]:
      dodaj w v Q; dosegljivo[w] := true; n := n + 1;
  for v := 1 to p do if not dosegljivo[v] then return true;
  return false;
```

(d) Hobotnice iz te podnaloge niso nič drugega kot to, čemur v teoriji grafov rečemo šibko povezane komponente. Tudi te lahko odkrivamo z iskanjem v širino, le da zdaj sledimo povezavam v obe smeri — ko pridemo v neko podjetje, pregledamo tako njegove lastnike kot njegovo lastnino (podjetja, v katerih ima delež). Vse, kar na ta način iz neke začetne točke grafa dosežemo, tvori eno šibko povezano komponento. Potem lahko začnemo v poljubni drugi točki (taki, ki je še nismo dosegli) in z enakim postopkom odkrijemo naslednjo šibko povezano komponento. Tako nadaljujemo, dokler ne pregledamo celotnega grafa.

funkcija HOBOTNICE:
  $\mathcal{H} := \{\}$; **for** $v := 1$ **to** $p + \ell$ **do** *dosegljivo*$[v]$:= FALSE;
  for $x := 1$ **to** $p + \ell$ **do if not** *dosegljivo*$[x]$:
    $H := \{x\}$; $Q := \{x\}$; *dosegljivo*$[x]$:= TRUE;
    while Q ni prazna:
      $v :=$ poljuben element Q-ja; pobriši v iz Q;
      for vsak $w \in D[v]$ **do if not** *dosegljivo*$[w]$:
        dodaj w v Q in v H; *dosegljivo*$[w]$:= TRUE;
      for vsak $w \in L[v]$ **do** (enako kot v prejšnji zanki);
    dodaj H v $\mathcal{H}$;
  return $\mathcal{H}$;

Točke, ki tvorijo trenutno šibko povezano komponento, torej sproti dodajamo v množico H, na koncu pa to množico dodamo v množico $\mathcal{H}$, ki tako sčasoma posane množica vseh šibko povezanih komponent grafa; to pa je tudi tisto, kar moramo na koncu vrniti. V praksi lahko H-je in $\mathcal{H}$ predstavimo s seznami ali čim podobno preprostim.

6. Šikaku

Nalogo lahko rešujemo z rekurzijo, pri kateri postopoma polagamo pravokotnike na mrežo, dokler ni ta v celoti pokrita. Recimo, da smo v nekem trenutku nekaj pravokotnikov že položili na mrežo. Poiščimo najvišje ležeče nepokrito polje; če je takih več, vzemimo med njimi najbolj levo; recimo mu (x, y). Prej ali slej bo treba tudi to polje pokriti z nekim pravokotnikom; toda ta pravokotnik se ne more začeti nad našim poljem (ker je tam že vse pokrito), pa tudi ne v isti vrstici in levo od njega (ker je v tej vrstici levo od njega tudi že vse pokrito). Polje (x, y) lahko torej pokrijemo le tako, da položimo v mrežo nov pravokotnik, ki ima svoj zgornji levi kót ravno v tem polju. Preizkusiti moramo vse možne višine in širine takega pravokotnika, za vsak tako dobljeni pravokotnik pa izvedemo nato rekurzivni klic, s katerim poskušamo nadaljevati pokrivanje mreže.

Razmislimo zdaj o tem, kakšne so možnosti glede višine in širine pravokotnika. Glede širine nas bo prej ali slej ustavil desni rob mreže, mogoče pa bomo že prej naleteli na neko polje (x', y) za $x' > x$, ki ga pokriva že kakšen od prej postavljenih pravokotnikov (ki imajo svoj zgornji levi kot nekje nad vrstico y). Recimo zdaj, da pri neki fiksni širini počasi povečujemo višino; tu se moramo ustaviti pri spodnjem robu mreže ali pa tik preden pokrijemo dve oštevilčeni polji. Ko imamo pokrito natanko eno oštevilčeno polje, lahko iz njegove vrednosti (ki nam predpisuje ploščino pravokotnika) in trenutne širine izračunamo potrebno višino — to je potem edina primerna višina pri tej širini in zgornjem levem kotu. Če je ta višina premajhna (ker z njo ne bi dosegli omenjenega oštevilčenega polja) ali prevelika (ker bi dosegli še kakšno dodatno oštevilčeno polje ali pa padli čez rob mreže), pa tega pravokotnika pač ne moremo uporabiti.

```
// Vhodni podatki o mreži. Neoštevilčena polja naj imajo vrednost 0.
enum { W = ..., H = ... };
int mreza[H][W];

// Globalna spremenljivka, ki pove, kateri pravokotnik pokriva katero polje.
```

```
// Nepokrita polja imajo vrednost 0.
int pokrito[H][W];

void Rekurzija(int x1, int y1, int stPravokotnika)
{
  int s = −1, ys = −1, yMax = H − 1;
  for (int x2 = x1; x2 < W && yMax >= y1; x2++)
  {
    // Na tem mestu velja: če nas zanimajo pravokotniki, ki imajo zgornji levi kot v
    // (x1, y1) in se raztezajo v desno do vključno stolpca x2 − 1, leži prvo število
    // (tisto z najmanjšo y-koordinato), ki ga lahko dosežemo, v vrstici ys,
    // njegova vrednost pa je s; najnižja vrstica, do katere lahko pravokotnik še gre, pa
    // je yMax (pod njo je bodisi še eno število, rob mreže ali pa že pokrito polje). —
    // Nas pa bodo zdaj zanimali pravokotniki, ki se raztezajo tudi v stolpec x2.
    // Kaj se pri zgornjih stvareh spremeni zaradi tega?
    for (int y = y1; y <= yMax; y++)
    {
      // Ustaviti se moramo pred pokritim poljem.
      if (pokrito[y][x2]) { yMax = y − 1; break; }

      // Prazna polja nas nič ne ovirajo.
      if (mreza[y][x2] == 0) continue;

      // Sicer imamo na (x, y) število. Če je to prvo število v našem pravokotniku,
      // si ga le zapomnimo.
      if (s <= 0) { s = mreza[y][x2]; ys = y; }

      // Če poznamo že neko nižje ležeče število, se bomo morali ustaviti, preden
      // bomo poleg našega novega števila pokrili še tisto od prej.
      else if (y < ys) { yMax = ys − 1; ys = y; s = mreza[y][x2]; }

      // Če poznamo že neko število na isti višini, se moramo ustaviti pred njima
      // in bomo imeli pravokotnik brez števila.
      else if (y == ys) { ys = −1; s = −1; yMax = y − 1; break; }

      // Če pa poznamo že neko višje ležeče število, se moramo ustaviti zdaj, preden
      // bomo pokrili še tole novo število.
      else { yMax = y − 1; break; }
    }

    // Če nimamo pokritega nobenega števila, pri tej širini ne moremo sestaviti
    if (s <= 0) continue;                                    // pravokotnika.

    // Sicer nam število določa ploščino, ta skupaj s širino pa višino.
    int w = x2 − x1 + 1;
    int h = s / w; if (w * h != s) continue;
    int y2 = y1 + h − 1; if (y2 < ys || y2 > yMax) continue;

    // Uporabimo ta pravokotnik.
    for (int y = y1; y <= y2; y++) for (int x = x1; x <= x2; x++)
      pokrito[y][x] = stPravokotnika;

    // Nadaljujmo pri naslednjem nepokritem polju.
    int nasl = (x2 + 1) + y1 * W;
    while (nasl < W * H && pokrito[nasl / W][nasl % W]) nasl++;
    if (nasl >= W * H) IzpisiRezultat();
    else rezultat += Rekurzija(nasl % W, nasl / W, stPravokotnika + 1);

    // Pobrišimo ta pravokotnik.
    for (int y = y1; y <= y2; y++) for (int x = x1; x <= x2; x++) pokrito[y][x] = 0;
  }
}
```

Gornja funkcija kliče podprogram IzpisiRezultat, če oz. ko ji uspe pokriti celo mrežo. Takrat je v vsakem elementu tabele pokrito številka tistega pravokotnika (od 1 naprej), ki pokriva to polje. Podrobnosti tega podprograma prepustimo bralcu za vajo, saj naloga nič ne govori o tem, kako naj izpišemo rezultate. Glavni del programa bi moral pripraviti tabelo mreza, postaviti v pokrito vse elemente na 0 in poklicati Rekurzija(0, 0, 1), da začne s pokrivanjem v zgornjem levem kotu mreže in s pravokotnikom številka 1.

Naloge bi se lahko lotili še na drugačne načine. Recimo, da imamo k polj, na katerih so števila; število na i-tem od teh polj označimo z n_i. Potem iz omejitev v besedilu naloge sledi, da bomo imeli natanko k pravokotnikov; za vsako polje s številom bo po en pravokotnik, njegova ploščina pa bo enaka vrednosti tega števila. Med drugim to pomeni, da mora biti vsota $\sum_{i=1}^{k} n_i$ ravno enaka površini cele mreže, sicer je naloga nerešljiva (ker bi se pravokotniki bodisi morali prekrivati bodisi ne bi mogli pokriti cele mreže). Naj bo R_i množica vseh možnih pravokotnikov, ki pokrivajo i-to polje s številom, imajo ploščino n_i, ne pokrivajo nobenega drugega števila in ne štrlijo čez rob mreže. Naloga potem pravzaprav zahteva, da iz vsake R_i izberemo po en pravokotnik in to tako, da se izbrani pravokotniki med seboj ne prekrivajo. To lahko zastavimo kot problem logičnega programiranja z omejitvami (*constraint logic programming*, CLP): imejmo k neznank $x_1, \ldots, x_k$, ki povedo, katere pravokotnike izberemo; neznanka x_i ima torej zalogo vrednosti R_i; za vsak par neznank (i, j) pa imamo po eno omejitev $Disjunktna_{ij}(x_i, x_j)$, ki je izpolnjena natanko tedaj, ko sta pravokotnika $x_i \in R_i$ in $x_j \in R_j$ disjunktna.[7]

Še ena možnost je, da nalogo prevedemo na problem celoštevilskega linearnega programiranja (*integer linear programming*, ILP). Za vsako število i in za vsak pravokotnik $j \in R_i$ imejmo zdaj po eno celoštevilsko neznanko x_{ij}, omejitve pa naj bodo: $0 \le x_{ij} \le 1$ za vse i in j (vrednost x_{ij} pove, ali pravokotnik j uporabimo ali ne); $\sum_j x_{ij} = 1$ za vsak i (i-to število moramo pokriti z natanko enim pravokotnikom); in če se pravokotnika $j \in R_i$ ter $j' \in R_{i'}$ kaj prekrivata, imejmo še omejitev $x_{ij} + x_{i'j'} \le 1$. Najti moramo poljuben nabor vrednosti neznank x_{ij}, ki ustreza vsem tem omejitvam. Celoštevilsko linearno programiranje je v splošnem NP-težak problem, vendar obstajajo zanj algoritmi oz. knjižnice, ki v obvladljivem času rešijo marsikateri praktično zanimiv primer tega problema.

7. Pretakanje tekočine

V obeh različicah naloge se skriva znani problem 0/1-nahrbtnika, ki ga lahko rešujemo z dinamičnim programiranjem.

(a) Recimo, da se omejimo le na prvih k tipov posod; naj bo zdaj $f(v, k)$ najnižja cena za nakup posod s skupno prostornino v. Pri takem nakupu imamo dve možnosti: (1) lahko ne kupimo nobene posode tipa k; potem nam ostane vprašanje, kako prostornino v doseči čim ceneje samo s prvimi $k-1$ tipi posod; najnižja cena tega pa je ravno $f(v, k-1)$. (2) Lahko pa kupimo posodo tipa k; s tem porabimo C_k denarja, ostane pa nam vprašanje, kako preostanek prostornine (to je $v - V_k$) doseči

[7] Z logičnim programiranjem z omejitvami smo se na naših tekmovanjih že srečali; glej npr. nalogo 1995.3.2 (str. 237 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004*). Probleme tega tipa rešujemo z rekurzijo, pri čemer omejitve uporabljamo za to, da sproti čim bolj ožimo nabor možnih vrednosti vsake neznanke.

čim ceneje s prvimi k tipi posod (tako bomo lahko kupili še kakšno posodo tipa k, saj naloga pravi, da je na voljo neomejeno število posod vsakega tipa); najnižja cena je potem skupaj $C_k + f(v - V_k, k)$.

Med omenjenima dvema možnostma vzamemo seveda tisto, ki doseže najnižjo ceno. Tako smo dobili naslednjo rekurzivno formulo za funkcijo f:

$$f(v,k) = \min\{f(v,k-1), C_k + f(v - V_k, k)\}.$$

Robni primeri so $f(v,k) = 0$ za $v \le 0$ (če ni treba pretočiti nič vode, tudi ni treba kupiti nobene posode) in $f(v,0) = \infty$ za $v > 0$ (če je treba pretočiti nekaj vode, pa ne smemo kupiti nobene posode, je problem nerešljiv).

Kot je običajno pri dinamičnem programiranju, si je koristno že izračunane vrednosti funkcije f shranjevati, ker jih bomo kasneje še večkrat potrebovali. Funkcijo bomo računali po naraščajočih v in naraščajočih k; tako bomo pri izračunu $f(v,k)$ zagotovo imeli že pripravljeni vrednosti $f(v,k-1)$ in $f(v-V_k,k)$, ki ju bomo takrat potrebovali.

for $v := 1$ **to** P **do for** $k := 1$ **to** n:
  $r := C_k$; **if** $v > V_k$ **then** $r := r + f[v - V_k, k]$;
  if $k > 1$ **then** $r := \min\{r, f[v, k-1]\}$;
  $f[v,k] := r$;

Najnižja cena, s katero lahko pretočimo celotno vsebino cisterne, je potem $f(P,n)$. Časovna zahtevnost te rešitve je $O(P \cdot n)$, prav tako pa tudi prostorska.

Naloga sprašuje še, koliko posod posameznega tipa potrebujemo; to lahko ugotovimo, če gremo od (P,n) nazaj po tabeli vrednosti funkcije f in na vsakem koraku pogledamo, katera od obeh možnosti v izrazu $f(v,k) = \min\{\ldots\}$ je bila manjša.

for $k := 1$ **to** n **do** *koliko*$[k] := 0$;
$v := P$; $k := n$;
while $v > 0$:
  if $k > 1$ **and** $f[v,k] = f[v,k-1]$ **then** $k := k - 1$;
  else $v := v - V_k$; *koliko*$[k]$:= *koliko*$[k] + 1$;

Če je $f[v,k] = f[v,k-1]$, to pomeni, da je prostornino v s prvimi k tipi posod mogoče najceneje doseči tako, da posode tipa k sploh ne uporabimo, zato takrat zmanjšamo k in ostanemo pri enaki prostornini; sicer pa povečamo števec posod tipa k in zmanjšamo prostornino za V_k. Zanka **while** porabi največ $O(P+n)$ časa, saj se v vsaki iteraciji zmanjša bodisi v bodisi k.

(*b*) Načeloma si lahko pomagamo kar s funkcijo f iz prvega dela naloge. Če gledamo vrednosti $f(v,n)$ za naraščajoče v, vidimo, da tudi te vrednosti počasi naraščajo (če zahtevano prostornino povečamo, se potrebna cena lahko poveča ali pa ostane enaka, ne more pa se zmanjšati). Podnaloga (*b*) torej pravzaprav sprašuje, kdaj ta vrednost preseže D — iščemo tisti v, pri katerem je $f(v,n) \le D < f(v+1,n)$.

Ena razlika v primerjavi z (*a*) je še ta, da smo tam vedeli, da moramo iti do prostornine $v = P$, tukaj pa največje prostornine ne vemo vnaprej, pač pa moramo sproti gledati, kdaj $f(v,n)$ preseže D.

$v := 1$;
while true:

for $k := 1$ **to** n:
  $r := C_k$; **if** $v > V_k$ **then** $r := r + f[v - V_k, k]$;
  if $k > 1$ **then** $r := \min\{r, f[v, k-1]\}$;
  $f[v, k] := r$;
if $f[v, n] > D$ **then return** $v - 1$
else $v := v + 1$;

Ker podnaloga (b) sprašuje le, kakšna je največja možna prostornina, ne pa tudi, s kakšnim naborom posod jo lahko dosežemo, nam tabele f ni treba hraniti v celoti. Opazimo lahko, da pri danem v potrebujemo le vrednosti f-ja za prostornine $v - V_k$ za vse možne k; če označimo z $V := \max_k V_k$ prostornino največjega tipa posode, to pomeni, da moramo pri danem v hraniti le vrednosti funkcije od $v - V$ naprej, tiste za manjše prostornine pa lahko sproti pozabljamo. Tako bo poraba prostora omejena na $O(V \cdot n)$ ne glede na to, kako veliko prostornino bomo na koncu vrnili.

Nalogo lahko elegantno rešimo tudi tako, da kot argument funkcije namesto prostornine vzamemo ceno. Naj bo torej $g(c, k)$ največja prostornina, ki jo lahko dosežemo, če imamo za nakup posode na voljo c evrov in smo omejeni na prvih k tipov posode. Naloga potem sprašuje po vrednosti $g(D, n)$. Pri izračunu $g(c, k)$ lahko razmišljamo podobno kot pri funkciji f: če ne kupimo nobene posode tipa k, lahko z ostalimi tipi dosežemo prostornino $g(c, k-1)$; če pa kupimo eno posodo tipa k, dosežemo prostornino $V_k + g(c - C_k, k)$. Tako dobimo

$$g(c, k) = \min\{g(c, k-1), V_k + g(c - C_k, k)\}$$

z robnimi primeri $g(c, k) = -\infty$ za $c < 0$ (do tega pride, če smo pred tem kupili posodo, za katero v resnici nismo imeli dovolj denarja) in $g(c, 0) = 0$ za $c \geq 0$ (takrat ni na voljo nobene posode, ki bi jo lahko kupili, zato je dosežena prostornina 0). Tudi to funkcijo lahko računamo po naraščajočih c in k ter si rezultate shranjujemo v tabelo:

for $k := 1$ **to** n **do for** $c := 0$ **to** P:
  if $k > 1$ **then** $r := g[c, k-1]$ **else** $r := 0$;
  if $c \geq C_k$ **then** $r := \min\{r, V_k + g[c - C_k, k]\}$;
  $g[c, k] := r$;
return $g[D, n]$;

Lepo pri tej rešitvi je, da ko računamo funkcijo g za nek k, potrebujemo le njene vrednosti za ta k in za $k - 1$, starejših (torej za $k - 2$, $k - 3$ itd.) pa ne in jih lahko sproti pozabljamo. Tako porabimo le $O(D)$ prostora.

8. Barvanje ograje

Ena možnost je, da pobarvamo ograjo s samimi navpičnimi potezami. Druga možnost je, da najprej naredimo nekaj vodoravnih potez po celi širini ograje — število teh potez naj bo kar enako višini najnižje deščice. S tem nam nepobarvani deli deščic v bistvu razpadejo na več ločenih manjših ograj, te pa nato pobarvamo z rekurzivnimi klici.

Ta rešitev torej pri vodoravnih potezah vedno razmišlja le o maksimalnem številu takih potez. Ali je lahko kakšna korist od tega, da bi naredili nekaj manj

vodoravnih potez? Prepričajmo se, da ne. Recimo, da naredimo manj kot maksimalno število vodoravnih potez; v eni vrstici (ki se razteza po celi širini ograje) torej ostanejo zaenkrat nepobarvani kvadratki v vseh celicah. Prej ali slej bo treba vse te kvadratke vendarle nekako pobarvati. Če bomo kakšnega od teh kvadratkov pobarvali z vodoravno potezo, bi lahko to potezo razširili po celi širini ograje in torej rešitve nič ne poslabšali, imela pa bi zdaj eno vodoravno potezo po celi širini več kot prej (ta razmislek lahko ponavljamo, dokler ne dosežemo maksimalnega števila vodoravnih potez). Če pa bomo vsakega od tistih kvadratkov pobarvali z navpično potezo, lahko te poteze podaljšamo po višini tako, da bodo pobarvale sploh celo ograjo, to možnost pa že tako ali tako preverjamo posebej (takrat sploh nima smisla uporabljati nobenih vodoravnih potez).

Enak razmislek kot v prejšnjem odstavku nam pove tudi, da je smiselno vodoravne poteze, dokler je to mogoče, uporabljati na dnu ograje, kjer se lahko taka poteza dotakne vseh deščic, ne pa višje gor, kjer bi se dotaknila mogoče le nekaterih.

```
#include <vector>
using namespace std;

// Vrne minimalno število potez, s katerimi lahko pobarvamo deščice
// od a do b − 1, če so bile do višine zePobarvano že pobarvane.
int SteviloPotez(const vector<int>& ograja, int a, int b, int zePobarvano)
{
  if (b <= a) return 0;

  // Poiščimo najnižjo deščico na tem območju.
  int najnizja = ograja[a];
  for (int i = a + 1; i < b; i++) if (ograja[i] < najnizja) najnizja = ograja[i];

  // Vse do te višine pobarvajmo z vodoravnimi potezami.
  int stPotez = najnizja − zePobarvano;

  // Nepobarvano območje nam zdaj razpade na več ločenih delov
  // (pri vsaki deščici z višino „najnizja“), ki jih bomo obdelali rekurzivno.
  for (int aa = a; aa < b; )
  {
    // Poiščimo naslednjo nizko deščico.
    int bb = aa; while (bb < b && ograja[bb] > najnizja) bb++;

    // Vse od trenutne do tik pred naslednjo nizko deščico pobarvajmo
    // z rekurzivnim klicem.
    if (aa < bb) stPotez += SteviloPotez(ograja, aa, bb, najnizja);

    // Nadaljujmo za naslednjo nizko deščico.
    aa = bb + 1;
  }
  // Razmislimo še o možnosti, da vse pobarvamo z navpičnimi potezami.
  return (stPotez < b − a) ? stPotez : b − a;
}

// Vrne minimalno število potez, s katerimi lahko pobarvamo celotno ograjo.
int SteviloPotez(const vector<int>& ograja)
  { return SteviloPotez(ograja, 0, ograja.size(), 0); }
```

Kakšna je časovna zahtevnost te rešitve? Recimo, da opazujemo nek rekurzivni klic, ki se nanaša na podzaporedje dolžine d. Če iz njega nastane kaj vgnezdenih klicev, imajo njihova podzaporedja vsa skupaj dolžino kvečjemu $d - 1$, saj najnižja deščica ne nastopa v nobenem vgnezdenem klicu. Če to seštejemo po vseh klicih

na isti globini rekurzije, vidimo, da je skupna dolžina vseh klicev na nekem nivoju manjša kot na prejšnjem nivoju. Začeli smo z glavnim klicem dolžine n (za celo ograjo), torej gre lahko rekurzija največ $n-1$ nivojev globoko, na vsakem nivoju pa je skupna dolžina podzaporedij pri vseh klicih le $O(n)$. Časovna zahtevnost posameznega klica za podzaporedje dolžine d je $O(d)$ (če ne štejemo časa, ki ga porabimo v vgnezdenih klicih, ki so nastali iz njega), ker se dvakrat sprehodimo po opazovanem podzaporedju dolžine d: prvič, da poiščemo najnižjo deščico, in drugič, da ga razbijemo na krajša podzaporedja in sprožimo rekurzivne klice. Za vse klice na posamezni globini rekurzije torej porabimo skupno $O(n)$ časa, časovna zahtevnost celotne rešitve pa je zato $O(n^2)$.

Da bo manj pisanja, bomo v nadaljevanju višine deščic označili z $v_0, \ldots, v_{n-1}$.

Gornjo rešitev lahko še malo poenostavimo: vrednosti zePobarvano nam ni treba prenašati kot parametra pri rekurzivnih klicih, saj si jo lahko vsak klic izračuna sam: to je preprosto maksimum višin obeh deščic, ki zamejujeta naše opazovano podzaporedje, torej v_{a-1} in v_b. (Za potrebe tega razmisleka si mislimo $v_{-1} = v_n = 0$.) Obenem velja tudi, da so vse višine od v_a do vključno v_{b-1} večje ali enake od zePobarvano.

O obojem se lahko prepričamo z indukcijo. Pri glavnem klicu (za $a = 0$ in $b = n$) ima zePobarvano vrednost 0, kar je res enako kot $\max\{v_{a-1}, v_b\} = \max\{v_{-1}, v_n\} = \max\{0, 0\} = 0$. Poleg tega so vse višine $v_0, \ldots, v_{n-1}$ večje ali enake 0, torej oba dela naše trditve držita.

Recimo zdaj, da naša trditev drži za nek klic, ki je opazoval podzaporedje od a do $b-1$; vrednost zePobarvano pri tem klicu je torej $z = \max\{v_{a-1}, v_b\}$, vse višine od v_a do v_{b-1} pa so $\geq z$. Višina, do katere je ta klic pobarval svoje deščice z dodatnimi vodoravnimi potezami, je $z' = \min\{v_a, v_{a+1}, \ldots, v_{b-1}\} \geq z$; to vrednost potem tudi poda kot zePobarvano svojim neposredno vgnezdenim rekurzivnim klicem. Poglejmo poljubnega od teh klicev, recimo od a' do $b'-1$. Na eni strani ga gotovo zamejuje deščica višine z', na drugi pa bodisi še ena deščica višine z' ali pa ena od deščic $a-1$ in b, ki sta zamejevali že nadrejeni rekurzivni klic. Za slednji dve po induktivni predpostavki velja, da nista višji od z, ta pa, kot smo videli, ni višji od z'. Pri rekurzivnem klicu od a' do $b'-1$ bo torej višja od obeh zamejujočih deščic gotovo tista, ki je visoka z'. Prav to pa je tudi vrednost, ki jo je ta rekurzivni klic dobil kot parameter zePobarvano, kar smo tudi hoteli dokazati. Poleg tega podzaporedje od a' do $b'-1$ gotovo ne vsebuje nobene deščice z višino z' (ker bi ga sicer že pri njej prekinili), torej so vse deščice v njem visoke več kot z', s čimer je dokazan tudi drugi del naše trditve. □

Razmislimo še o učinkovitejšem postopku za računanje minimalnega števila potez s čopičem. Oglejmo si za začetek primer iz besedila naloge. Začnemo z glavnim klicem, ki pokriva celotno ograjo:

$$[2 \quad 4 \quad 1 \quad 3 \quad 3 \quad 5 \quad 2 \quad 2 \quad 4 \quad 2]$$

Najnižja deščica, 1, razbije to zaporedje na dve podzaporedji, ki predstavljata vgnezdena klica:

$$[2 \quad 4] \quad 1 \quad [3 \quad 3 \quad 5 \quad 2 \quad 2 \quad 4 \quad 2]$$

Naslednja najnižja višina je 2; za vsako deščico s to višino poglejmo, v katerem podzaporedju leži, in ga razbijmo okoli nje (če je del podzaporedja levo ali desno

od deščice, pri kateri ga razbijamo, prazen, takega praznega podzaporedja ne bomo pisali, saj tudi rekurzivnega klica zanj ni treba izvajati):

2 [4] 1 [3 3 5] 2 2 [4] 2

Zdaj podobno naredimo z vsako deščico višine 3:

2 [4] 1 3 3 [5] 2 2 [4] 2

Nato z vsako deščico višine 4:

2 4 1 3 3 [5] 2 2 4 2

Ko edino preostalo podzaporedje razbijemo še pri deščici višine 5, nam ne ostane nobeno podzaporedje več in postopek je končan:

2 4 1 3 3 5 2 2 4 2

Ko vidimo te stvari takole napisane, nam lahko pride na misel, da bi takšna zaporedja vzdrževali v nekakšnem seznamu. Pri zgornjem primeru smo začeli z enim samim zaporedjem, ki je pokrivalo celo ograjo, in smo ga potem drobili na krajša podzaporedja. Toda še bolj koristno bi bilo iti v obratni smeri: začeti s krajšimi podzaporedji in jih počasi združevati v daljša, kajti preden rešimo problem za neko daljše podzaporedje, moramo poznati rešitve krajših podzaporedij, na katera to razpade, ko ga razrežemo pri najnižji deščici (oz. deščicah, če je v njem več enako visokih najnižjih deščic).

Začnimo torej s praznim seznamom podzaporedij; to ustreza stanju na koncu gornjega primera. Nato obravnavajmo deščice eno po eno od višjih proti nižjim. Recimo, da je trenutna deščica i z višino v_i. Poglejmo, če obstaja v našem seznamu kako podzaporedje, ki se konča z deščico $i-1$, in če obstaja kako podzaporedje, ki se začne z deščico $i+1$. Če obstajata obe, ju moramo združiti v eno (ki zdaj pokrije tudi deščico i); če obstaja le eno od njiju, ga podaljšajmo še na deščico i; če ne obstaja nobeno, pa ustvarimo novo podzaporedje, ki pokriva le deščico i.

Pri vsakem podzaporedju hranimo tudi najmanjše število potez, s katerim se ga dá pobarvati. Vprašanje je zdaj, kako te rezultate skombinirati, ko krajša podzaporedja združujemo v daljša. Spomnimo se, kako je to počela naša prvotna rekurzivna rešitev; recimo, da imamo neko daljše podzaporedje, ki smo ga razbili na več krajših in iz rekurzivnih klicev dobili rezultate $r_1, \ldots, r_k$; rezultat za naše daljše podzaporedje je potem

$$\min\{b-a, (r_1+\ldots+r_k)+(\mathit{najnižja}-\mathit{žePobarvano})\}.$$

Tega ne moremo dokončno izračunati, dokler ne vemo, kako daleč nas bo združevanje podzaporedij pripeljalo (šele takrat bomo poznali b in znali potem iz a in b tudi izračunati *žePobarvano*). Zato bomo izračun rezultata za daljše podzaporedje razdelili na dva dela: vsoto $r_1+\ldots+r_k$ bomo računali sproti, med združevanjem krajših podzaporedij; preostanek izračuna pa bomo opravili šele, ko bomo vedeli, da se (pri trenutni višini deščic) naše podzaporedje ne bo več podaljševalo. Temu drugemu koraku bomo rekli, da podzaporedje *zaključimo*. Da bomo lažje vedeli,

kdaj smemo podzaporedje zaključiti, je koristno, če deščice enake višine obravnavamo od leve proti desni. Podzaporedje, v katero je prišla trenutna deščica, lahko zaključimo, če je naslednja deščica nižja od nje (ali pa če naslednje sploh ni, ker je trenutna zadnja) ali pa če naslednja ne bo prišla v isto podzaporedje kot trenutna.

Vidimo torej, da bomo morali znati preverjati, ali za dano deščico i obstaja v seznamu kakšno podzaporedje z začetkom pri $i+1$ ali s koncem pri $i-1$. Poleg tega bomo morali podzaporedja v seznam dodajati, jih brisati in včasih kakšnemu premakniti krajišče. Za vse to pravzaprav ne potrebujemo seznama, ampak je bolje vzeti razpršeno tabelo, v kateri bosta za vsako podzaporedje dva ključa, eden za levi konec in eden za desni konec podzaporedja, pripadajoča vrednost pa bo kazalec na majhno podatkovno strukturo, ki hrani podatke o tem podzaporedju. Tako bomo lahko podzaporedja dodajali, brisali, spreminjali in združevali v $O(1)$ časa; z vsako deščico imamo tako le $O(1)$ dela, zato je časovna zahtevnost glavnega dela našega postopka $O(n)$. Pred tem pa potrebujemo še $O(n \log n)$ časa, da deščice uredimo po višini, zato je časovna zahtevnost naše rešitve skupaj $O(n \log n)$.

```
#include <unordered_map>
#include <utility>
#include <memory>
#include <algorithm>
#include <vector>
using namespace std;

struct Interval
{
  int a, b, najnizja, stPotez;
  Interval(int A, int B, int N) : a(A), b(B), najnizja(N), stPotez(0) { }

  void Zakljuci(const vector<int>& ograja) {
    int zePobarvano = max((a == 0 ? 0 : ograja[a - 1]),
                          (b == ograja.size() ? 0 : ograja[b]));
    stPotez = min(b - a, najnizja - zePobarvano + stPotez); }
};

int SteviloPotez2(const vector<int>& ograja)
{
  // Pripravimo pare (ograja[i], i) in jih uredimo padajoče po višini.
  typedef pair<int, int> IntPr;
  vector<IntPr> pari; pari.resize(ograja.size());
  for (int i = 0; i < ograja.size(); i++) pari[i] = {ograja[i], i};
  sort(pari.begin(), pari.end(), [] (const auto & x, const auto & y) {
    return x.first > y.first || x.first == y.first && x.second < y.second; });

  // Razpršena tabela za krajišča naših podzaporedij.
  enum { Levo = 0, Desno = 1 };
  struct PairHash {
    int operator ()(const IntPr& pr) const { return 2 * pr.first + pr.second; } };
  unordered_map<IntPr, shared_ptr<Interval>, PairHash> M;

  shared_ptr<Interval> I = nullptr, ID; // I je podzaporedje, v katero je šla prejšnja deščica.
  int vPrej = -1; // Višina prejšnje deščice (da vemo, kdaj zaključiti podzaporedje I).

  for (const auto [vi, i] : pari)
  {
    // Če je treba, zaključimo dosedanje podzaporedje I in poiščimo
    // tisto podzaporedje (če obstaja), ki leži tik levo ob deščici i.
    if (I && (vi < vPrej || I->b != i)) { I->Zakljuci(ograja); I = nullptr; }
```

```
    if (! I) if (auto it = M.find({i, Desno}); it != M.end()) I = it->second;
    // Poiščimo tisto podzaporedje (če obstaja), ki leži tik desno ob deščici i.
    if (auto it = M.find({i + 1, Levo}); it == M.end()) ID = nullptr;
    else ID = it->second;
    if (I) { // Če obstaja levo podzaporedje, se mu deščica i pridruži.
      M.erase({I->b, Desno}); I->b = i + 1; M.insert({{I->b, Desno}, I});
      I->najnizja = vi;
      if (ID) { // Če obstaja tudi desno podzaporedje, se združita v eno.
        I->stPotez += ID->stPotez;
        M.erase({I->b, Desno}); M.erase({ID->a, Levo});
        M.at({ID->b, Desno}) = I; I->b = ID->b; }}
    else if (ID) { // Če obstaja le desno podzaporedje, se deščica i pridruži njemu.
      I = ID; M.erase({I->a, Levo});
      I->a = i; M.insert({{I->a, Levo}, I});
      I->najnizja = vi; }
    else { // Sicer pa deščica i zaenkrat tvori sama svoje podzaporedje.
      I = make_shared<Interval>(i, i + 1, vi);
      M.insert({{I->a, Levo}, I}); M.insert({{I->b, Desno}, I}); }
    vPrej = vi;
  }
  if (I) I->Zakljuci(ograja); // Ostalo je eno samo podzaporedje; zaključimo ga
  return M.at({0, Levo})->stPotez; // in vrnimo potrebno število potez iz njega.
}
```

9. Krojač

Naj bo $f(y)$ najboljša rešitev, s katero pridemo od začetnega x do izbranega y.

Če je $y < 2x$, do njega ne moremo priti s korakom $\times 2$, torej moramo uporabljati le $+1$; takrat je $f(y) = y - x$.

Če je y lih, do njega tako ali tako ne moremo priti s korakom $\times 2$, torej potrebujemo $+1$; takrat je torej $f(y) = 1 + f(y-1)$.

Če je y sod, lahko do njega pridemo s korakom $\times 2$; tedaj je $f(y/2)+1$ kandidat za $f(y)$. Če bi namesto tega uporabili korak $+1$, bi to pomenilo, da smo do y prišli iz $y-1$, ki je lih, torej smo tudi do njega prišli s $+1$; tako bi imeli $f(y-2)+2$ korakov. Ali je to lahko pri kakšnem y bolje od $f(y/2)+1$? Mislimo si najmanjši y, pri katerem je to res. Torej je $f(y-2)+2 < f(y/2)+1$. Zaradi predpostavke, da je y najmanjše sodo število, do katerega se pride s $+1$ namesto $\times 2$, lahko sklepamo, da se do $y-2$ (ki je tudi sodo) pride s $\times 2$; tako imamo $f(y-2) = 1+f((y-2)/2) = 1+f(y/2-1)$. Če to nesemo v prejšnjo neenakost, dobimo $f(y/2-1)+2 < f(y/2)$. Desna stran je naprej $\leq f(y/2-1)+1$, saj za vsak z, tudi za $z = y/2$, velja $f(z) \leq f(z-1)+1$. Tako smo dobili protislovje $f(y/2-1)+2 < f(y/2-1)+1$.

Dosedanji razmislek lahko zapišemo s preprostim podprogramom:

```
int StPotez(int x, int y)
{
  if (y < 2 * x) return y - x;
  else if (y % 2 == 1) return StPotez(x, y - 1) + 1;
  else return StPotez(x, y / 2) + 1;
}
```

Vidimo lahko, kaj se pri tem algoritmu dogaja z argumentom y: dokler je $y \geq 2x$, mu na vsakem koraku odrežemo spodnji bit (če je bil ugasnjen) oz. ga najprej postavimo

na 0 in nato odrežemo (če je bil prej prižgan). Prvi parameter, x, pa se sploh ne spreminja. Ta postopek lahko torej zapišemo tudi kot zanko:

```
int StPotez2(int x, int y)
{
  int n = 0;
  while (y >= 2 * x)
  {
    n += 1 + (y % 2);
    y /= 2;
  }
  return n + (y − x);
}
```

Razmislimo zdaj še o težji različici naloge, pri kateri je poleg prištevanja 1 in podvajanja dovoljeno tudi odštevanje 1. Prej se števila ni dalo zmanjševati, zato med računanjem nikoli nismo smeli preseči končnega y; zdaj pa to smemo in je včasih to tudi res koristno. Na primer: od $x = 10$ do $y = 19$ smo prej prišli v 9 potezah (vse $+1$), zdaj pa lahko že v dveh potezah ($\times 2$ in potem -1).

Ena možnost za rešitev je še vedno ta, da potez $\times 2$ ne uporabljamo in tako od x do y pridemo z $y - x$ potezami (vse oblike $+1$). Zdaj moramo razmisliti še o takih zaporedjih potez, ki vsebujejo tudi podvajanja (operacije $\times 2$). Recimo, da imamo k podvajanj (za nek $k \geq 1$), med vsakima dvema zaporednima podvajanjema (ter pred prvim in za zadnjim) pa je neka strnjena skupina 0 ali več potez oblike $+1$ in/ali -1.

Nobene koristi ni od tega, da bi neka taka skupina vsebovala tako poteze $+1$ kot -1, saj bi v tem primeru lahko eno $+1$ in eno -1 pobrisali in tako dobili krajše zaporedje (z enakim končnim rezultatom). Skupino lahko torej enolično opišemo že s tem, da povemo vsoto vseh prištevanj oz. odštevanj v njej; naj bo n_0 taka vsota za skupino, ki nastopi pred prvim podvajanjem, in naj bo n_i taka vsota za skupino, ki nastopi za i-tim (in pred morebitnim $(i+1)$-vim) podvajanjem.

Nobene koristi tudi ni od tega, da bi imeli v skupini, ki pride za nekim podvajanjem, dve potezi $+1$ (ali dve -1), saj bi bilo tedaj bolje prišteti (ali odšteti) 1 pred podvajanjem. Tako je torej za $i > 0$ vrednost n_i lahko le $+1$, -1 ali 0, odvisno pač od tega, ali to „skupino“ tvori eno prištevanje, eno odštevanje ali pa je prazna.

Število n_0 pa je načeloma lahko poljubno celo število, kajti pred prvim podvajanjem lahko pride več prištevanj ali več odštevanj; na primer, če hočemo priti od $x = 10$ do $y = 24$, je najbolje narediti poteze $+1$, $+1$, $\times 2$, tako da imamo takrat $n_0 = 2$.

Naše zaporedje potez torej lahko popolnoma opišemo s števili $n_0, n_1, \ldots, n_k$. Izračun y torej poteka tako, da začnemo pri x, mu prištejemo n_0, pomnožimo z 2, prištejemo n_1, pomnožimo z 2 in tako naprej:

$$\begin{aligned} y &= (\cdots((x + n_0) \cdot 2 + n_1) \cdot 2 + \ldots + n_{k-1}) \cdot 2 + n_k \\ &= 2^k(x + n_0) + 2^{k-1}n_1 + 2^{k-2}n_2 + \ldots + 2n_{k-1} + n_k \\ &= 2^k(x + n_0) + \textstyle\sum_{i=1}^{k} 2^{k-i-1}n_i. \end{aligned}$$

Da bo manj pisanja, vpeljimo oznako $\tilde{n}_i = n_{k-1-i}$; vsota $\sum_{i=1}^{k} 2^{k-i-1}n_i$ tako postane $\sum_{i=0}^{k-1} 2^i\tilde{n}_i$. Če hočemo rešitev s čim manj potezami, moramo torej $z :=$

$y-2^k(x+n_0)$ izraziti z vsoto $\sum_{i=0}^{k-1} 2^i\tilde{n}_i$ tako, da bo čim več izmed števil $\tilde{n}_0,\ldots,\tilde{n}_{k-1}$ enakih 0. K temu problemu (kako izbrati $\tilde{n}_0,\ldots,\tilde{n}_{k-1}$) se bomo vrnili kasneje, za zdaj pa nadaljujmo z razmišljanjem o tem, kako izbrati n_0 in k.

Ker v izražavi $z=\sum_{i=0}^{k-1} 2^i\tilde{n}_i$ nastopajo potence števila 2 le do 2^{k-1}, je absolutna vrednost vsote na desni gotovo manjša od 2^k; torej je taka izražava možna le, če je $|z|<2^k$. Če upoštevamo, da je $z=y-2^k(x+n_0)$, nas pogoj $|z|<2^k$ pripelje do $u-1<n_0<u+1$ za $u=y/2^k-x$. Ker mora biti n_0 celo število, sta primerni vrednosti n_0 torej le $\lfloor u\rfloor$ in $\lceil u\rceil$.

Pri $k>\log_2 y$ je $2^k>y$ in $y/2^k$ leži med 0 in 1, tako da je takrat $\lfloor u\rfloor=x$ in $\lceil u\rceil=x+1$. Od tam naprej torej k-ja nima smisla več povečevati, saj se $\lfloor u\rfloor$ in $\lceil u\rceil$ ne bosta več spreminjala, zato se tudi rezultati ne bodo več spreminjali.

Naše dosedanje razmišljanje lahko strnemo v naslednji postopek:

funkcija NajmanjšeŠteviloPotez(x, y):
  $naj := y-x$;
  for $k:=0$ **to** $\lceil\log_2 y\rceil$:
    $u := y/2^k - x$;
    za $n_0\in\{\lfloor u\rfloor,\lceil u\rceil\}$:
      $z := y-2^k(x+n_0)$;
      poišči tako izražavo $z=\sum_{i=0}^{k-1}2^i\tilde{n}_i$, pri kateri so
        vsi $\tilde{n}_i\in\{1,0,-1\}$ in jih je čim manj neničelnih;
      $kand := k+|n_0|+\sum_{i=0}^{k-1}|\tilde{n}_i|$;
      if $kand<naj$ **then** $naj:=kand$;
  return naj;

Pri vsakem k torej preizkusimo možnosti $n_0=\lfloor u\rfloor$ in $n_0=\lceil u\rceil$, za vsako od teh pa izračunamo z in potem zanj poiščemo najboljšo izražavo z n_i-ji (torej táko, ki ima najmanj neničelnih n_i-jev). Skupno število potez pri tem scenariju je: k podvajanj (operacij $\times 2$); $|n_0|$ prištevanj ali odštevanj pred prvim podvajanjem; in nato še največ eno prištevanje ali odštevanje po vsakem od ostalih k podvajanj, kar ustreza številu neničelnih vrednosti v $\tilde{n}_0,\ldots,\tilde{n}_{k-1}$. Med tako dobljenimi kandidati za rešitev si zapomnimo najmanjšega in ga na koncu vrnemo.

Za učinkovito implementacijo opisanega postopka je koristno razmisliti še o nekaterih podrobnostih. Zapišimo y po bitih kot $y=(y_m y_{m-1}\ldots y_1 y_0)_2$. Naj bo t število ugasnjenih bitov na koncu (lahko je tudi $t=0$, če je y lih); torej je $y_0=y_1=\ldots=y_{t-1}=0$ in $y_t=1$. Izberimo si zdaj nek k; število y lahko zdaj v mislih razdelimo na spodnji del $s_k=(y_{k-1}\ldots y_0)_2$ (spodnjih k bitov števila y), in zgornji del $g_k=(y_m\ldots y_k)_2$ (preostanek števila y), tako da je $y=2^k g_k+s_k$. Potem je $\lfloor u\rfloor=g_k-x$ in $\lceil u\rceil=g_k-x+1$ (razen če je $s_k=0$, tedaj pa je $\lceil u\rceil=\lfloor u\rfloor=g-x_k$; to, da je $s_k=0$, velja za $k=0,\ldots,t$, kasneje pa nikoli več). Za n_0 takrat torej preizkusimo vrednost g_k-x in mogoče še g_k-x+1. Če to dvoje nesemo v formulo za z, torej $z=y-2^k(x+n_0)$, nam pri $n_0=g_k-x$ nastane $z=s_k$, pri $n_0=g_k-x+1$ pa $z=s_k-2^k$. Slednji z je gotovo manjši od 0; enaka izražava $z=\sum_{i=0}^{k-1}2^i\tilde{n}_i$ je potem mogoča tudi pri $-z$, le vsi $\tilde{n}_i$ se pomnožijo z -1. Tako lahko torej za z namesto s_k-2^k uporabimo $s'_k:=2^k-s_k$, ki je gotovo pozitiven.

Kako se vrednosti s_k in s'_k spreminjajo, če počasi povečujemo k za 1? Število s_{k+1} se razlikuje od s_k le po tem, da pridobi na levem koncu še en bit z vrednostjo

y_k. Glede s' je stvar malo bolj zapletena. Naj bo $\tilde{s}_k$ število, ki nastane, če v s_k vsak bit obrnemo (spremenimo ničle v enice in obratno); torej $\tilde{s}_k = 2^k - 1 - s_k$. Potem je $s'_k = \tilde{s}_k + 1$. Kot smo omenili že v prejšnjem odstavku, nas bo s'_k zanimal le pri $k > t$. Spomnimo se, da se y konča na t ničel, pred njimi pa je enica; torej se pri $k > t$ konča $\tilde{s}_k$ na t enic, pred njimi pa je ničla; ko mu prištejemo 1, se vse tiste enice spremenijo v ničle, ničla pred njimi v enico, prenos naprej pa se s tem konča. Tako torej vidimo, da se vrednosti s'_k začnejo pri $k = t + 1$ z vrednostjo $s'_{t+1} = 2^t$, od tam naprej pa dobimo s'_{k+1} iz s_k tako, da mu na levi pritaknemo še en bit z vrednostjo $1 - y_k$.

Vidimo torej, da ko se k poveča za 1, se vrednosti z, za kateri moramo iskati izražavo $z = \sum_{i=0}^{k-1} 2^i \tilde{n}_i$, spremenita le tako, da pridobita na levi strani vsaka po en bit — ena pridobi y_k, druga pa $1 - y_k$. Lepo pri tem je, da nam (kot se bo izkazalo) takšne izražave zato ni treba iskati vsakič od začetka in pri vsakem k porabiti zanjo $O(k)$ časa, pač pa lahko nadaljujemo postopek za iskanje takšne izražave tam, kjer se je pri $k - 1$ končal.

Razmislimo torej zdaj o tem, kako najti izražavo $z = \sum_{i=0}^{k-1} 2^i \tilde{n}_i$ s čim manj neničelnimi $\tilde{n}_i$. Na vsoto vplivajo le tisti členi, ki imajo $\tilde{n}_i \neq 0$; te pa lahko razdelimo na dve skupini glede na to, ali je $\tilde{n}_i$ enak $+1$ ali -1. Tako dobimo $z = P - N$ za $P = \sum_{i:\tilde{n}_i=1} 2^i$ in $N = \sum_{i:\tilde{n}_i=-1} 2^i$. Če ta izraz predelamo v $z + P = N$, si lahko naš problem (iskanja izražave s čim manj neničelnimi $\tilde{n}_i$) predstavljamo takole: z-ju želimo prišteti nek P tako, da bo skupno število prižganih bitov v P-ju in v vsoti $z + P$ čim manjše.

Na primer, če imamo v z-ju skupino več zaporednih prižganih bitov, jih lahko spremenimo v enega samega, če v P prižgemo bit na najnižjem mestu te skupine:

$$\begin{array}{lll} z & & \ldots 00\ 11111\ 00\ldots \\ P & + & \ldots 00\ 00001\ 00\ldots \\ \hline N & = & \ldots 01\ 00000\ 00\ldots \end{array}$$

V gornjem primeru smo imeli v z skupino 5 strnjenih enic, enako pa bi seveda lahko naredili pri skupini poljubne dolžine; v vsakem primeru smo porabili za obdelavo te skupine dve enici (eno v P-ju in eno v N-ju). Le če bi „skupina“ v z-ju obsegala eno samo enico, se to ne bi splačalo; takrat je bolje pustiti v P-ju na tistem mestu ničlo in v N-ju enico (tako imamo tu skupaj eno enico, ne pa dveh).

Če imamo v z-ju več skupin enic, ločenih s po eno samo ničlo, lahko to rešitev še izboljšamo:

$$\begin{array}{lll} z & & \ldots 00\ 11111\ 0\ 111\ 0\ 1\ 0\ 1111\ 00\ldots \\ P & + & \ldots 00\ 00000\ 1\ 000\ 1\ 0\ 1\ 0001\ 00\ldots \\ \hline N & = & \ldots 01\ 00000\ 0\ 000\ 0\ 1\ 0\ 0000\ 00\ldots \end{array}$$

Ideja je torej v tem: če v mislih sledimo seštevanju od desne proti levi, vidimo, da kjer bi zaradi prenosa iz prejšnje skupine enic z-ja nastala v N-ju enica (pod ničlo iz z-ja), postavimo enico v P; zato potem dobimo v N-ju ničlo (tako da s tem nismo nič na slabšem), obenem pa pride do prenosa na naslednje mesto, kjer se v z-ju že začne naslednja skupina enic; zato nam tam ni treba postaviti enice v P, saj enak učinek nastane že zaradi prenosa s prejšnjega mesta. Tako nam ena enica (v P-ju) hkrati služi za konec prejšnje skupine in začetek naslednje. Za q skupin smo

na ta način porabili q enic v P-ju in eno v N-ju, torej skupaj $q+1$ enic (če pa bi obdelali vsako skupino posebej z dvema enicama, bi porabili $2q$ enic). Kot vidimo že iz gornjega primera, deluje ta pristop tudi, če imajo nekatere skupine samo eno enico. Le če je *vsaka* skupina sestavljena iz ene same enice, se to ne splača in je bolje v P-ju pri vseh pustiti ničle, pri N-ju pa enice (tako porabimo q enic namesto $q+1$).

Tak blok več skupin enic, ločenih s po eno ničlo, je od sosednjih blokov razmejen s po vsaj dvema ničlama na vsaki strani (če bi bila med blokoma le ena ničla, bi ju oba skupaj šteli za en sam večji blok). Ni se težko prepričati, da lahko obravnavamo vsak blok posebej. Pri seštevanju po bitih lahko vpliva en blok na drugega le tako, da pride do prenosa 1 iz nižjega (desnega) bloka v višjega (levega). Toda če je vmes v z-ju recimo $t \geq 2$ ničel, lahko ta prenos nadaljujemo čez teh t mest le tako, da v P-ju na vsa ta mesta postavimo enice; tako porabimo t enic, kar je slabše, kot če bi imeli v P-ju tam ničle, v N-ju pod najnižjo od obeh ničel enico, nato pa bi postavili še enico v P na najnižje mesto naslednjega bloka (kar bi dalo tam enak učinek, kot če bi prišel do tja prenos iz nižjega bloka). Primer (pri razmišljanju o spodnjem primeru seveda ne pozabimo, da je mišljeno, da v nižjem bloku prihaja do prenosa pri seštevanju zaradi neke nižje ležeče enice v P-ju):

		višji blok	ničle	nižji blok			višji blok	ničle	nižji blok
z		...111	00000	111...			...111	00000	111...
P	+	...000	11111	000...		+	...001	00000	000...
N	=	...000	00000	000...		=	...000	00001	000...

V levem primeru smo porabili $t = 5$ enic (vse v P), v desnem pa le 2 enici (neodvisno od t; eno v P in eno v N).

Najmanjše število potrebnih enic (v P in N skupaj) torej izračunamo tako, da razdelimo z na bloke, za vsak blok pa preštejemo skupine enic v njem in pogledamo, če kakšno skupino tvorita vsaj dve enici; če da, potrebujemo za blok s q skupinami $q+1$ enic, sicer pa le q enic. Pravzaprav nam ni treba računati celega q; moramo si le zapomniti, ali smo v trenutnem bloku že videli kakšno skupino vsaj dveh enic ali ne. Na začetku vsake skupine povečamo števec potrebnih enic za 1, ko pa v bloku prvič vidimo skupino vsaj dveh enic, povečamo števec še za 1. Zapišimo ta postopek s psevdokodo; recimo, da je z v dvojiškem zapisu oblike $(z_{k-1}z_{k-2}\ldots z_1z_0)_2$.

```
funkcija KolikoEnic(z):
  r := 0; b := 0; p := 0;
  for i := 0 to k − 1:
    if z_i = 1:
      if p = 0 then r := r + 1; (* začetek skupine *)
      if b = 0 then b := 1; (* začetek bloka *)
      if b = 1 and p = 1 then
        (* Prvič v trenutnem bloku vidimo skupino vsaj dveh enic. *)
        b := 2; r := r + 1;
    else: (* torej če je z_i = 0 *)
      if p = 0 then b := 0; (* smo zunaj bloka *)
    p := z_i;
```

```
  return r;
```

Spremenljivka p torej hrani prejšnji bit z-ja, spremenljivka b pa pove, ali smo trenutno med dvema blokoma ($b = 0$) ali v bloku, ki doslej še ni imel skupine vsaj dveh enic ($b = 1$), ali pa v bloku, ki je že imel tako skupino ($b = 2$).[8]

Prej smo videli, da bomo morali funkcijo KolikoEnic klicati za števila s_k in s'_k, pri čemer se, ko k narašča, ta števila razlikujejo tako, da vsakič na levi pridobijo po en bit. Vidimo lahko, da v ta namen v KolikoEnic ni treba izvajati cele zanke po i vsakič od 0 naprej; lahko si le zapomnimo vrednosti z, b in p, s katerimi se je končala prejšnja iteracija, nato pa, ko se nam s_k (ali s'_k) podaljša z novim bitom na levi, izvedemo naslednjo iteracijo tiste zanke in si zapomnimo nove vrednosti z, b in p. V ta namen si lahko omislimo preprost razred, ki hrani trenutni rezultat (najmanjše potrebno število enic v P in N skupaj — ali za drugimi besedami: najmanjšo možno vsoto $\sum_i |\tilde{n}_i|$) v polju r, naslednji bit števila s_k oz. s'_k pa mu bomo podajali z operatorjem $\ll$:

```
struct IskanjeIzrazave
{
  int r = 0, b = 0, p = 0;
  IskanjeIzrazave& operator << (int bit) {
    if (bit) {
      if (! p) r++;                              // Začetek skupine.
      if (b == 0) b = 1;                         // Začetek bloka.
      if (b == 1 && p) { b = 2; r++; } }         // Prva skupina dveh enic v bloku.
    else if (! p) b = 0;                         // Smo zunaj bloka.
    p = bit; return *this; }
};
```

Zdaj lahko zapišemo še podprogram, ki bo ta razred uporabljal. Pri tem moramo le slediti psevdokodi postopka NajmanjšeŠteviloPotez, ki smo ga videli zgoraj, in upoštevati naša dosedanja razmišljanja o njegovi učinkoviti implementaciji.

```
int NajmanjseSteviloPotez(int x, int y)
```

[8] Z enakim problemom kot tukaj — torej kako izraziti z kot vsoto $\sum_i 2^i \tilde{n}_i$, pri čemer morajo biti $\tilde{n}_i \in \{1, 0, -1\}$ in jih mora biti čim manj neničelnih — smo se na naših tekmovanjih že srečali: to je bila naloga Tehtnica na šolskem tekmovanju leta 2015. Takrat smo do rešitve prišli po malo drugačni poti, čeprav so rezultati seveda v vsakem primeru enaki. Povezavo med tedanjo rešitvijo in našo tokratno lahko opazimo, če si ogledamo diagram stanj na str. 83 v *Biltenu* 2015 in opazimo naslednje: (1) posebnega začetnega stanja niti ne potrebujemo, lahko bi ga ukinili in namesto tega začeli v stanju $(r, r+1)$ z $r = 0$; (2) stanje $(r+1, r)$ lahko preimenujemo v $(r, r-1)$, če operacijo $r := r + 1$ premaknemo s prehoda $(r + 1, r) \to (r, r)$ na prehod $(r, r) \to (r, r - 1)$; (3) posledica te spremembe je, da postopek vedno vrača r (tako kot naša letošnja rešitev), nikoli $r + 1$; (4) stanje $(r, r + 1)$ potem pomeni, da smo trenutno ali v praznem prostoru med dvema blokoma ali pa smo v bloku, v katerem doslej še nismo videli skupine vsaj dveh enic; (5) stanje $(r, r - 1)$ potem pomeni, da smo trenutno v bloku, v katerem smo že videli skupino vsaj dveh enic; (6) za prehod iz $(r, r - 1)$ v $(r, r + 1)$ moramo torej videti dve zaporedni ničli, v obratno smer pa dve zaporedni enici; in ker tak prehod zahteva dva koraka, potrebujemo še eno vmesno stanje — temu je namenjeno stanje (r, r).

Za namen naše letošnje naloge je pomembno, da lahko vhodno število obdelujemo od nižjih bitov proti višjim, postopek iz leta 2015 pa jih je obdeloval od višjih proti nižjim in pri njem na prvi pogled ni tako očitno, ali ga smemo uporabiti v nasprotni smeri. Naš letošnji razmislek z bloki in skupinami enic je koristen tudi s tega vidika, saj je obstoj blokov in skupin neodvisen od tega, ali beremo bite od leve proti desni ali od leve proti desni; to pomeni, da če v z-ju obrnemo vrstni red bitov, bo rezultat enak, kar tudi pomeni, da vsak postopek za reševanje tega problema daje enake rezultate tudi, če bere vhodno število z desne proti levi namesto od desne proti levi.

```
{
  Iskanjelzrazave izrS, izrS2;
  int naj = y − x, g = y; bool s0 = true;
  for (int k = 0; g; k++)
  {
    // g vsebuje g_k = ⌊y/2^k⌋; s0 pove, ali je s_k = y mod 2^k enak 0.
    int yk = g & 1;
    // Preizkusimo n_0 = ⌊u⌋ = ⌊y/2^k − x⌋ = g_k − x.
    int n0 = g − x; naj = min(naj, k + abs(n0) + izrS.r);
    // Preizkusimo n_0 = ⌈u⌉, če u ni cel (= če je s_k > 0).
    if (! s0) { n0++; naj = min(naj, k + abs(n0) + izrS2.r); }
    // Popravimo izražavi za s_k in s'_k v izražavi za s_{k+1} in s'_{k+1}.
    g >>= 1; izrS <<= yk;
    if (! s0) izrS2 <<= (1 − yk);
    else if (yk) { s0 = false; izrS2 <<= 1; }
  }
  return naj;
}
```

10. Najkrajše poti

Število točk našega grafa označimo z n, število povezav pa z m. Naj bo $d(u, v)$ dolžina najkrajše poti od u do v in naj bo $r(u, v)$ število takih poti. Če v ni dosegljiv iz u, si mislimo $d(u, v) = \infty$ in $r(u, v) = 0$. Z iskanjem v širino iz s in t poiščimo $d(s, u)$, $r(s, u)$, $d(u, t)$ in $r(u, t)$ za vse $u \in V$; te vrednosti shranimo v tabelah, saj jih bomo kasneje še potrebovali. To nam vzame $O(n + m)$ časa.

Opazimo lahko, da je vedno mogoče dodati tako povezavo, zaradi katere se število najkrajših poti od s do t poveča. Res: (1) če v prvotnem grafu ni bilo nobene poti od s do t, lahko dodamo povezavo $s \to t$ in se število najkrajših poti od s do t poveča z 0 na 1. (2) Če pa je v prvotnem grafu že obstajala neka pot od s do t, pa lahko dodamo še eno povezavo, vzporedno prvi povezavi na tej poti, in se število najkrajših poti od s do t zaradi tega tudi poveča. □

Ena od posledic gornjega opažanja je, da je smiselno dodati povezavo $x \to y$ le tedaj, če že od prej obstaja neka pot od s do x in neka pot od y do t, saj drugače ne bo potekala po novi povezavi nobena pot od s do t in se zato tudi število najkrajših poti od s do t ne bo nič spremenilo. Če je obstajala neka pot od s do t že v prvotnem grafu, lahko ta pogoj še zaostrimo: povezavo $x \to y$ je smiselno dodati le, če je $d(s, x) < d(s, t)$ in $d(y, t) < d(s, t)$, saj drugače pot, ki bi šla od s do t in uporabila novo povezavo $x \to y$, gotovo ne bi mogla biti najkrajša pot od s do t in število najkrajših poti se zaradi dodajanja nove povezave ne bi nič povečalo.

Še ena posledica je, da ni smiselno dodati povezave oblike $x \to x$ (takim povezavam pravimo *zanke*), saj čeznjo tudi ne bo tekla nobena najkrajša pot od s do t (kajti pot, ki vsebuje tako povezavo, ne more biti najkrajša, saj jo lahko skrajšamo, če to povezavo vzamemo iz nje),[9] zato se število najkrajših poti od s do t ne bo zaradi dodajanja zanke nič spremenilo, to pa gotovo ne bo najboljša možna rešitev.

Recimo torej zdaj, da dodamo povezavo $x \to y$, pri čemer je $x \neq y$ in je x dosegljiv iz s, točka t pa iz y. Zdaj so možne od s do t nove poti oblike $s \rightsquigarrow x \to$

[9] Naloga ne pove eksplicitno, ali se lahko zanke pojavljajo že v prvotnem grafu. Tudi če se pojavljajo v njem, take zanke gotovo niso in nikoli ne bodo del nobene najkrajše poti od s do t; zato se nič ne spremeni, če zanke iz grafa kar pobrišemo. V nadaljevanju bomo torej predpsotavili, da graf nima zank.

$y \rightsquigarrow t$. Dolžina najkrajše take poti je $d(s,x)+1+d(y,t)$, število takih poti te dolžine pa bo $r(s,x)\cdot r(y,t)$. V poštev pridejo le taki pari (x,y), pri katerih bo dolžina nove poti $\leq d(s,t)$, saj drugače nova pot ne bo najkrajša pot od s do t. Imamo torej pogoj $d(s,x)+1+d(y,t) \leq d(s,t)$ oz. z drugimi besedami $d(s,x)+d(y,t) < d(s,t)$.

Ali je mogoče, da tako sestavljena nova najkrajša pot ne bi bila zares pot, torej da bi kakšno točko obiskala več kot enkrat? Na delu $s \rightsquigarrow x$ se ne more nobena točka pojaviti večkrat, ker je tisto najkrajša pot od s do x; podobno tudi na delu $y \rightsquigarrow t$. Večkratno ponavljanje neke točke bi bilo torej možno le tako, da se enkrat pojavi na delu $s \rightsquigarrow x$ in enkrat na $y \rightsquigarrow t$. Recimo, da je to točka w. Imamo torej $s \rightsquigarrow w \rightsquigarrow x$ (ki je najkrajša pot od s do x v prvotnem grafu) in $y \rightsquigarrow w \rightsquigarrow t$ (ki je najkrajša pot od y do t v prvotnem grafu). Torej je v prvotnem grafu obstajala pot $s \rightsquigarrow w \rightsquigarrow t$ (recimo ji ρ), ki je gotovo krajša od poti $s \rightsquigarrow w \rightsquigarrow x$ in $y \rightsquigarrow w \rightsquigarrow t$ skupaj (edini način, da ne bi bila krajša, bi bil ta, da bi bila dela $w \rightsquigarrow x$ in $y \rightsquigarrow w$ oba dolga 0, torej bi bilo $w = x$ in $y = w$, to pa je nemogoče, saj je $x \neq y$). Dolžina poti ρ je torej $d(\rho) < d(s,x) + d(y,t)$. Toda po drugi strani, ker je ρ pot od s do t, gotovo velja $d(\rho) \geq d(s,t)$. Tako imamo $d(s,t) < d(s,x) + d(y,t)$; toda spomnimo se, da smo x in y izbrali tako, da je $d(s,x) + d(y,t) < d(s,t)$. Tako nas je predpostavka, da se na novi poti kakšna točka pojavi večkrat, pripeljala v protislovje, torej se to ne more zgoditi. □

Razmislimo še o tem, kako učinkovito poiskati tak par (x,y), pri katerem se število najkrajših poti od s do t poveča najbolj. Iščemo ga seveda le med tistimi pari, pri katerih je izpolnjen pogoj $d(s,x) + d(y,t) + 1 \leq d(s,t)$. Te pare lahko ločimo na dve skupini: če velja v tem pogoju enakost, imamo nove poti, ki so enako dolge dosedanji najkrajši poti, torej bo število najkrajših poti od s do t po novem $r(s,t)+r(s,x)\cdot r(y,t)$; če pa v pogoju velja strogo $<$, so nove poti krajše od dosedanje najkrajše, zato bo po novem število najkrajših poti od s do t le $r(s,x) \cdot r(y,t)$.

Opazimo lahko še, da pridejo za x v poštev le točke, ki so dosegljive iz s in od njega niso oddaljene za več kot $d(s,t) - 1$ (ta slednji pogoj sicer odpade, če v prvotnem grafu sploh ni bilo poti od s do t).

Z eno zanko po vseh vozliščih grafa si lahko v neki tabeli za vsako možno dolžino k od 0 do $n-1$ pripravimo $f[k] := \arg\max_u\{r(u,t) : d(u,t) = k\}$; to je torej med točkami, iz katerih je mogoče priti do t s k koraki (ne pa z manj), tista, pri katerih je takih poti največ. Nato lahko z eno zanko po tej tabeli izračunamo za vsak k tudi $g[k] := \arg\max_u\{r(u,t) : d(u,t) \leq k\}$. Zdaj pa lahko razmišljamo takole: za vsak možni x, ki je dosegljiv iz s (in od njega ni oddaljen za več kot $d(s,t)-1$), imamo pri roki njegovo $d(s,x)$ in torej vemo, da mora y ustrezati pogoju $d(y,t) \leq d(s,t) - 1 - d(s,x)$. Desni strani tega pogoja recimo k. Če nas zanimajo poti, ki so dolge natanko toliko kot dosedanja najkrajša pot, moramo za y vzeti $f[k]$, če pa nas zanimajo krajše poti, moramo za y vzeti $g[k-1]$. V resnici nam je seveda vseeno, ali je nova pot krajša od dosedanje ali ne, zato moramo preizkusiti obe možnosti in pogledati, pri kateri je število najkrajših poti večje.

Pri tem lahko nastopita dva posebna primera: (1) Če je $d(s,x) = d(s,t) - 1$, druga možnost odpade: takrat je namreč x tako daleč od s, da pot $s \rightsquigarrow x \to y \rightsquigarrow t$ zagotovo ne more biti krajša od $d(s,t)$, v najboljšem primeru je lahko enako dolga. Takrat dobimo $k = 0$, zato je vrednost $g[k-1]$ tako ali tako nedefinirana.) (2) Če v prvotnem grafu sploh ni bilo poti od s do t, odpade prva možnost (torej da je nova

pot enako dolga kot najkrajša dosedanja pot); za y je potem vseeno, kako dolga je pot od njega do t (samo da sploh obstaja), torej je smiselno vzeti tisti y, ki ima med vsemi največjo vrednost $r(y,t)$; to je $g[n-1]$.

Časovna zahtevnost te rešitve je $O(n+m)$; toliko časa porabimo za izračun tabel d in s z iskanjem v širino, nato še $O(n)$ časa za pripravo tabel f in g ter $O(n)$ časa, da za vsak x izberemo primerna y.

Razmislimo še o težji različici, pri kateri ne delamo z multigrafi, zato smemo dodati povezavo $x \to y$ le, če v prvotnem grafu take povezave še ni bilo. Ko smo si pri posameznem x izbirali y, smo si želeli tak y, od katerega bi se dalo do t priti v k ali manj korakih; med takimi y pa seveda tistega z največ potmi $r(y,t)$. Tak y smo si zapomnili v $f[k]$ (za $d(y,t)=k$) oz. v $g[k-1]$ (za $d(y,t)<k$). Po novem to ne bo dovolj, kajti povezavo $x \to y$ smemo dodati le, če je v prvotnem grafu še ni bilo; to pomeni, da tisti y, ki ima (pri nekem k) največjo vrednost $r(y,t)$, mogoče ne bo primeren (ker povezava $x \to y$ že obstaja) in bomo morali uporabiti tistega z drugo največjo vrednostjo $r(y,t)$; če niti ta ne bo primeren, pa tistega s tretjo največjo $r(y,t)$ in tako naprej.

Definirajmo torej zdaj $f[k]$ kot seznam vseh tistih y, za katere je $d(y,t)=k$, ta seznam pa naj bo urejen padajoče po vrednosti $r(y,t)$. Ko pri nekem x izbiramo primeren y z $d(y,t)=k$, se moramo sprehoditi po tem seznamu in se ustaviti pri prvem takem y, za katerega povezava $x \to y$ v prvotnem grafu ne obstaja. Lahko se tudi zgodi, da takega y sploh ni in primerne povezave iz x sploh ne moremo dodati. Pri vsakem x moramo narediti največ toliko korakov po seznamu, kolikor povezav z začetkom pri x obstaja v prvotnem grafu; skupna cena (po vseh x) vseh teh iskanj je torej $O(m)$, če je m število povezav v grafu.

Naj bo n_k število elementov v seznamu $f[k]$. Velja seveda $\sum_k n_k \le n$, ker je vsaka od n točk grafa prisotna v največ enem od teh seznamov. Urejanje vseh seznamov nam torej vzame $O(\sum_k n_k \log n_k) = O(\sum_k n_k \log n) = O(n \log n)$ časa.

Podobno lahko definiramo tudi $g[k-1]$ kot seznam vseh y z $d(y,t)<k$; tudi ta naj bo urejen padajoče po $r(y,t)$; ko pri nekem x izbiramo primeren y z $d(y,t)<k$, se moramo sprehoditi po tem seznamu, dokler ne pridemo do takega y, za katerega povezava $x \to y$ v prvotnem grafu ne obstaja.

Neugodno pa je, da se zdaj lahko isti y pojavlja v več takih seznamih (če se pojavlja v $g[k-1]$, se bo tudi v $g[k]$ in vseh naslednjih), zato je vsak seznam lahko dolg $O(n)$ elementov in urejanje vseh seznamov bi nam vzelo $O(n^2 \log n)$ časa. Boljša rešitev je, da uporabimo kakšno drevesasto strukturo; vzemimo na primer B-drevo, ki naj ima vse ključe v listih (v notranjih vozliščih pa le kopije s kazalci na poddrevesa), poleg tega pa naj bodo listi povezani v dvojno povezano verigo (*doubly linked list*). Kot ključe v B-drevesu uporabimo $r(y,t)$, pripadajoča vrednost k takemu ključu pa naj bo y. Po verigi listov se lahko sprehajamo enako, kot bi se po seznamu $g[k-1]$, zato za iskanje primernega y porabimo prav toliko časa, kot bi ga, če bi res imeli seznam $g[k-1]$.

Namesto da bi za vsak k zgradili ločeno drevo, lahko najprej začnemo s praznim drevesom in potem počasi povečujemo k ter vsakič dodamo v drevo primerne nove elemente: $g[k]$ dobimo iz $g[k-1]$ tako, da vanj dodamo tiste y, ki imajo $d(y,t)=k-1$. Ko imamo pripravljeno drevo za $g[k]$, moramo obdelati vse tiste x, ki potrebujejo prav ta k (torej tiste, ki imajo $d(s,t)-1-d(s,x)=k$; na začetku si lahko v $O(n)$ časa

pripravimo sezname, ki za vsak k povedo, pri katerih x potrebujemo ta k), potem pa drevo dopolnimo z dodatnimi y, da nastane drevo za $g[k+1]$ in tako naprej. V drevo moramo tako dodati največ n elementov, vsako dodajanje pa vzame $O(\log n)$ časa. Časovna zahtevnost celotne rešitve je tako $O(m + n \log n)$.

11. Planinci

Graf poti v naši nalogi je neusmerjen, acikličen in povezan — takemu v teoriji grafov pravijo *drevo*. Drevo smo si običajno navajeni predstavljati s korenom, iz katerega se potem navzdol širijo veje. V naši nalogi korena ni, zato si bomo za začetek izbrali poljubno točko grafa in jo razglasili za koren. Ko imamo koren, lahko vsaki točki u pripišemo *globino* g_u — to je dolžina poti od korena do te točke — in *starša* — to je neposredni predhodnik te točke na tej poti.

```
#include <vector>
#include <stack>
#include <utility>
using namespace std;

void PripraviDrevo(int n, const vector<pair<int, int>> &povezave,
                   vector<int> &globina, vector<int> &starsi)
{
  // Določimo stopnjo (število sosedov) vsake točke.
  vector<int> stopnja(n, 0), prviSosed(n), sosedje(2 * povezave.size());
  for (const auto [u, v] : povezave) { ++stopnja[u]; ++stopnja[v]; }

  // Pripravimo sezname sosedov vseh točk. Sosedje točke u bodo v vektorju
  // „sosedje“ na indeksih prviSosed[u], ..., prviSosed[u] + stopnja[u] - 1.
  prviSosed[0] = 0; for (int u = 1; u < n; u++)
    prviSosed[u] = prviSosed[u - 1] + stopnja[u - 1];
  for (int u = 0; u < n; u++) stopnja[u] = 0;
  for (const auto [u, v] : povezave) {
    sosedje[prviSosed[u] + stopnja[u]++] = v;
    sosedje[prviSosed[v] + stopnja[v]++] = u; }

  // Izberimo si koren in določimo globine in starše vseh točk.
  const int koren = 0; globina.resize(n); starsi.resize(n);
  stack<int> sklad; sklad.push(koren); starsi[koren] = koren; globina[koren] = 0;
  while (! sklad.empty()) {
    int u = sklad.top(); sklad.pop();
    for (int i = 0; i < stopnja[u]; i++) {
      int v = sosedje[prviSosed[u] + i]; if (v == starsi[u]) continue;
      sklad.push(v); globina[v] = globina[u] + 1; starsi[v] = u; } }
};
```

Koren starša sploh nima, zato smo v tabeli `starsi` označili, kot da je starš samega sebe.

V drevesu obstaja od vsake točke natanko ena pot do vsake druge točke. Naš planinec lahko pride od točke s do točke t le tako, da se iz s najprej vzpne po drevesu do najglobljega skupnega prednika točk s in t, od tam pa se spusti proti točki t. Za to, kako učinkovito poiskati najglobljega skupnega prednika, obstaja več postopkov. Lahko si na primer vnaprej za vsako točko u pripravimo podatke o tem, kateri je njen prednik 2^k nivojev nad njo — recimo mu $p_k(u)$. Potrebovali jih bomo za k-je od 0 do največ $\lceil \log_2 n \rceil$. Pri pripravi teh podatkov si lahko pomagamo z dejstvom,

da je $p_0(u)$ kar starš točke u, od tam naprej pa je $p_{k+1}(u) = p_k(p_k(u))$ — če se hočemo iz u-ja pomakniti za 2^{k+1} nivojev navzgor, se moramo najprej pomakniti za 2^k nivojev in nato od tam še za 2^k nivojev. Vse $p_k(u)$ lahko tako izračunamo v $O(n \log n)$ časa in zanje porabimo tudi $O(n \log n)$ prostora.

```
// Predpostavimo, da so podatki o starših že v predniki[0].
void PripraviPrednike(vector<vector<int>>& predniki)
{
  const int n = predniki[0].size();
  for (int k = 1; k < predniki.size(); k++) {
    predniki[k].resize(n);
    for (int u = 0; u < n; u++) predniki[k][u] = predniki[k - 1][predniki[k - 1][u]]; }
}
```

Če je u na globini manj kot 2^k, potem prednika $p_k(u)$ v resnici sploh nima; gornji podprogram bo v takem primeru v elementu predniki[k][u] shranil številko korena.

Pripravimo si strukturo, v kateri bomo hranili podatke o drevesu:

```
struct Drevo
{
  int n, logN;
  vector<int> globina;
  vector<vector<int>> predniki;

  Drevo(int N, const vector<pair<int, int>> &povezave) : n(N)
  {
    logN = 0; while (n > (1 << logN)) logN++;
    predniki.resize(logN + 1);
    PripraviDrevo(n, povezave, globina, predniki[0]);
    PripraviPrednike(predniki);
  }

  ...  // Tu bodo prišle še druge metode.

};
```

Ko imamo vse prednike $p_k(u)$, lahko v $O(\log n)$ časa poiščemo prednika točke u poljubno število nivojev nad njo: če nas na primer zanima prednik d nivojev na u-jem, moramo le izraziti d kot vsoto potenc števila 2. Pri $d = 13$ na primer vidimo, da je $13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0$, zato je u-jev prednik 13 nivojev nad njim preprosto $p_3(p_2(p_0(u)))$.

```
int Drevo::Prednik(int u, int d) const
{
  for (int k = 0; d > 0; d >>= 1, k++) if (d & 1) u = predniki[k][u];
  return u;
}
```

Če je u na globini manj kot d (in torej prednika d nivojev nad sabo sploh nima), bo ta funkcija vrnila kar koren drevesa.

Razmislimo zdaj o tem, kako učinkovito poiskati najglobljega skupnega prednika točk s in t.[10]

[10] Za problem najglobljega skupnega prednika obstajajo tudi še učinkovitejši algoritmi od tega, ki ga bomo opisali tukaj, vendar je ta prikladen za naš namen zato, ker nam bodo tabele prednikov prišle prav tudi v nadaljevanju rešitve, pri ugotavljanju, kje na poti se je planinec izgubil. Za več o problemu najglobljega skupnega prednika gl. npr. Wikipedijo *s. v.* Lowest common ancestor in tam navedeno literaturo.

Recimo, da je s globlje v drevesu kot t; poiščimo tedaj s-jevega prednika $g_s - g_t$ nivojev nad njim; recimo mu s' — to je torej prednik na isti globini kot t. Mogoče se bo izkazalo, da je ta prednik kar t sam, in v tem primeru vemo, da je t ravno tisti skupni prednik, ki ga iščemo; če pa je $s' \neq t$, potem vemo vsaj to, da je najglobljи skupni prednik točk s in t isti kot najglobljи skupni prednik točk s' in t (ki pa sta na isti globini).

V primerih, ko je t globlje v drevesu kot s, bi lahko razmišljali podobno. V nadaljevanju je torej dovolj, če se omejimo na primere, ko sta s in t na isti globini, recimo na globini g. Označimo z g_i število, ki nastane, če v dvojiškem zapisu števila g ugasnemo spodnjih i bitov; in naj bo s_i (oz. t_i) prednik s-ja (oz. t-ja) na nivoju g_i. Teh prednikov ni težko računati: ker je $g_0 = g$, je $s_0 = s$; od tam naprej pa je tako, da če je bit i v g ugasnjen, je $g_{i+1} = g_i$ in zato $s_{i+1} = s_i$, sicer pa je $g_{i+1} = g_i - 2^i$ in zato $s_{i+1} = p_i(s_i)$. Podobno je tudi s t-jevimi predniki.

Prej ali slej se pri nekem i zgodi, da je $s_i = t_i$ — če ne prej, pa pri $i = \lfloor \log_2 n \rfloor + 1$, ko se gotovo ugasnejo že vsi biti, torej je $g_i = 0$ in zato za s_i in t_i dobimo koren drevesa. Vzemimo zdaj največji i, pri katerem sta s_i in t_i še različna; pri naslednjem je torej že $s_{i+1} = t_{i+1}$. To je seveda mogoče le, če je bil bit i v g prižgan, saj bi drugače imeli $g_{i+1} = g_i$ in zato $s_{i+1} = s_i$ in $t_{i+1} = t_i$, tedaj pa bi iz $s_i \neq t_i$ sledilo tudi $s_{i+1} \neq t_{i+1}$. Zdaj torej vemo, da je $g_{i+1} > g_i$ in da najnižji skupni prednik s-ja in t-ja leži na enem od nivojev $g_i, \ldots, g_{i+1} - 1$. Med njima bomo z bisekcijo poiskali najnižji nivo, na katerem je skupni prednik še enak, pri čemer nam pride prav dejstvo, da sta se začetna nivoja, g_i in g_{i+1}, razlikovala za potenco števila 2, torej $g_{i+1} - g_i = 2^i$; zato bomo imeli tudi kasneje pri bisekciji vedno opravka z intervalom, čigar širina bo potenca števila 2. To pa pomeni, da lahko prednika na tisti višini dobimo že z enim pogledom v tabelo predniki, kar vzame $O(1)$ časa, in nam ni treba klicati funkcije Prednik, kar bi vzelo $O(\log n)$ časa. Celoten postopek iskanja najglobljega skupnega prednika tako vzame le $O(\log n)$ časa.

```
int Drevo::NajgloblјiSkupniPrednik(int s, int t) const
{
  // Poiščimo globljemu izmed s in t prednika na nivoju plitvejšega izmed s in t.
  int gs = globina[s], gt = globina[t];
  if (gs > gt) { s = Prednik(s, gs − gt); gs = gt; }
  else if (gt > gs) { t = Prednik(t, gt − gs); gt = gs; }
  if (s == t) return s;

  // Zdaj sta s in t na isti globini (in sta različna).
  int gi = gs, i = 0;
  for ( ; ; i++)
  {
    int bit = 1 << i; if ((gi & bit) == 0) continue;
    // s in t sta na globini gi in sta različna. V gi je bit i prižgan.
    // Poglejmo njuna prednika na globini gi − 2^i.
    int s1 = predniki[i][s], t1 = predniki[i][t];
    if (s1 == t1) break;
    s = s1; t = t1; gi &= bit;
  }

  // s in t sta na globini gi in sta različna, imata pa skupnega prednika na globini gi − 2^i.
  while (i−− > 0)
  {
    // s in t sta na globini gi, sta različna in imata skupnega prednika na globini gi − 2^(i+1).
```

```
    int gm = gi - (1 << i), s1 = predniki[i][s], t1 = predniki[i][t];
    if (s1 != t1) { s = s1; t = t1; gi = gm; }
  }
  // s in t sta na globini gi in sta različna, imata pa skupnega prednika na globini gi - 1.
  return predniki[0][s];
}
```

Zdaj znamo izračunati skupnega prednika točk s in t — recimo mu r. Planinec gre torej od s do r (to je $g_s - g_r$ korakov) in nato od tam do t (to je $g_t - g_r$ korakov). Dolžina te poti je $d := g_s + g_t - 2g_r$ korakov. Kolikšno je zdaj (v najslabšem primeru) najmanjše potrebno število klicev v planinske koče? Pri vsakem klicu sta dve možnosti: ali nam povedo, da je planinec že obiskal tisto kočo, ali pa, da je še ni. Po k klicih je vsega skupaj 2^k možnosti, mi pa moramo znati ločiti med seboj vsaj d različnih možnosti, saj je na planinčevi poti d povezav in se lahko ponesreči na katerikoli od njih. Tako imamo pogoj $2^k \geq d$, sicer s k klici gotovo ne bomo mogli ločiti d možnosti. Iz tega dobimo $k \geq \log_2 d$; ker mora biti k celo število, je najmanjši k, ki ustreza temu pogoju, enak $k = \lceil \log_2 d \rceil$.

Primer postopka, ki z največ toliko klici tudi res poišče planinca, je bisekcija. Naj bo u koča na polovici poti od s do t — recimo, da je $\lfloor d/2 \rfloor$ korakov naprej od s po tej poti. Pokličimo v kočo u; če nam povedo, da jih je planinec že obiskal, to pomeni, da se je planinec ponesrečil nekje na poti od u do t, torej lahko postavimo s na u in nadaljujemo z enakim postopkom. Podobno pa, če nam povedo, da jih planinec še ni obiskal, to pomeni, da se je ponesrečil na poti od s do u, torej lahko postavimo t na u in nadaljujemo z enakim postopkom. Dolžina poti, ki jo še pregledujemo, se tako po vsakem klicu približno razpolovi (pot od s do u je dolga $\lfloor d/2 \rfloor$, pot od u do t pa $\lceil d/2 \rceil$). Sčasoma nam ostane le še pot dolžine 1, to je ena sama povezava, in takrat vemo, da se je planinec ponesrečil na njej.

Naj bo $f(d)$ največje možno potrebno število klicev pri tem postopku, če začnemo s potjo dolžine d. Iz razmisleka v prejšnjem odstavku vidimo, da je $f(1) = 0$ in $f(d) = 1 + \max\{f(\lfloor d/2 \rfloor), f(\lceil d/2 \rceil)\}$. Z indukcijo po d se lahko prepričamo, da za vse d velja $f(d) = \lceil \log_2 d \rceil$,[11] to pa pomeni, da naš postopek res doseže najmanjše možno število klicev v planinske koče.

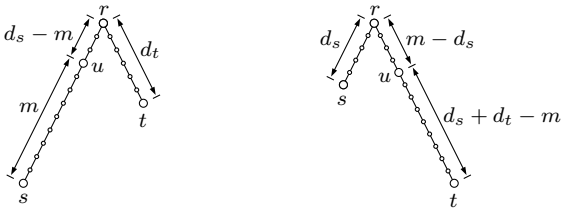
Zapišimo zdaj podprogram, ki za dana s in t izračuna minimalno potrebno število klicev, po katerem sprašuje podnaloga (a):

```
int Drevo::MinStKlicev(int s, int t) const
{
  int r = NajgloljiSkupniPrednik(s, t);
  int dolzPoti = (globina[s] - globina[r]) + (globina[t] - globina[r]);
  if (dolzPoti <= 1) return 0;
  int stKlicev = 0; while (dolzPoti > (1 << stKlicev)) stKlicev++;
  return stKlicev;
}
```

[11] Res: za $d = 1$ je $\lceil \log_2 d \rceil = 0$, kar je res enako $f(1)$. Recimo zdaj, da trditev velja do $d - 1$; prepričajmo se, da velja tudi za d. (1) Pri sodem d, recimo $d = 2c$, imamo $\lfloor d/2 \rfloor = \lceil d/2 \rceil = c$, zato je $f(d) = 1 + f(c)$, kar je po induktivni predpostavki naprej enako $1 + \lceil \log_2 c \rceil = \lceil \log_2 2c \rceil = \lceil \log_2 d \rceil$. (2) Pri lihem d, recimo $d = 2c - 1$, pa imamo $\lfloor d/2 \rfloor = c - 1$ in $\lceil d/2 \rceil = c$, zato je $f(d) = 1 + \max\{f(c-1), f(c)\}$, kar je po induktivni predpostavki enako $1 + f(c) = 1 + \lceil \log_2 c \rceil = \lceil \log_2 2c \rceil = \lceil \log_2 (d+1) \rceil$; to pa bi bilo lahko večje od $\lceil \log_2 d \rceil$ le, če bi bil d potenca števila 2, kar pa ni, saj je vendar lih; torej je res $f(d) = \lceil \log_2 d \rceil$. □

Časovna zahtevnost te rešitve je $O(n \log n)$ za predpripravo podatkov o prednikih, nato pa $O(\log n)$ za vsako poizvedbo (torej za vsak klic funkcije MinStKlicev).

Oglejmo si zdaj še podprogram, ki planinca tudi zares poišče, kot zahteva podnaloga (b). Postopek z bisekcijo smo opisali že zgoraj, razmisliti moramo le o tem, kako poiskati točko u na pol poti med s in t. Spomnimo se, da imata s in t skupnega prednika r; pot od s do t lahko torej razdelimo na levi krak, dolg $d_s := g_s - g_r$ korakov, in desni krak, dolg $d_t := g_t - g_r$ korakov. Skupaj je pot dolga $d := d_s + d_t$ korakov, točka u pa naj leži $m := \lfloor d/2 \rfloor$ korakov naprej od s na tej poti. Ločimo torej lahko dva primera: (1) če je $m < d_s$, leži točka u na levem kraku — dobimo jo torej lahko tako, da poiščemo s-jevega prednika m nivojev nad njim; (2) sicer pa leži točka u na desnem kraku in jo lahko dobimo tako, da poiščemo t-jevega prednika $d - m$ nivojev nad njim. Oba primera kaže spodnja slika:



(1) $m < d_s$ (2) $m \geq d_s$

Ko pri bisekciji premikamo s in t, moramo paziti še na to, da primerno popravimo tudi r: če u leži na desnem kraku in tja prestavimo s, to pomeni, da bo ta novi s prednik t-ja, zato moramo tudi r postaviti na s. Podobno se zgodi, če u leži na levem kraku in tja prestavimo t.

```
pair<int, int> Drevo::KjeJePlaninec(int s, int t) const
{
  if (s == t) return {s, s};
  int r = NajgloblјiSkupniPrednik(s, t), ds, dt;
  while ((ds = globina[s] − globina[r]) + (dt = globina[t] − globina[r]) > 1)
  {
    // Planinec je nekje na poti od s do t, pri čemer je s že obiskal, t pa še ne.
    // Najglobljі skupni prednik s in t je r, ki leži ds nivojev nad s in dt nivojev nad t.
    int m = (ds + dt) / 2; bool levo = (m < ds);
    int u = levo ? Prednik(s, m) : Prednik(t, ds + dt − m);
    // Točka u je na pol poti med s in t. Ali jo je planinec obiskal?
    if (JeObiskal(u)) { // Če je že obiskal u, premaknimo s v u.
      s = u; if (! levo) r = s; }
    else { // Če u-ja še ni obiskal, premaknimo t v u.
      t = u; if (levo) r = t; }
  }
  return {s, t};
}
```

Ta funkcija izvede le minimalno število klicev v planinske koče, to je $\lceil \log_2 d \rceil$ klicev, če je d dolžina poti od s do t. Drugače pa je njena časovna zahtevnost $O((\log n)^2)$, kajti med drugim $O(\log n)$-krat pokliče funkcijo Prednik, vsak od teh klicev pa, kot vemo, porabi $O(\log n)$ časa.

To rešitev lahko še malo izboljšamo, da bo še vedno poklicala le v največ $\lceil \log_2 d \rceil$ koč, njena časovna zahtevnost pa bo le še $O(\log n)$. Spomnimo se, da je veliko lažje narediti skok za 2^k nivojev navzgor po drevesu (kar lahko naredimo v $O(1)$ časa s pogledom v tabelo predniki) kot pa za neko poljubno število nivojev, ki ni potenca števila 2 (takrat potrebujemo $O(\log n)$ časa za klic funkcije Prednik). Na srečo pri bisekciji ni nujno, da za u izberemo kočo točno na sredini poti od s do t; dovolj je že, če je približno na sredini — recimo na srednji tretjini poti. Vzemimo torej $m = 2^k$, eksponent k pa izberimo tako, da bo m ležal na območju $d/3 \le m < 2d/3$. Iz te neenačbe vidimo, da moramo vzeti $k = \lceil \log_2(d/3) \rceil$.

Zdaj lahko razmišljamo takole: če je levi krak dolg vsaj 2^k korakov, torej če je $d_s \ge 2^k$, lahko primerno točko u izberemo tako, da se iz s premaknemo za 2^k nivojev navzgor; podobno razmišljamo, če je desni krak dolg vsaj 2^k korakov. Če pa sta oba kraka krajša od 2^k, vzemimo za u kar točko r, torej najglobljega skupnega prednika točk s in t. V vsakem primeru lahko točko u izberemo v $O(1)$ časa namesto v $O(\log n)$. Tako dobimo naslednji postopek:

```
pair<int, int> Drevo::KjeJePlaninec2(int s, int t) const
{
  if (s == t) return {s, s};
  int r = NajgloblјiSkupniPrednik(s, t);
  int ds = globina[s] − globina[r], dt = globina[t] − globina[r];
  int k = 0; while (ds + dt > (1 << (k + 1))) k++;
  while (ds + dt > 1)
  {
    // Planinec je nekje na poti od s do t. Njun najglobljі skupni prednik
    // je točka r, ki leži ds nivojev nad s in dt nivojev nad t.
    // Poiščimo potenco števila 2 na območju [d/3, 2d/3) za d = ds + dt; recimo ji 2^k.
    while (3 << (k − 1) >= ds + dt) k−−;
    int m = 1 << k; // zdaj je ds + dt <= 3m < 2(ds + dt)

    // Če je levi krak dolg vsaj m, izberimo u na njem.
    if (ds >= m) { if (int u = predniki[k][s]; JeObiskal(u)) s = u; else t = u, r = u; }

    // Podobno tudi za desni krak.
    else if (dt >= m) { if (int u = predniki[k][t]; JeObiskal(u)) s = u, r = u; else t = u; }

    // Sicer vzemimo za u kar skupnega prednika, r.
    else { if (JeObiskal(r)) s = r; else t = r; }

    // Eno od krajišč smo premaknili v u; izračunajmo novo dolžino krakov.
    ds = globina[s] − globina[r]; dt = globina[t] − globina[r];
  }
  return {s, t};
}
```

Naj bo $f(d)$ število koč, v katere pokliče ta postopek, če je pot od s do t dolga d korakov. Prepričali bi se radi, da je $f(d) = \lceil \log_2 d \rceil$, torej da tudi ta rešitev res ne preseže minimalnega potrebnega števila klicev. Pri $d = 1$ vidimo, da funkcija ne pokliče niti v eno kočo, torej je $f(d) = 0$, kot si tudi želimo. Za večje d bomo dokazovali z indukcijo po d; recimo torej, da naša trditev že velja vse do $d - 1$, in poglejmo, kaj se zgodi pri d.

Spomnimo se, da naša funkcija takrat vzame $k = \lceil \log_2(d/3) \rceil$ in $m = 2^k$, pri čemer slednji zagotovo leži na območju $d/3 \le 2^k < 2d/3$. Funkcija si izbere neko točko u na poti od s do t in nato pomakne eno od krajišč (s ali t) v točko u. Pot

se s tem skrajša z d na neko krajšo dolžino, recimo d'. Ker smo pri tem poklicali v kočo u, dobimo rekurzivno zvezo $f(d) = 1 + f(d')$. Toda kakšen je d'? (1) Če je bil eden od krakov dolg vsaj m, smo u postavili m nivojev nad spodnji konec tistega kraka, tako da nam je u razdelil dosedanjo pot dolžine d na dva kosa, dolga po m in $d - m$. Takrat je torej d' enak bodisi m bodisi $d - m$. (2) Sicer, torej če sta bila oba kraka krajša od m, pa smo u postavili kar v skupnega prednika. Takrat je torej d' enak dolžini enega od krakov (d_s ali d_t), vsekakor pa je manjši od m.

Tako vidimo, da je največja možna vrednost za $f(d)$ enaka $1 + \max\{f(m), f(d - m), f(d') : d' < m\}$. Ker po induktivni predpostavki vemo, da do vključno $d - 1$ zagotovo velja $f(x) = \lceil \log_2 x \rceil$, in ker je ta funkcija naraščajoča (ne nujno strogo) in ker d' in m ležita na tem območju, to pomeni, da iz $d' < m$ sledi $f(d') \leq f(m)$, zato $f(d')$ nič ne vpliva na naš max in ga lahko poenostavimo: $f(d) = 1 + \max\{f(m), f(d - m)\}$. Po induktivni predpostavki tudi vemo, da je $f(m) = \lceil \log_2 m \rceil = k$, torej lahko zapišemo celo $f(d) = 1 + \max\{k, f(d - m)\}$. Radi pa bi dokazali, da je to enako $\lceil \log_2 d \rceil$.

Videli smo, da $m = 2^k$ leži na območju $d/3 \leq m < 2d/3$. To pa tudi pomeni, da d leži na območju $(3/2)m < d \leq 3m$. Ločimo zdaj dve možnosti.

(1) Lahko da d leži na $(3/2)m < d \leq 2m$. Na tem območju je edina potenca števila 2 kar število $2m$, zato je tu $\lceil \log_2 d \rceil = \log_2(2m) = k + 1$. Iz dejstva, da je $d \leq 2m$, sledi tudi, da je $d - m \leq m$, torej je po induktivni predpostavki $f(d - m) \leq f(m) = k$. Zato je $f(d) = 1 + \max\{k, f(d - m)\} = 1 + k = \lceil \log_2 d \rceil$.

(2) Lahko pa d leži na $2m < d \leq 3m$. Na tem območju ni nobene potence števila 2, naslednja je šele $4m$, zato je tu $\lceil \log_2 d \rceil = \log_2(4m) = k + 2$. Iz $2m < d \leq 3m$ sledi tudi, da je $m < d - m \leq 2m$, torej je po induktivni predpostavki $f(d - m) = \lceil \log_2(2m) \rceil = k + 1$. Potem je $f(d) = 1 + \max\{k, f(d - m)\} = 1 + (k + 1) = k + 2 = \lceil \log_2 d \rceil$.

Tako torej vidimo, da če velja $f(x) = \lceil \log_2 x \rceil$ za vse $x < d$, velja tudi za $x = d$, torej velja sploh za vse x, prav to pa smo tudi želeli dokazati. □

12. Kontrolne vsote

Za začetek vpeljimo nekaj oznak, ki nam bodo prišle prav pri opisu rešitve te naloge. Z $V(n)$ označimo vsoto števk v desetiškem zapisu n-ja, s $K_1(n)$ oz. $K_2(n)$ pa enomestno oz. dvomestno kontrolno vsoto n-ja. Množico $\{a, a + 1, \ldots, b - 1, b\}$ bomo krajše zapisali kot $\langle a, b \rangle$. Zapis $d^{\underline{k}}$ bo predstavljal niz k pojavitev števke d; na primer: $4^{\underline{3}} = 444$.

(a) Najprej si pripravimo pomožni podprogram, ki izračuna vsoto števk v danem številu. Če je n podan kot celoštevilska spremenljivka, lahko njegovo najbolj desno števko (enice) dobimo tako, da izračunamo njegov ostanek pri deljenju z 10; celi del količnika pri tem deljenju pa je ravno tisto število, ki nastane, če n-ju to najbolj desno števko pobrišemo. Tako lahko v zanki režemo n-jeve števke od desne proti levi in jih seštevamo:

```
int VsotaStevk(long long n)
{
  int vsota = 0;
  while (n > 0) { vsota += n % 10; n /= 10; }
  return vsota;
```

```
}
```

Če je n podan kot niz, moramo iti v zanki po znakih tega niza, izračunati za vsak znak vrednost števke, ki jo predstavlja, in te vrednosti seštevati.

```
int VsotaStevk(const char *s)
{
  int vsota = 0;
  while (*s) vsota += *s++ - '0';
  return vsota;
}
```

Glavni podprogram za izračun kontrolne vsote mora zdaj v zanki računati vsoto števk, pred vsakim izračunom pa še preveriti, če je že dovolj majhna (eno- ali dvomestna, karkoli je uporabnik pač zahteval), da se lahko ustavi:

```
int KontrolnaVsota(long long n, bool dvomestna)
{
  while (n >= (dvomestna ? 100 : 10)) n = VsotaStevk(n);
  return (int) n;
}
```

Če je n podan kot niz, lahko najprej pogledamo, če je dovolj kratek (1 ali 2 znaka); če je, moramo le vrniti njegovo številsko vrednost, sicer pa lahko izračunamo vsoto njegovih števk in potem izračunamo *njeno* kontrolno vsoto:

```
#include <cstdlib>
#include <string>
using namespace std;

int KontrolnaVsota(const char *s, bool dvomestna)
{
  if (strlen(s) <= (dvomestna ? 2 : 1)) return atoi(s);
  else return KontrolnaVsota(VsotaStevk(s), dvomestna);
}
```

(b) Če s podprogramom iz rešitve podnaloge (a) izračunamo enomestne kontrolne vsote za nekaj zaporednih n, lahko opazimo, da se te kontrolne vsote ciklično ponavljajo: če ima n kontrolno vsoto c, ima $n+1$ kontrolno vsoto $c+1$, razen pri $c = 9$, ko ima $n+1$ kontrolno vsoto 1.

Prepričajmo se, da to res velja na splošno. Lažje kot n in $n+1$ je primerjati n in $n+9$. Kaj se spremeni v desetiškem zapisu n-ja, ko mu prištejemo 9? Ločimo dve možnosti. (1) Če se n konča na števko 0, se ta zdaj spremeni v 9, višje ležeče števke pa se ne spremenijo. Vsota števk se tako poveča za 9. (2) Sicer se n konča na neko neničelno števko, recimo c. Levo od te števke je mogoče nekaj devetk, prej ali slej pa gotovo nastopi neka števka, manjša od 9; recimo ji d. (Če ni drugega, si lahko za d mislimo vodilno ničlo). Torej imamo n oblike

$$n = \ldots d\, 99 \ldots 9\, c$$

(pri čemer je skupina devetk lahko tudi prazna). Če zdaj poskusimo temu n-ju ročno prišteti 9, vidimo, da pri enicah nastane $c+9$, kar je ≥ 10, vendar < 20; zato nesemo 1 naprej, pri enicah pa obdržimo le ostanek po deljenju $(c+9)$ z 10, to pa je

$(c+9)-10 = c-1$. Prenos naprej nam potem spremeni vse devetke v ničle, števko d spremeni v $d+1$, tu pa se prenašanje ustavi, saj je $d < 9$. Tako smo dobili

$$n + 9 = \ldots (d+1)\, 00 \ldots 0\, (c-1).$$

Če primerjamo n in $n+9$, vidimo, da se je števka c zmanjšala za 1, d povečala za 1, nekaj devetk pa se je spremenilo v ničle. Učinek na vsoto števk je torej ta, da se je zmanjšala za nek večkratnik števila 9.

V gornjem odstavku smo torej ugotovili, da se $v(n)$ in $v(n+9)$ razlikujeta za nek večkratnik števila 9. Iz tega pa tudi sledi, da če se n in m razlikujeta za nek večkratnik 9, se potem tudi $v(n)$ in $v(m)$ razlikujeta za nek večkratnik 9. Iz *tega* pa zdaj sledi, da se tudi $v(v(n))$ in $v(v(m))$ razlikujeta za nek večkratnik 9, ravno tako tudi $v(v(v(n)))$ in $v(v(v(m)))$ in tako naprej. Tako sčasoma pridemo do tega, da se tudi kontrolni vsoti $K_1(n)$ in $K_1(m)$ razlikujeta za nek večkratnik 9. Toda ker sta tidve kontrolni vsoti enomestni, se lahko razlikujeta za večkratnik 9 le tako, da sta enaki. Tako smo torej pokazali, da če se n in m razlikujeta za večkratnik 9, imata enako kontrolno vsoto, torej se kontrolne vsote res pojavljajo ciklično, kot smo opazili na začetku.

Če upoštevamo še, da so za n od 1 do 9 kontrolne vsote kar enake n-ju samemu, lahko naše ugotovitve povzamemo s formulo $K_1(n) = ((n-1) \bmod 9) + 1$.

Naloga zahteva, da za dani k poiščemo neko k-mestno število n s kontrolno vsoto $K_1(n) = c$, pri čemer pa n ne sme vsebovati ničel. Postavimo na začetku vse n-jeve števke na 1. Vsota števk bo tedaj $v(n) = k$, iz te pa lahko izračunamo n-jevo kontrolno vsoto po formuli iz prejšnjega odstavka, torej $K_1(n) = K_1(v(n)) = K_1(k)$. Če ta kontrolna vsota ni enaka c, moramo pogledati, koliko ji še manjka do c, in za toliko povečati eno od n-jevih števk (recimo, da kar najbolj levo); s tem se bo za toliko povečala tudi $v(n)$, zato pa bo tudi kontrolna vsota dosegla c. Paziti moramo še na možnost, da je $K_1(k)$ večja od c; ker n-jevih števk ne moremo zmanjševati (ker manjše od 1 ne smejo biti), bomo v tem primeru eno od njih povečali za $9 + (c - K_1(k))$.

```
long long SestaviStevilo(int k, int c)
{
  // Kakšno kontrolno vsoto bi imelo število iz samih enic?
  int kv = (k - 1) % 9 + 1;
  // Prva števka bo za toliko večja od 1, kolikor kv-ju manjka do c.
  long long n = 1 + (9 + c - kv) % 9;
  // Dodajmo še ostale števke, same enice.
  for (int i = 1; i < k; i++) n = (10 * n) + 1;
  return n;
}
```

Za večje k je bolje, če število sestavimo kot niz namesto kot celoštevilsko spremenljivko:

```
string SestaviSteviloKotNiz(int k, int c)
{
  string n(k, '1'); // Sestavimo n iz samih enic.
  int kv = (k - 1) % 9 + 1; // To je njegova kontrolna vsota.
  n[0] += (9 + c - kv) % 9; // Prištejmo, kolikor ji manjka do c.
  return n;
}
```

(c) Kot smo videli pri (b), imajo v številu, ki ga sestavi naša rešitev, vse števke vrednost 1, razen prve (najbolj leve) števke, ki je večja za toliko, kolikor je treba, da bomo dobili želeno kontrolno vsoto. Tega ni težko zapisati kot podprogram:

```
int SestaviSteviloPoStevkah(long long k, int c, long long i)
{
  int kv = (k − 1) % 9 + 1; // Kontrolna vsota števila iz k enic.
  int d = 1; // Vse števke bomo postavili na 1,
  // prvi pa nato še prištejemo razliko med c in kv.
  if (i == 1) d += (9 + c − kv) % 9;
  return d;
}
```

(d) Najmanjše število z vsoto števk s označimo z μ_s. Če hočemo, da bo to število čim manjše, mora za začetek imeti čim manj števk (če primerjamo dve števili in ima prvo več števk od drugega, je prvo gotovo tudi večje od drugega). Vsaka števka prispeva k vsoti največ 9, torej potrebujemo $\lceil s/9 \rceil$ števk. Če je s večkratnik 9, lahko vsoto s dosežemo le tako, da vse te števke postavimo na 9 in problem je rešen.

Če pa s ni večkratnik 9, bi nam zaporedje $\lceil s/9 \rceil$ devetk dalo preveliko vsoto, in sicer za $9 - (s \bmod 9)$ preveliko. Eno ali več števk v našem številu moramo torej zmanjšati. Ker hočemo, da bo število na koncu čim manjše, se splača zmanjšati le najbolj levo števko, saj ima ta največjo težo: že če zmanjšamo to števko samo za 1, je učinek večji, kot če bi vse števke desno od nje zmanjšali čisto na 0. (Primer: če v 9999 zmanjšamo prvo števko za 1, dobimo 8999, če pa pustimo prvo števko pri miru in zmanjšamo vse ostale na 0, dobimo 9000, kar je večje od 8999.)

Tako dobljena števila μ_s (za $s = 1, 2, \ldots$) so torej po vrsti naslednja: $1, 2, \ldots, 9$, $19, 29, \ldots, 99$, $199, 299, \ldots, 999$, $1999, 2999, \ldots, 9999$ in tako naprej. Lahko bi jih zapisali celo s formulo (od česar sicer ne bo velike koristi). Spomnimo se, da ima število iz t devetk vrednost $10^t - 1$ in da ima najbolj leva števka v njem utež 10^{t-1}; tako dobimo

$$\begin{aligned}\mu_s &= 10^{\lceil s/9 \rceil} - 1 - (9 - (s \bmod 9)) \cdot 10^{\lceil s/9 \rceil - 1} \\ &= (1 + (s \bmod 9)) \cdot 10^{\lceil s/9 \rceil - 1} - 1.\end{aligned}$$

Oglejmo si še primer izračuna vrednosti μ_s v jeziku C++:

```
long long NajmanjseSteviloZVsotoStevk(int s)
{
  long long n = s % 9; s −= n; // Prva števka je lahko manjša od 9.
  while (s > 0) n = 10 * n + 9, s −= 9; // Dodajmo še dovolj devetk.
  return n;
}
```

(e) Naloga pravi, da ima b v desetiškem zapisu t števk in da je $a \le b$; če ima a manj kot t števk, mu dodajmo na začetek še nekaj vodilnih ničel, tako da bo imel potem a tudi t števk. Zapišimo a po števkah kot $a = a_{t-1} \ldots a_1 a_0$ in podobno za b.

Oglejmo si najprej poseben primer: recimo, da se a in b v zgornjih nekaj števkah ujemata, nato se v eni ne ujemata, od tam navzdol pa ima a same ničle, b pa same devetke. Z drugimi besedami imamo torej $a = c\, a_u\, 0^{\underline{u}}$ in $b = c\, b_u 9^{\underline{u}}$ za $c = a_{t-1} \ldots a_{u+1} = b_{t-1} \ldots b_{u+1}$. Možne vsote števk gredo v tem primeru od $V(a) = V(c) + a_u$ do $V(b) = V(c) + b_u + 9u$ za $V(c) = \sum_{i=u+1}^{t-1} a_i$. Če počasi povečujemo vrednosti spodnjih u števk (eno po eno) od 0 do 9 in vrednost števke u od a_u do b_u,

lahko dobimo tudi vse vmesne vsote. Primer: pri $a = 1400$, $b = 1599$ lahko dobimo vse vsote od 5 do 24.

V splošnem pa lahko interval $\langle a, b\rangle$ razbijemo na nekaj disjunktnih podintervalov take oblike, o kakršni smo govorili v prejšnjem odstavku. Na primer: $\langle 1367, 1631\rangle = \langle 1367, 1369\rangle \cup \langle 1370, 1399\rangle \cup \langle 1400, 1599\rangle \cup \langle 1600, 1629\rangle \cup \langle 1630, 1631\rangle$. Za vsak tak interval lahko določimo pripadajoči interval vsot po postopku iz prejšnjega odstavka, nato pa moramo le še izračunati njihovo unijo. Kot vidimo že iz tega primera, lahko prvo polovico intervalov sestavimo tako, da a najprej počasi povečujemo do naslednjega večkratnika števil 10, 100, 1000 in tako naprej, drugo polovico intervalov pa tako, da b počasi zmanjšujemo do predhodnega večkratnika števil 10, 100, 1000 in tako naprej.

Zapišimo psevdokodo takega postopka:

funkcija VsoteŠtevk(a, b):
Vhod: $a = (a_{t-1}a_{t-2}\ldots a_1a_0)_{10}$ in $b = (b_{t-1}b_{t-2}\ldots b_1b_0)_{10}$, pri čemer je $a < b$.
Izhod: $\{V(n) \,:\, a \le n \le b\}$.

(* *Odrežimo skupni začetek (prefiks) a-ja in b-ja.* *)
$w := 0$; (* *w bo hranil vsoto odrezanih števk.* *)
while $t > 0$ **do**
  if $a_{t-1} = b_{t-1}$ **then** $t := t - 1$; $w := w + a_t$;
  else break;
if $t = 0$ **return** $\{w\}$; (* *To je pri a = b.* *)

(* *Zdaj je* $a_{t-1} < b_{t-1}$. *Obdelajmo preostanek števil a in b.* *)
$M :=$ prazna množica; $v := 0$; **for** $i := 1$ **to** $t - 1$ **do** $v := v + a_i$;
$u := 0$;
while $u < t - 1$:
  (* *Zdaj je a oblike* $a = c\,a_u\,0^{\underline{u}}$ *in* $v = V(c)$. *Obdelajmo interval* $\langle a, c\,9^{\underline{u+1}}\rangle$. *)
  $M := M \cup \langle w + v + a_u, w + v + 9 \cdot (u + 1)\rangle$;
  (* *Povečajmo a na* $c\,9^{\underline{u+1}} + 1$, *kar je enako* $(c + 1)\,0^{\underline{u+1}}$. *)
  while true:
    $a_u := 0$; $u := u + 1$; $v := v - a_u$; $a_u := a_u + 1$;
    if $a_u < 10$ **then break**;

(* *Zdaj je a oblike* $a_{t-1}0^{\underline{t-1}}$. *Števka* a_{t-1} *je za* 1 *večja kot pri prvotnem a. Velja tudi* $a_{t-1} \le b_{t-1}$. *)
$v := 0$;
for $u := t - 1$ **downto** 0:
  (* *Zdaj je* $a = c\,a_u\,0^{\underline{u}}$ *in* $b = c\,b_u\ldots b_0$ *in* $v = V(c)$ *in* $a_u \le b_u$.
    Obdelajmo interval $\langle a, c\,(b_u - 1)\,9^{\underline{u}}\rangle$.
    Izjema je pri $u = 0$, *kjer moramo za zgornje krajišče intervala vzeti* $c\,b_u\,9^{\underline{u}}$, *saj moramo pokriti števila do b, ne le do* $b - 1$. *)
  if $a_u < b_u$:
    if $u = 0$ **then** $d := b_u$ **else** $d := b_u - 1$;
    $M := M \cup \langle w + v + a_u, w + v + d + 9 \cdot u\rangle$;
    $a_u := b_u$; $v := v + a_u$;
return M;

Razmislimo o časovni zahtevnosti tega postopka. Najprej imamo dve vgnezdeni

zanki, od katerih notranja vsakič poveča u za 1, tako da se skupaj izvede le $O(t)$-krat; kasneje imamo še zanko **for**, ki tudi izvede le t iteracij. To je torej $O(t)$ iteracij; koliko časa pa vzame vsaka iteracija? V vsaki iteraciji imamo nekaj preprostih računskih operacij, ki vzamejo vsakič le $O(1)$ časa, in še eno dodajanje intervala v množico M. Trajanje te operacije je odvisno od tega, kako predstavimo množico. Ker sta naši števili a in b dolgi po t števk, so možne vsote števk od 0 do $9t$, torej bo vsebovala M največ $O(t)$ elementov. Tudi intervali, ki jih vsakič dodajamo v M, vsebujejo do $O(t)$ elementov. Če bomo dodajali vsak element posebej (in množico predstavili npr. z razpršeno tabelo ali s tabelo $9t$ bitov), bomo imeli s tem $O(t)$ dela za vsako operacijo oblike $M := M \cup \langle \ldots \rangle$, celoten postopek pa bo imel zato časovno zahtevnost $O(t^2)$. Lažje je, če M predstavimo kot seznam parov, pri čemer vsak par predstavlja krajišči enega od intervalov, ki smo jih dodali v M. Ker se lahko intervali prekrivajo, je koristno potem na koncu postopka krajišča vseh intervalov urediti naraščajoče ter s preletom (*sweep*) po številski premici predelati M v seznam neprekrivajočih se intervalov ali pa celo eksplicitен seznam vseh elementov M-ja. To nam vzame le $O(t)$ časa, saj lahko uporabimo urejanje s štetjem (*counting sort*). Časovna zahtevnost celotnega postopka je tako le $O(t)$, kot je tudi zahtevala naša naloga.

(f) Pri podnalogi (e) smo za vsa števila n z intervala $\langle a, b \rangle$ izračunali vsoto števk $V(n)$ in to natanko enkrat. Pri izračunu $K_2(n)$ pa je včasih treba vsoto števk izračunati po večkrat zaporedoma, $V(V(n))$ ali $V(V(V(n)))$ in podobno, lahko pa tudi nobenkrat: če je $n \leq 99$, je $K_2(n) = n$ in to je lahko različno od $V(n)$. Postopek VsoteŠtevk iz (e) bo torej treba dopolniti oz. zaviti v zanko tako, da bo vsote števk računal primerno mnogokrat.

Funkcijo K_2, ki izračuna dvomestno kontrolno vsoto, lahko lepo zapišemo takole:

$$K_2(n) = \begin{cases} n, & \text{če } n \leq 99 \\ K_2(V(n)) & \text{sicer.} \end{cases}$$

Če nas zanimajo kontrolne vsote števil z intervala $\langle a, b \rangle$, je torej koristno za začetek pogledati, če ta interval deloma ali pa celo ves leži znotraj območja $\langle 1, 99 \rangle$; tista števila so že kar sama svoje dvomestne kontrolne vsote, zato jih lahko brez sprememb prenesemo v izhodno množico, ki jo bomo na koncu vrnili kot rezultat. Za števila s preostanka intervala $\langle a, b \rangle$ pa bo treba vsaj enkrat izračunati vsoto števk, kar lahko naredimo s postopkom VsoteŠtevk, in nato izračunati dvomestne kontrolne vsote tako dobljenih vsot števk.

Spomnimo se, da je postopek VsoteŠtevk vrnil množico M, ki jo je dobil tako, da je vanjo vedno dodajal po cel interval števil naenkrat. Za naše potrebe je koristno, če imamo M predstavljeno kar kot seznam parov, pri čemer vsak par predstavlja krajišči enega takega intervala (če so se prvotni intervali kaj prekrivali, jih prej še zlijmo, da se znebimo prekrivanj).

Zapišimo psevdokodo našega postopka:

funkcija KontrolneVsote(a, b):
(* *Vrne množico* $M := \{K_2(n) : a \leq n \leq b\}$*, predstavljeno kot seznam parov* (ℓ_i, u_i)*, ki povedo, da je* M *unija disjunktnih intervalov* $\langle \ell_i, u_i \rangle$*.* *)
$M :=$ prazen seznam (ki predstavlja prazno množico);

```
dodaj v M interval ⟨a, b⟩;
while M vsebuje kakšno število, večje od 99:
  M′ := prazen seznam;
  za vsak interval ⟨ℓ, r⟩ iz M:
    if r ≤ 99 then dodaj ⟨ℓ, r⟩ v M′; continue;
    if ℓ ≤ 99 then dodaj ⟨ℓ, 99⟩ v M′; ℓ := 100;
    dodaj v M′ vse intervale iz VsoteŠtevk(ℓ, r);
  z urejanjem krajišč vseh intervalov v M′ in preletom številske premice
    predelaj M′ tako, da se intervali v njej ne bodo prekrivali;
  M := M′;
return M;
```

V realističnih okoliščinah si je težko predstavljati, da bi morali vsote števk računati več kot dvakrat, vsekakor pa ne več kot trikrat. Pri podnalogi (d) smo označili z μ_s najmanjše število z vsoto števk s; tukaj ga pišimo raje kot $\mu(s)$. Pri (d) smo videli, da ima $\mu(s)$ v desetiškem zapisu $\lceil s/9 \rceil$ števk, torej je približno reda velikosti $10^{s/9}$.

Pri izračunu $K_2(n)$ potrebujemo drugo računanje vsote števk le tedaj, če je bila vsota n-jevih števk več kot dvomestna, torej $V(n) \geq 100$, kar je možno šele pri $n \geq \mu(100) = 199\,999\,999\,999$ (to je 1 manj kot 200 milijard).

Tretje računanje vsote števk bi bilo potrebno šele, če bi bil tudi $V(V(n))$ večji ali enak 100, to pa bi pomenilo, da mora biti $V(n) \geq \mu(100)$ in zato $n \geq \mu(\mu(100))$. Z drugimi besedami, vsota n-jevih števk bi morala biti okrog 200 milijard, to pa je mogoče le, če ima n vsaj kakšnih $\mu(100)/9 \approx 22{,}2$ milijard števk. Taka števila pa je počasi že težko spraviti v pomnilnik našega računalnika, sploh če jih potrebujemo po več naenkrat.

Četrto računanje vsote števk bi bilo potem potrebno šele pri $n \geq \mu(\mu(\mu(100)))$, dolžina takega n-ja bi bila približno $\mu(\mu(100))/9$ števk, kar je približno $10^{22{,}2\cdot 10^9}$. Za primerjavo: fiziki ocenjujejo, da je v znanem vesolju okrog 10^{80} atomov. Četudi bi torej predelali célo vesolje v pomnilniške medije, bi lahko vanje shranili le neznatno majhen delež tako velikega n-ja. Zaključimo torej lahko, da se bo zunanja zanka našega postopka KontrolneVsote izvedla kvečjemu trikrat.

(g) Za začetek je koristno pognati postopka VsotaŠtevk in KontrolneVsote iz podnalog (e) in (f) za $b = 9a$ in za različne a in si ogledati množice, ki jih vračata. Videli bomo, da so te množice precej pravilne in predvidljive obilke. Naša naloga pa je zdaj ta, da te množice opišemo in pokažemo, da naš opis velja na splošno.

(1) Če je $1 \leq a \leq 11$, so števila od a do $9a$ vsa eno- ali dvomestna (saj je $9a \leq 9 \cdot 11 = 99$), zato so že kar sama svoje dvomestne kontrolne vsote. Takrat je torej $M(a) = \langle a, 9a \rangle$.

(2) Če je $12 \leq a \leq 99$, so števila od a do $9a$ nekatera dvomestna (ker je $a \leq 99$), nekatera pa tromestna (ker je $9a \geq 9 \cdot 12 = 108$ in po drugi strani $9a \leq 9 \cdot 99 = 891$). Dvomestna so že sama svoje kontrolne vsote in prispevajo v $M(a)$ interval $\langle a, 99 \rangle$.

Tromestna števila pa tvorijo interval $\langle 100, 9a \rangle$ in razmisliti moramo o tem, kakšne so njihove vsote števk $V(n)$. Največjo med temi vsotami označimo s s; recimo, da jo dosežemo pri številu n; če zdaj počasi zmanjšujemo števke tega n-ja vse do 0, razen vodilne števke, ki jo zmanjšamo le do 1, lahko dobimo tudi vse vsote od 1 do $s-1$ (najmanjšo vsoto, 1, dobimo pri številu 100). Tromestna števila $\langle 100, 9a \rangle$ torej

prispevajo v M interval $\langle 1, s\rangle$; ugotoviti moramo le še to, kakšen je s v odvisnosti od a.

Ker je $a \geq 12$, je $9a \geq 108$, torej gre n lahko vsaj do 108, ki ima vsoto števk 9, tako da je $s \geq 9$. Po drugi strani, ker je $a \leq 99$, je $9a \leq 891$, torej gre lahko n največ do 891; največjo vsoto števk med števili do 891 pa ima $n = 889$ z vsoto števk 25. Radi bi torej za vsak s od 9 do 25 pogledali, katero je najmanjše tromestno število, ki ima vsoto števk s. Spomnimo se, da je najmanjše število z vsoto števk s enako μ_s, težava je le, da μ_s ni nujno tromestno. Če je μ_s eno- ali dvomesten, bomo najmanjše primerno tromestno število (recimo mu μ'_s) dobili tako, da bomo njegovo vodilno števko zmanjšali za 1 in na mesto stotic vrinili števko 1. Ko enkrat poznamo μ'_s, pa ni težko izračunati, pri katerem a ga dosežemo: ker gre n do $9a$, bomo $n = \mu'_s$ dosegli pri $\mu'_s \leq 9a$, torej od $a = \lceil \mu'_s/9 \rceil$ naprej.

s	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
μ_s	9	19	29	39	49	59	69	79	89	99	199	299	399	499	599	699	799
μ'_s	108	109	119	129	139	149	159	169	179	189	199	299	399	499	599	699	799
$a = \lceil \mu'_s/9 \rceil$	12	13	14	15	16	17	18	19	20	21	23	34	45	56	67	78	89

Primer: če nas zanima $a = 30$, lahko v gornji tabeli vidimo, da lahko vsoto $s = 19$ dosežemo od $a = 23$ naprej, vsoto $s = 20$ pa šele od $a = 34$ naprej. Pri $a = 30$ je torej največja možna vsota $s = 19$, zato tromestna števila v tem primeru prispevajo v M interval $\langle 1, 19\rangle$.

Zaključimo lahko torej, da je $M(a) = \langle 1, s\rangle \cup \langle a, 99\rangle$, pri čemer s dobimo po zgoraj opisanem postopku. Tadva intervala, ki tvorita $M(a)$, sta vedno disjunktna, ker je a za več kot 1 večji od s; o tem se prepričamo takole: pri $a = 12$ dobimo $s = 9$, torej trditev drži; odtlej pa, če se a poveča za 1, se s poveča za 0 ali za 1, torej se razlika $a - s$ nikoli ne zmanjša (ampak se počasi celo povečuje).

(3) Če je $100 \leq a < \lceil \mu_{100}/9 \rceil$ (torej $a \leq 2\underline{11}$), so števila iz $\langle a, 9a\rangle$ vsa vsaj tromestna, torej sama po sebi niso primerna za kontrolne vsote; vendar pa so njihove vsote števk manjše od 100, torej so primerne za kontrolne vsote. Tu je torej $M(a) = \{V(n) : a \leq n \leq 9a\}$. Recimo, da ima a v desetiškem zapisu t števk. Iz tega potem sledi, da so števila na $\langle a, 9a\rangle$ bodisi vsa t-mestna (do tega pride, če je $9a < 10^t$, torej $a \leq 1\underline{t}$) ali pa so nekatera t-mestna in nekatera $(t+1)$-mestna (to je pri $a > 1\underline{t}$).

(3.1) Kdaj $M(a)$ vsebuje število 1? To se zgodi, če obstaja na $\langle a, 9a\rangle$ nek n z vsoto števk $V(n) = 1$; taki n-ji pa so le potence števila 10. Videli smo, da so števila $\langle a, 9a\rangle$ vsa ali t-mestna ali (mogoče) $(t+1)$-mestna; možni potenci števila 10 sta torej le 10^{t-1} in 10^t. Prvo dosežemo le pri $a = 10^{t-1}$, saj je to najmanjši možni t-mestni a, vsi drugi a-ji pa so že preveliki. Drugo pa dosežemo, če je $9a \geq 10^t$, torej $a > 1\underline{t}$. Tako torej lahko zaključimo, da pri t-mestnem številu a množica $M(a)$ vsebuje element 1 natanko tedaj, ko a ni z območja $10^{t-1} < a \leq 1\underline{t}$.

(3.2) Katera števila, večja od 1, vsebuje $M(a)$? Poiščimo tisti s, pri katerem je $\mu_s \leq 9a < \mu_{s+1}$; takrat torej vemo, da lahko za vsako vsoto $u \in \langle 2, s\rangle$ najdemo nek tak $n \in \langle 1, \mu_s\rangle \subseteq \langle 1, 9a\rangle$, ki ima $V(n) = u$. Toda mi v definiciji množice $M(a)$ ne gledamo celega območja $\langle 1, 9a\rangle$, pač pa le $\langle a, 9a\rangle$. Ali zaradi te razlike kakšna vsota postane nedosegljiva? Prepričajmo se, da do te težave ne more priti.

Vzemimo torej nek $n < a$ z vsoto števk $u = V(n)$ (za $u > 1$ — primere, ko je vsota števk enaka 1, smo obravnavali že zgoraj v točki 3.1). Radi bi pokazali, da obstaja neko število z vsoto števk u tudi na območju $\langle a, 9a\rangle$.

Spomnimo se, da je a neko t-mestno število, torej je $T \le a < 10T$ za $T = 10^{t-1}$. Če je $a < 2T$, pokriva $\langle a, 9a\rangle$ med drugim tudi celotno območje $\langle 2T, 9T\rangle$; če pa je $a \ge 2T$, pokriva $\langle a, 9a\rangle$ med drugim tudi celotno območje $\langle 10T, 18T\rangle$. Če torej uspemo najti nek tak $n' \in \langle 2T, 9T\rangle$ in nek tak $n'' \in \langle 10T, 18T\rangle$, ki bosta tudi imela vsoto števk u, bomo s tem pokazali, kar hočemo (torej da je u gotovo vsota števk nekega števila iz $\langle a, 9a\rangle$).

Zapišimo n po številkah kot $n = (n_{t-1}n_{t-2}\ldots n_1n_0)_{10}$; če je bil prvotno krajši od t števk, vrinimo na začetek ustrezno število vodilnih ničel (spomnimo se še, da je $t \ge 3$, saj gledamo $a \ge 100$). Opazimo še to, da če v n-ju kakorkoli premešamo vrstni red števk, se njihova vsota seveda nič ne spremeni.

Gotovo je vsaj ena n-jeva števka neničelna (sicer bi bila vsota $u = 0$, ne $u > 1$); in če je samo ena neničelna, je ta števka gotovo večja od 1 (sicer bi bila $u = 1$), v tem primeru pa lahko to števko zmanjšamo za 1 in eno od ničel povečamo za 1 (vsota števk pa ostane enaka). V nadaljevanju smemo torej predpostaviti, da ima n vsaj dve neničelni števki, recimo c in d.

Gotovo tudi niso vse n-jeve števke enake 9, saj bi bil tedaj n največje t-mestno število, kar je v protislovju z dejstvom, da je $a > n$ in da je a tudi t-mestno število. To pomeni, da če so vse neničelne števke enake 9, mora obstajati še vsaj ena ničelna števka, torej lahko eno od devetk zmanjšamo na 8, eno od ničel pa povečamo na 1 in vsota ostane enaka. V nadaljevanju smemo torej predpostaviti, da moremo c in d izbrati tako, da nista obe enaki 9. Brez izgube za splošnost recimo, da je $c \le d$; c je torej z območja $\langle 1, 8\rangle$. Ostale števke staknimo (v poljubnem vrstnem redu) v niz y (dolžine $t - 2$).

Sestavimo zdaj $n' = cdy$ (če je $c > 1$) oz. $n' = 2(d-1)y$ (če je $c = 1$); vidimo lahko, da leži na $\langle 2T, 9T\rangle$ in da ima enako vsoto števk kot u. Potem sestavimo še $n'' = 1(c-1)dy$; vidimo, da leži na $\langle 10T, 18T\rangle$ in da ima enako vsoto števk kot u. Iz tega, kar smo videli zgoraj, sledi, da interval $\langle a, 9a\rangle$ vsebuje vsaj eno od števil n' in n'', torej je mogoče tu doseči tudi vsoto števk u.

Tako lahko torej točko (3.2) zaključimo z ugotovitvijo, da takrat $M(a)$ vsebuje vsa števila od 2 do s (in nobenega večjega od s).

(4) Ostane še primer, ko je $a > 2\underline{11}$. Takrat je $9a \ge \mu_{100}$, zato nam enak razmislek kot v (3.2) pokaže, da zdaj množica vsot števk $\langle V(n) : a \le n \le 9a\}$ vsebuje med drugim vse vsote od 2 do 100. Vsote od 2 do 99 so s tem že tudi dvomestne kontrolne vsote, iz vsote 100 pa bo po še eni iteraciji seštevanja števk nastala kontrolna vsota 1. S tem pa smo dobili že tudi vse možne dvomestne kontrolne vsote, tako da lahko zaključimo, da je v tem primeru $M(a) = \langle 1, 99\rangle$. □

Podnaloga (g) sprašuje še, pri katerih a je $M(a) = \langle 1, 99\rangle$. Pravkar smo videli, da to drži za vse a-je iz točke (4). Pri a-jih iz točk (1) in (2) to ni mogoče, saj smo videli, da tam kakšna kontrolna vsota vedno manjka. Pri a-jih iz točke (3) vidimo, da mora biti $9a \ge \mu_{99} = 9\underline{11}$, torej $a \ge 1\underline{11}$, sicer vsota 99 ne bo dosegljiva; toda za $a = 1\underline{11}$ smo pri (3.1) videli, da takrat ne bo dosegljiva vsota 1; od tam naprej do $a = 2\underline{11}$ te težave ni (vsoto 1 dobimo takrat pri $n = 10^{11}$, ki je tedaj že $\le 9a$), od $a = 2\underline{10}3$ naprej pa pademo tako ali tako že pod točko (4). Tako lahko torej zaključimo, da je $M(a) = \langle 1, 99\rangle$ natanko tedaj, ko je $a \ge 11\,111\,111\,112$.

(h) Iščemo k-mestno število n, v katerem so vse števke neničelne in ki ima $K_2(n) = c$. Primere, ko je $k \le 2$, lahko obravnavamo posebej; eno- in dvomestna števila so

namreč kar sama svoje dvomestne kontrolne vsote, torej je takrat $K_2(n) = n$. Če hočemo kontrolno vsoto c, moramo torej vrniti $n = c$; če pa ta za naše namene ni primeren (ker nima k števk ali pa vsebuje ničlo), moramo javiti napako.

Pri daljših številih ($k \geq 3$) je pri izračunu dvomestne kontrolne vsote treba vsaj enkrat sešteti števke; za tako število torej velja $K_2(n) = K_2(V(n))$. Vsoto n-jevih števk označimo z m, torej $m = V(n)$. Vsaka od n-jevih k števk ima vrednost od 1 do 9 (ker n ne sme imeti ničelnih števk), zato leži njihova vsota m na območju $\langle k, 9k\rangle$. Če uspemo na tem območju najti nek m s kontrolno vsoto c, lahko potem za n vzamemo poljubno k-mestno število z vsoto števk m. Za začetek lahko na primer vse števke n-ja postavimo na $\lfloor m/k \rfloor$; do vsote m nam potem manjka še $m \bmod k$, zato toliko števk povečamo za 1, pa imamo primeren n. Zapišimo ta postopek kot funkcijo v C++ (n bomo vrnili kot niz, če pa primernega n sploh ni, bomo vrnili prazen niz):

```
string SestaviSteviloKotNiz2(int k, int c)
{
  string n;
  // Za eno- in dvomestna števila je število enako svoji dvomestni kontrolni vsoti.
  if (k == 1) { if (1 <= c && c <= 9) n += '0' + c; return n; }
  if (k == 2) {
    if (11 <= c && c <= 99 && c % 10) {
      n += '0' + c / 10; n += '0' + c % 10; }
    return n; }
  // Sicer naj bo m vsota števk n-ja. m leži na ⟨n, 9n⟩ in ima enako
  // dvomestno kotrolno vsoto kot n. Poiščimo kakšen tak m.
  long long m = SteviloSKontrolnoVsoto(k, c);
  if (m == 0) return n; // Takega števila sploh ni.
  // Vrnimo k-mestno število z vsoto števk m.
  n.resize(k, '0' + m / k);
  for (int i = 0; i < m % k; i++) ++n[i];
  return n;
}
```

Podobno je, če hočemo — kot pri podnalogi (c) — izračunati le i-to števko n-ja (če primeren n ne obstaja, bomo vrnili -1):

```
int SestaviSteviloPoStevkah2(long long k, int c, long long i)
{
  // Za eno- in dvomestna števila je število enako svoji dvomestni kontrolni vsoti.
  if (k == 1) return (1 <= c & c <= 9) ? c : -1;
  if (k == 2) return (11<=c && c<=99 && c%10) ? (i==1 ? c/10 : c%10) : -1;
  // Sicer naj bo m vsota števk n-ja. m leži na ⟨n, 9n⟩ in ima enako dvomestno kotrolno
  long long m = SteviloSKontrolnoVsoto(k, c);                    // vsoto kot n.
  if (m == 0) return -1; // Takega števila sploh ni.
  return m / k + (i <= m % k ? 1 : 0);
}
```

V obeh funkcijah smo za izračun m-ja klicali funkcijo SteviloSKontrolnoVsoto, ki mora vrniti poljubno tako število $m \in \langle k, 9k\rangle$, ki ima $K_2(m) = c$. Tu gre torej za prav tisti problem, o katerem smo razmišljali že pri podnalogi (g); ko smo tam ugotavljali, katere kontrolne vsote c so pri danem k sploh mogoče, smo pravzaprav že tudi videli primere m-jev, ki imajo tako kontrolno vsoto. Oglejmo si zdaj implementacijo funkcije, ki sledi našemu razmisleku iz rešitve (g) in izračuna primerno število:

```
// Vrne tak n ∈ ⟨a, 9a⟩, ki ima K_2(n) = s. Če takega ni, vrne 0.
long long SteviloSKontrolnoVsoto(long long a, int s)
{
  if (a <= 0 || s < 0 || s > 99) return 0;
  // Za a od 1 do 11 je primerno število le s sam, če je na intervalu ⟨a, 9a⟩.
  if (a <= 11) return (a <= s && s <= 9 *a) ? s : 0;

  // Za a od 12 do 99 lahko vrnemo s, če je na intervalu ⟨a, 99⟩, sicer pa moramo
  // sestaviti primerno tromestno število iz ⟨100, 9a⟩, če obstaja (torej če s ni prevelik).
  if (a <= 99) {
    if (s >= a) return s;
    long long mi = NajmanjseSteviloZVsotoStevk(s);
    if (mi < 10) mi += 99; else if (mi < 100) mi += 90;
    if (mi > 9 * a) return 0;
    return mi; }

  // Za a od 100 do ⌈μ_100/9⌉ − 1 = 22 222 222 222 lahko kontrolno vsoto 1 dosežemo
  // le tako, da je vsota števk n-ja enaka 1, to pa je mogoče le, če je n potenca števila 10.
  if (s == 1 && a <= 22222222222ull) {
    long long n = 1; while (n < a) n *= 10;
    return (n <= 9 * a) ? n : 0; }

  // Pri večjih a pa se bo kontrolno vsoto 1 zanesljivo dalo doseči
  // tako, da sestavimo n z vsoto števk 100.
  else if (s == 1) s = 100;

  // Poiščimo zdaj primeren n z vsoto števk s, če sploh obstaja.
  long long mi = NajmanjseSteviloZVsotoStevk(s); // μ_s iz podnaloge (d)
  if (mi > 9 * a) return 0;  // vsote s na območju ⟨a, 9a⟩ ni mogoče doseči
  if (mi >= a) return mi;   // μ_s leži na ⟨a, 9a⟩ in je primeren

  // Naj bo t število števk a-ja in naj bo T = 10^(t−1).
  int t = 1; long long T = 1; while (a >= 10 * T) T *= 10, ++t;

  // Število μ_s je oblike c 9^r.
  int c = s % 9, r = s / 9, d; if (c == 0) { c = 9; --r; }

  // Če ima μ_s eno samo neničelno števko, jo razbijmo na 1 in c − 1.
  // Tudi če so neničelne števke same devetke, razbijmo eno na 1 in 8.
  if (r == 0 || c == 9) { d = c − 1; c = 1; }
  else { d = 9; --r; }

  // Zdaj imamo število c d 9^r (pri čemer je c ≤ d in c < 9) z vsoto števk s.
  long long y = 0; for (int i = 0; i < r; i++) y = 10 * y + 9;
  long long n1 = (c>1) ? (c*T + d * (T/10) + y) : (2*T + (d−1) * (T/10) + y);
  long long n2 = (10 + c − 1) * T + d * (T / 10) + y;

  // n1 leži na ⟨2T, 9T⟩, n2 pa na ⟨10T, 18T⟩; oba imata vsoto s.
  // Vsaj eden od njiju torej gotovo leži na ⟨a, 9a⟩; vrnimo tistega.
  return (a < 2 * T) ? n1 : n2;
}
```

13. Marsovska števila

(a) Na prvi pogled se zdi, kot da moramo le preverjati, če so oklepaji pravilno gnezdeni (in še nekaj drugih malenkosti, ki jih tudi ni težko preverjati: da se začetni oklepaj konča šele na koncu niza in da niz ne vsebuje drugih znakov kot oklepajev, zaklepajev in malih črk), vendar se naloga nekoliko zaplete zaradi pravila v besedilu naloge, ki pravi, da je število oblike $(s_1 \ldots s_k)$ dovoljeno le pri $k > 1$. (Namen tega pravila je prepovedati odvečne oklepaje, npr. v nizih `(c)` ali `(c((cc))c)`.) Zato

ni dovolj, da med branjem niza vzdržujemo števec s trenutno globino gnezdenja oklepajev, pač pa moramo šteti tudi, koliko členov s_i smo že prebrali pri vsakem od trenutno odprtih oklepajev; tako bomo lahko, ko se nek oklepaj zapre s pripadajočim zaklepajem, preverili, če sta bila vmes vsaj dva člena (torej $k > 1$). Ker potrebujemo ta podatek za vsak trenutno odprt nivo gnezdenja, jih moramo hraniti v nekakšnem seznamu oz. skladu (saj jih bomo dodajali in brisali le na koncu). V spodnji rešitvi smo si v ta namen pomagali z razredom stack iz C++ove standardne knjižnice.

```
#include <stack>
using namespace std;

bool JeMarsovsko(const char *s)
{
  // Prazen niz ni marsovsko število.
  if (! s || ! *s) return false;

  // Prvi znak je lahko črka, če je to tudi edini znak.
  if ('a' <= *s && *s <= 'z') return ! s[1];

  // Sicer mora biti prvi znak oklepaj. Premaknimo se mimo njega.
  if (*s++ != '(') return false;

  // Preverimo, če so vsi znaki oklepaji, zaklepaji in črke,
  // ter sproti vzdržujmo število členov na vsakem nivoju.
  stack<int> stClenov; stClenov.push(0);
  for ( ; *s; ++s)
    // Pri oklepaju se začne nov nivo gnezdenja.
    if (*s == '(') stClenov.push(0);

    // Pri zaklepaju se trenutni nivo konča.
    // Preverimo, če sta bila v tem številu vsaj dva člena.
    else if (*s == ')') {
      if (stClenov.top() < 2) return false;
      stClenov.pop();

      // Če se je s tem končal najbolj zunanji zaklepaj,
      // se premaknimo mimo njega in končajmo.
      if (stClenov.empty()) { ++s; break; }

      // Pravkar končano število je nov člen v zunanjem številu,
      // v katerem je bilo vgnezdeno.
      else ++stClenov.top(); }

    // Pri črki le povečamo števec členov na trenutnem nivoju.
    else if ('a' <= *s && *s <= 'z') ++stClenov.top();

    // Drugi znaki so neveljavni.
    else return false;

  // Če smo na koncu niza in sklad ni prazen, to pomeni, da nek oklepaj nima
  // pripadajočega zaklepaja. Če pa je sklad prazen, mi pa nismo na koncu niza, to pomeni,
  // da se je začetni oklepaj prehitro zaključil, kot npr. v "(cc)c". Oboje je neveljavno.
  return !*s && stClenov.empty();
}
```

(b) Računanje vrednosti števila pravzaprav ni dosti bolj zapleteno od preverjanja, ali je niz sploh veljavno število, kar smo naredili v prejšnji podnalogi. Razlika je predvsem ta, da ko v številu $s = (s_1 s_2 \ldots s_k)$ beremo člene $s_1, s_2, \ldots, s_k$, zdaj ne bo dovolj, če jih bomo le šteli (in na koncu preverimo, če sta bila člena vsaj dva), ampak bomo morali tudi seštevati in odštevati njihove vrednosti, da dobimo vrednost števila

s. Toda ko preberemo člen s_i, še ne moremo vedeti, ali bo treba njegovo vrednost prišteti ali odšteti, saj je to odvisno od tega, ali za njim stoji še člen s_{i+1} z večjo vrednostjo. Zato je koristno poleg trenutne vrednosti števila s hraniti tudi vrednost prejšnjega prebranega člena; za začetek vrednost vsakega člena prištejmo, ko pa preberemo naslednji člen, ga primerjajmo s prejšnjim, da vidimo, če bi bilo treba prejšnjega v resnici odšteti (v tem primeru ga potem odštejmo dvakrat, ker smo ga prej enkrat že prišteli). Paziti pa moramo tudi na možnost, da smo od prejšnjega člena že odšteli predprejšnji člen, zato prejšnjega člena zdaj ne moremo odšteti od naslednjega. Naš postopek bo torej deloval približno takole:

$v := 0$; $p := \infty$;
while nismo pri zaklepaju:
  $x :=$ vrednost naslednjega člena s_i;
  if $p < x$ **then** $v := v - 2p + x$; $p := \infty$;
  else $v := v + x$; $p := x$;

Vrednost števila s torej računamo v spremenljivki v; prejšnji člen hranimo v p, z vrednostjo $p = \infty$ pa predstavimo primere, ko prejšnjega člena ni ali pa ta člen ni na voljo za odštevanje (ker smo od njega že odšteli predprejšnji člen).

Če so oklepaji vgnezdeni več nivojev globoko, mora takšen izračun potekati na vsakem nivoju. Lahko bi uporabili sklad, tako kot pri prejšnji podnalogi, vendar bi moral zdaj vsak element sklada poleg števila členov hraniti še v in p. Lažje in preprosteje pa bo, če uporabimo rekurzijo:

```
// Naslednja rekurzivna funkcija prebere marsovsko število, na katero kaže s,
// in premakne s na prvi znak po koncu števila. Funkcija vrne vrednost
// prebranega števila. Če pride do napake, vrne −1 in postavi s na nullptr.
int PreberiNaslednje(const char *&s, int vrednostiCrk[26])
{
  if (! s) return −1;
  if (! *s) { s = nullptr; return −1; }

  // Če je prvi znak črka, jo preberimo in vrnimo njeno vrednost.
  if ('a' <= *s && *s <= 'z') return vrednostiCrk[*s++ − 'a'];

  // Preskočimo začetni oklepaj (če prvi znak ni oklepaj, javimo napako).
  if (*s++ != '(') { s = nullptr; return −1; }

  // Prebirajmo člene do zaklepaja in seštevajmo/odštevajmo njihove vrednosti.
  int v = 0, n = 0, xPrej = 0; bool jePrej = false;
  while (*s && *s != ')')
  {
    // Preberimo naslednji člen in izračunajmo njegovo vrednost.
    int x = PreberiNaslednje(s, vrednostiCrk); n++;
    if (! s) return −1; // Prišlo je do napake.

    // Če imamo prejšnji člen, ki ga nismo uporabili pri odštevanju
    // in ki je manjši od x, ga odštejmo od v in prištejmo njuno razliko.
    if (jePrej && xPrej < x) v = v − xPrej + (x − xPrej), jePrej = false;

    // Sicer le prištejmo trenutni člen in si ga zapomnimo kot
    // novi prejšnji člen.
    else v += x, jePrej = true, xPrej = x;
  }

  // Če nismo pri zaklepaju, je niz neveljavne oblike.
  // Neveljaven je tudi, če smo imeli manj kot dva člena.
```

```
  if (n < 2 || *s != ')') { s = nullptr; return -1; }
  // Sicer preskočimo končni zaklepaj.
  s++; return v;
}
```

Glavni podprogram mora le poklicati PreberiNaslednje in preveriti, če je uspešno prišel do konca niza. Naloga ne pove, kaj naj vrnemo, če niz ni veljavno marsovsko število; spodaj bomo vrnili -1, saj če so vrednosti črk nenegativne, potem je tudi vrednost vsakega veljavnega marsovskega števila nenegativna (o tem se lahko prepričamo z indukcijo po dolžini niza).

```
int Vrednost(const char *s, int vrednostiCrk[26])
{
  int v = PreberiNaslednje(s, vrednostiCrk);
  // Če je PreberiNaslednje postavil s na nullptr, to pomeni, da je prišlo do napake.
  // Po drugi strani, če s še ni na koncu niza, to pomeni, da niz kot celota tudi ni
  // veljavno marsovsko število — npr. "(ee)e". Sicer pa vrnimo v.
  return (!s || *s) ? -1 : v;
}
```

(*c*) Označimo z $g(n)$ odgovor, po katerem sprašuje naša podnaloga, torej število načinov, kako postaviti oklepaje v niz n črk, da nastane veljavno marsovsko število.

Definicija marsovskih števil pravi, da je število oblike $(s_1 \dots s_k)$ dovoljeno le pri $k > 1$ (takemu številu bomo na krajše rekli, da je k-*členo*). Številu, ki ga tvori ena sama črka (in nič oklepajev), pa bomo rekli, da je 1-*členo*.

Označimo s $f(n,k)$ število načinov, na katere je mogoče v niz n črk vriniti oklepaje tako, da nastane k-členo marsovsko število. Ker vsak člen pokrije vsaj eno črko, mora biti $1 \le k \le n$, drugod je $f(n,k) = 0$. Odgovor, po katerem sprašuje naša podnaloga, potem ni nič drugega kot $g(n) := \sum_{k=1}^{n} f(n,k)$.

Če imamo niz dolžine $n = 1$, je edini način, da iz njega dobimo marsovsko število, ta, da oklepajev sploh ne postavimo, tako da imamo $f(1,1) = 1$.

Pri $n > 1$ pa moramo v naš niz nujno postaviti vsaj en par oklepajev (na začetku in koncu niza). S takšnim dodajanjem oklepajev nastane neko k-členo število za nek $k \ge 2$; takrat je torej $f(n,1) = 0$.

Za $n > 1$ in $k > 1$ pa lahko funkcijo $f(n,k)$ računamo rekurzivno. Če hočemo iz niza n črk narediti k-členo marsovsko število za $k \ge 2$, bo prvi člen pokril recimo prvih m črk, ostalih $k-1$ členov skupaj pa preostalih $n-m$ črk. Pri tem vemo, da mora biti $m \ge 1$ (da pokrije prvi člen vsaj eno črko) in $m + k - 1 \le n$ (da ostane tudi za vsakega od preostalih $k-1$ členov še vsaj po ena črka). Za prvi člen vemo, da ga lahko iz m črk naredimo na $g(m)$ načinov. Če je $k = 2$, nam ostane potem le še en člen, ki ga lahko iz preostalih $n-m$ črk naredimo na $g(n-m)$ načinov; pri $k > 2$ pa moramo iz $n-k$ črk narediti $k-1$ členov, kar lahko naredimo na $f(n-m, k-1)$ načinov. Tako smo dobili:

$$\begin{aligned} f(n,k) &= \textstyle\sum_{m=1}^{n-k+1} g(m) f(n-m, k-1) && \text{za } n \ge k > 2 \\ f(n,2) &= \textstyle\sum_{m=1}^{n-k+1} g(m) g(n-m) && \text{za } n \ge 2. \end{aligned}$$

Vidimo, da je ta naloga zelo primerna za reševanje z dinamičnim programiranjem. Funkcijo f računajmo sistematično po naraščajočih n (in pri vsakem n še po vseh

k) ter shranjujmo njene vrednosti v tabelo, kjer nam bodo prišle prav, ko jih bomo potrebovali pri kasnejših izračunih. Sproti lahko vrednosti $f(n,k)$ za različne k pri posameznem n tudi seštevamo in tako dobimo $g(n)$. Oglejmo si implementacijo te rešitve v C++:

```
#include <vector>
using namespace std;

int NaKolikoNacinov(int n)
{
  if (n <= 0) return 0;
  vector<vector<int>> f; vector<int> g;
  f.resize(n + 1); g.resize(n + 1);
  f[1].resize(2); f[1][1] = 1; g[1] = 1;
  for (int N = 2; N <= n; ++N)
  {
    g[N] = 0; f[N].resize(N + 1);
    for (int k = 2; k <= N; ++k)
    {
      int r = 0;
      for (int m = 1; m + k - 1 <= N; ++m)
        r += g[m] * (k == 2 ? g[N - m] : f[N - m][k - 1]);
      f[N][k] = r; g[N] += r;
    }
  }
  return g[n];
}
```

Naslednja tabela kaže rezultate za nekaj majhnih n:[12]

n	1	2	3	4	5	6	7	8	9	10
$g(n)$	1	1	3	11	45	197	903	4 279	20 793	103 049

(d) Pri prejšnji podnalogi smo prešteli, koliko marsovskih števil lahko nastane iz niza n črk z vrivanjem oklepajev; zdaj bi načeloma lahko vsa ta marsovska števila tudi zgenerirali in za vsako od njih izračunali njegovo vrednost s postopkom iz podnaloge (b). Toda ta rešitev je zelo potratna, saj je možnih vrednosti naših marsovskih števil veliko manj kot marsovskih števil samih. O tem se prepričamo takole: iz definicij na začetku besedila naloge vidimo, da lahko vrednost v danega marsovskega števila vedno dobimo tako, da inicializiramo v na 0 in gremo po vseh črkah v številu ter vrednost posamezne črke bodisi prištejemo k v bodisi odštejemo od v. Vse, na kar oklepaji v številu vplivajo, je to, pri katerih črkah se vrednost prišteje k v, pri katerih pa odšteje od v. Ker imamo n črk in pri vsaki le dve možnosti (ali njeno vrednost prištejemo ali odštejemo), vidimo, da lahko nastane pri danem zaporedju n črk največ 2^n različnih vrednosti marsovskega števila.[13] Vseh marsovskih števil nad tistim zaporedjem n črk pa je, kot smo videli pri podnalogi (c), približno $5{,}83^n$,

[12] Na spletni enciklopediji celoštevilskih zaporedij (*The On-line Encyclopedia of Integer Sequences*, OEIS) najdemo to zaporedje pod številko A001003; tam izvemo, da tem številom pravijo tudi mala Schröderjeva števila, naraščajo pa približno tako hitro kot $(3+\sqrt{8})^n$, kar je približno $5{,}83^n$.

[13] Še tesnejšo zgornjo mejo za število različnih vrednosti lahko dobimo, če resno vzamemo podatek, da kot črke v naših marsovskih številih nastopajo le male črke angleške abecede in da ima enaka črka vedno enako vrednost. Abeceda je torej omejena na 26 črk; pa recimo v splošnem, da imamo abecedo z a črkami. Pomembno je, da je ta a omejen, tudi če n narašča

torej se neizogibno velikokrat zgodi, da ima po več različnih marsovskih števil enako vrednost.

Koristno je torej, če se namesto generiranja vseh možnih marsovskih števil nad danim nizom črk ukvarjamo z generiranjem vseh možnih vrednosti, ki jih ta števila imajo; na koncu bomo potem pač vrnili največjo od njih.

Definirajmo torej $f(s)$ (pri čemer je s nek niz n črk) kot množico vrednosti vseh marsovskih števil, ki jih je mogoče dobiti iz s z dodajanjem oklepajev. Če je s dolžine 1, ga torej tvori ena sama črka in edini element množice $f(s)$ je vrednost tiste črke. Pri daljših nizih, $n > 1$, pa lahko iz s nastane veljavno k-člena marsovsko število le za $k > 1$. Takrat moramo torej niz s razbiti na podnize $s_1, \ldots, s_k$ z dolžinami $n_1, \ldots, n_k$ (pri čemer mora biti $n_1 + \ldots + n_k = n$) iz vsakega podniza narediti nekakšno marsovsko število, nato pa ta števila stakniti skupaj in oviti v še en par oklepajev. Tako pridemo do naslednjega postopka za izračun $f(s)$:

funkcija $f(s)$:
  $n :=$ dolžina niza s;
  if $n = 1$ **then return** $\{v\}$, kjer je v vrednost prve (in edine) črke niza s;
  $M :=$ prazna množica;
  for $k := 2$ **to** n:
    za vsak nabor $(n_1, \ldots, n_k)$, kjer je $n_1 + \ldots + n_k = n$:
      razbij s na nize $s_1, \ldots, s_k$ z dolžinami $n_1, \ldots, n_k$;
      za vsako $x_1 \in f(s_1), x_2 \in f(s_2), \ldots, x_k \in f(s_k)$:
        iz zaporedja $x_1, \ldots, x_k$ izračunaj vrednost v po
          definiciji iz besedila naloge;
        dodaj v v množico M;
  return M;

Ko smo torej s razbili na podnize $s_1, \ldots, s_n$, smo upoštevali, da je iz s_i mogoče narediti marsovska števila z vrednostmi iz množice $f(s_i)$, in za vsako kombinacijo teh vrednosti $x_i \in f(s_i)$ (za $i = 1, \ldots, k$) lahko potem izračunamo vrednost k-členega marsovskega števila, pri katerem imajo členi vrednosti $x_1, \ldots, x_k$.

Neugodno pri tej rešitvi je, da so zanke vgnezdene zelo globoko; že v vrstici „za vsak nabor $(n_1, \ldots, n_k)$“ se pravzaprav skriva $k - 1$ vgnezdenih zank, kasneje pa imamo še nadaljnjih k vgnezdenih zank po $x_1, \ldots, x_k$, ki gredo po vseh elementih množic $f(s_i)$ (te množice pa so lahko tudi precej velike).

Bolje je, če ne poskušamo deliti s-ja na k členov naenkrat, pač pa na dva dela $s = s_L s_D$, pri čemer levi del s_L predstavlja prvih $k - 1$ členov, desni del s_D pa zadnji člen. Imamo torej $s_L = s_1 s_2 \ldots s_{k-1}$ in $s_D = s_k$. Lepo bi bilo, če bi lahko možne vrednosti s-ja (torej množico $f(s)$) izračunali iz možnih vrednosti s_L-ja in s_D-ja (torej iz $f(s_L)$ in $f(s_D)$), vendar se pri tem malo zaplete. Označimo s $\tilde{s}_i$ marsovska števila, ki se jih da dobiti z dodajanjem oklepajev v niz s_i. V množici

v neskončnost. V našem nizu s dolžine n črk je recimo n_i izvodov i-te črke abecede (in seveda velja $n_1 + \ldots + n_a = n$). Pri izračunu vrednosti s-ja se lahko vrednost i-te črke abecede največ n_i-krat prišteje ali odšteje; če recimo vrednost te črke označimo z x_i, vidimo, da k vrednosti s-ja prispeva $c_i \cdot x_i$ za neko celo število c_i z območja $-n_i \le c_i \le n_i$. Vseh možnih vrednosti s-ja je torej kvečjemu toliko, kolikor je vseh možnih naborov $(c_1, \ldots, c_a)$, to pa je $\prod_{i=1}^{a}(2n_i + 1)$. Ta produkt je največji takrat, ko so vse črke približno enako pogoste, torej $n_i \approx n/a$, možnih vrednosti s-ja pa je tedaj približno $(n/a)^a$. Lepo pri tej meji je, da je polinomska v odvisnosti od n, ne eksponentna.

$f(s_L)$ imamo torej vrednosti vseh marsovskih števil oblike $\tilde{s}_L := (\tilde{s}_1 \dots \tilde{s}_{k-1})$, v množici $f(s_D)$ pa imamo vrednosti vseh možnih $\tilde{s}_k$. Nas pa za v $f(s)$ v resnici zanimajo vrednosti vseh marsovskih števil oblike $\tilde{s} := (\tilde{s}_1 \dots \tilde{s}_{k-1}\tilde{s}_k)$. Vrednosti takega števila ne moremo preprosto dobiti iz vrednosti marsovskih števil $\tilde{s}_L$ in $\tilde{s}_k$ — to bi se dalo, če bi imeli število oblike $((\tilde{s}_1 \dots \tilde{s}_{k-1})\tilde{s}_k)$, pri nas pa tistega notranjega para oklepajev ni. Če hočemo iz vrednosti števil $\tilde{s}_L$ in $\tilde{s}_k$ dobiti vrednost števila $\tilde{s}$, moramo vedeti vsaj še to, kakšna je bila pri $\tilde{s}_L$ vrednost zadnjega člena v njem, torej števila $\tilde{s}_{k-1}$. To vrednost bomo namreč morali primerjati z vrednostjo števila $\tilde{s}_k$, da bomo pri izračunu vrednosti števila $\tilde{s}$ vedeli, ali moramo vrednost števila $\tilde{s}_{k-1}$ prišteti ali odšteti. Paziti moramo še na možnost, da je bil pri izračunu vrednosti števila $\tilde{s}_L$ zadnji člen $\tilde{s}_{k-1}$ že porabljen za to, da smo od njega odšteli predzadnji člen (torej $\tilde{s}_{k-2}$), zato $\tilde{s}_{k-1}$ ne bo na voljo za odštevanje od $\tilde{s}_k$; to možnost lahko, podobno kot že pri podnalogi (b), predstavimo tako, kot da bi bila vrednost člena $\tilde{s}_{k-1}$ takrat neskončna.

Definirajmo torej zdaj $g(s)$ kot množico vseh takih parov (v, z), za katere velja, da je mogoče z dodajanjem oklepajev iz niza črk s dobiti takšno marsovsko število $\tilde{s}$, ki ima vrednost v, zadnji člen tega števila pa ima vrednost z (če je bil zadnji člen uporabljen za to, da smo od njega odšteli predzadnji člen, se šteje, da je $z = \infty$). Ko bomo enkrat imeli množico $g(s)$, je iz nje seveda zelo lahko dobiti $f(s)$: obdržati moramo le prvo komponento vsakega para, $f(s) = \{v : (v, z) \in g(s)\}$.

Oglejmo si, kako lahko množico $g(s)$ računamo z veliko manj vgnezdenimi zankami kot prej $f(s)$:

funkcija $g(s)$:
  $n :=$ dolžina niza s;
  if $n = 1$ **then return** $\{(v, v)\}$, kjer je v vrednost prve (in edine) črke niza s;
  $M :=$ prazna množica;
  for $m := 1$ **to** $n - 1$:
    $s_L :=$ prvih m znakov niza s; $s_D :=$ zadnjih $n - m$ znakov niza s;
    za vsak $(v_L, z) \in g(s_L)$ in vsak $v_D \in f(s_D)$:
      (* *Vemo, da je mogoče iz s_L narediti neko marsovsko število $\tilde{s}_L$ z vrednostjo v_L in zadnjim členom z vrednostjo z in da je mogoče iz s_D narediti neko marsovsko število $\tilde{s}_D$ z vrednostjo v_D. Iz s je potem mogoče narediti marsovsko število $(\tilde{s}_L\tilde{s}_D)$.* *)
      if $v_L < v_D$ **then** dodaj $(v_D - v_L, \infty)$ v M
      else dodaj $(v_L + v_D, v_D)$ v M;
      (* *Pri $m > 1$ je $\tilde{s}_L$ veččleno število, torej $\tilde{s}_L = (\tilde{s}_1 \dots \tilde{s}_\ell)$ za nek $\ell > 1$. Iz s je potem mogoče narediti tudi marsovsko število $(\tilde{s}_1 \dots \tilde{s}_\ell\tilde{s}_D)$. Hitro pa se vidi, da ni nobene škode, če spodnji stavek **if** izvedemo tudi pri $m = 1$, saj se takrat obnaša enako kot prejšnji.* *)
      if $z < v_D$ **then** dodaj $(v_L - z + v_D - z, \infty)$ v M
      else dodaj $(v_L + v_D, v_D)$ v M;
  return M;

Tu imamo le tri gnezdene zanke — po m, eno po parih iz $g(s_L)$ in eno po vrednostih iz $f(s_D)$ (slednjo izračunamo iz $g(s_D)$). S prvim stavkom **if** v glavni zanki smo pokrili primere, ko iz s z dodajanjem oklepajev naredimo dvočleno marsovsko število, z drugim pa primere, ko iz s naredimo k-člено število za $k > 2$. V obeh primerih iz

tega, kar vemo o $\tilde{s}_L$ (namreč njegovo vrednost v_L in vrednost njegovega zadnjega člena z) in o $\tilde{s}_D$ (namreč njegovo vrednost v_D) ni težko izračunati vrednosti števila $\tilde{s}$ (in njegovega zadnjega člena), ki ga naredimo iz s.

Vidimo lahko, da nam pri rekurziji nastanejo klici funkcije g tudi za vse podnize prvotnega s-ja; kot je pri dinamičnem programiranju običajno, si je pametno množice $g(t)$ in $f(t)$ za vse podnize t našega prvotnega s-ja zapomniti, saj jih bomo pri izračunu še večkrat potrebovali. Če je več podnizov enakih (npr. v $s =$ `abab` se podniz `ab` pojavi na dveh mestih), bodo imeli tudi enako $g(t)$, tako da je smiselno paziti tudi na to, da računamo $g(t)$ v takem primeru le enkrat, četudi se isti podniz pojavi na več mestih v s-ju.

(e) Če je naše marsovsko število sestavljeno iz ene same črke, je naloga trivialna, saj tako število oklepajev sploh nima in jih tudi ne moremo brisati. V nadaljevanju se bomo torej ukvarjali le s števili oblike $(s_1 \ldots s_k)$. V marsovskem številu z n črkami je lahko kvečjemu $n-1$ parov oklepajev (o tem se lahko prepričamo z indukcijo po n). Za vsak par razen najbolj zunanjega imamo dve možnosti: lahko ga pobrišemo ali pa ne; zunanjega para pa ne smemo pobrisati, saj bi število potem postalo neveljavne oblike. Tako lahko torej z brisanjem oklepajev dobimo do 2^{n-1} različnih marsovskih števil. Lahko si predstavljamo postopek, ki bi zgeneriral vsa ta števila, nato pa s postopkom iz podnaloge (b) izračunal njihove vrednosti in pogledal, katera imajo enako vrednost kot prvotno število in katero med takimi ima najmanj oklepajev. Toda ta rešitev je časovno potratna, saj imajo mnoga med tistimi 2^{n-1} števili marsikaj skupnega, zato je neučinkovito, če se ukvarjamo z vsakim od njih posebej.

Namesto tega začnimo razmišljati podobno kot pri podnalogi (d). Tam smo najprej definirali funkcijo, ki za dani niz vrne množico vrednosti vseh marsovskih števil, ki jih je mogoče dobiti z dodajanjem oklepajev v tisti niz; tukaj bi nas bolj zanimala množica vrednosti vseh marsovskih števil, ki jih je mogoče dobiti z brisanjem oklepajev iz niza. Toda te vrednosti same še niso dovolj, saj potrebujemo za vsako vrednost tudi podatek o tem, koliko največ oklepajev lahko pobrišemo, da to vrednost dosežemo. Naj bo torej $f(s)$ množica vseh možnih parov (v, m), za katere velja, da je z brisanjem m parov oklepajev iz s mogoče dobiti marsovsko število z vrednostjo v in da takega števila ni mogoče dobiti z brisanjem več kot m parov oklepajev. (Matematično gledano je taka množica pravzaprav funkcija, v računalniku pa bi jo predstavili z razpršeno tabelo oz. slovarjem.)

Vprašanje je zdaj, kako bi poceni izračunali množico $f(s)$ za $s = (s_1 \ldots s_k)$, pri čemer bi si želeli seveda pomagati z množicami $f(s_1), \ldots, f(s_k)$. Pri podnalogi (d) smo v podobni situaciji ugotovili, da ni dovolj, če o podnizih poznamo le vrednosti števil, ki se jih dá dobiti iz njih, ampak smo morali poznati tudi vrednost zadnjega člena v vsakem takem številu (in to, ali je bil ta člen že uporabljen v odštevanju), saj ta člen vpliva na izračun, ko več takih števil staknemo skupaj. Pri našem sedanjem problemu brisanja oklepajev je stvar podobna, vendar malo bolj zapletena.

Recimo, da imamo $s = (s_1 s_2 s_3)$ in da je število s_2 tudi sámo veččleno, torej $s_2 = (t_1 \ldots t_\ell)$. Ko brišemo oklepaje iz s, je zdaj poleg tega, da pobrišemo nekaj oklepajev znotraj podštevila s_2, mogoče pobrisati tudi zunanji par oklepajev števila s_2, tako da iz s nastane na primer $\tilde{s} = (s_1 t_1 \ldots t_\ell s_3)$. V tem primeru zdaj prvi člen števila s_2, to je t_1 (oz. nekaj, kar je iz njega nastalo po morebitnem brisanju

oklepajev), stoji za s_1 in pri izračunu vrednosti števila $\tilde{s}$ bomo morali znati primerjati vrednosti števil s_1 in t_1, da bomo videli, ali ju moramo odšteti. In če t_1 uporabimo v takšnem odštevanju, vpliva to na nadaljevanje izračuna; pri izračunu vrednosti števila s_2 smo mogoče odšteli t_1 od t_2 (in zato nismo odšteli t_2 od t_3), zdaj pa, če smo odšteli s_1 od t_1, števila t_1 gotovo ne bomo odšteli od t_2, je pa zato zdaj mogoče, da bomo odšteli t_2 od t_3 — in tako naprej. Pri številu s_2 nas torej ne bo zanimala le njegova vrednost, ampak tudi njegova „alternativna“ vrednost, kakršno bi imel, če bi se pri izračunu dogovorili, da njegovega prvega člena nikakor ne bomo odšteli od drugega (četudi je manjši od drugega).

Pri izračunu vrednosti $\tilde{s}$ se lahko tudi zgodi, da bo treba razmisliti o odštevanju t_ℓ od s_3, tako da bomo o številu s_2 želeli vedeti tudi vrednost njegovega zadnjega člena — podobno, kot smo hoteli že pri podnalogi (d). Če je bil zadnji člen že uporabljen v odštevanju (ker smo od njega odšteli $t_{\ell-1}$), si tudi tokrat mislimo, da ima vrednost ∞. Če računamo alternativno vrednost števila s_2, torej tako, da t_1 zagotovo ne odštejemo od t_2, se lahko vrednost zadnjega člena zaradi tega spremeni (ker smo mogoče od njega prej odšteli $t_{\ell-1}$, zdaj pa ne ali pa obratno), zato moramo vzdrževati tudi vrednost zadnjega člena za ta primer.

Zdaj imamo vse, kar potrebujemo za naš izračun. Za poljubno marsovsko število t definirajmo *ključ* $K(t) = (v, v', p, z, z', b)$, ki nam o t-ju pove naslednje stvari: če izračunamo vrednost t-ja po običajnem postopku, ima vrednost v, njegov zadnji člen pa z (če smo pri izračunu vrednosti t-ja odšteli njegov predzadnji člen od zadnjega, vzamemo $z = \infty$); če izračunamo vrednost t-ja po alternativnem postopku, torej tako, da prvega člena nikakor ne odštejemo od drugega, ima t vrednost v', njegov zadnji člen pa z' (če smo od njega odšteli predzadnji člen, vzamemo $z' = \infty$); prvi člen t-ja ima vrednost p; b pa je logična vrednost, ki pove, ali je t veččleno število ($b = \textsc{false}$ pomeni, da je t ena sama črka). Ni seveda nujno, da so vse te vrednosti različne; na primer, če je t oblike $(t_1 \dots t_\ell)$ in ima t_1 večjo vrednost od t_2, potem je $v' = v$ in tudi $z' = z$, saj takrat dodatna omejitev, da t_1 ne smemo odšteti od t_2, ničesar ne spremeni (ker ga tudi sicer ne bi odšteli od t_2). Pri z in z' opazimo še, da se lahko razlikujeta le tako, da je eden ∞, drugi pa ne; če imata oba končno vrednost, mora biti pri obeh enaka (in sicer enaka vrednosti zadnjega člena, to je t_ℓ).

Definirajmo zdaj še funkcijo g takole: za poljubno marsovsko število s naj bo njena vrednost, $g(s)$, slovar, v katerem kot ključi nastopajo $K(t)$ za vsa števila t, ki jih je mogoče dobiti iz s z brisanjem oklepajev; pripadajoča vrednost pri vsakem takem ključu pa je največje možno število oklepajev, ki jih še lahko pobrišemo, če hočemo doseči neko število s tem ključem.

Ko bomo za število s, ki nas zanima, znali izračunati $g(s)$, ne bo težko poiskati odgovora, po katerem sprašuje naša naloga: v slovarju $g(s)$ bomo morali pregledati vse tiste ključe, katerih v je enak vrednosti prvotnega števila s, in med pripadajočimi vrednostmi vseh teh ključev bomo morali vrniti največjo.

Zdaj nam ostane le še razmislek o tem, kako računati funkcijo g. Če je s ena sama črka z vrednostjo x, je stvar preprosta: oklepajev nima in jih tudi ne moremo brisati, zato bo v $g(s)$ en sam ključ, $K(s) = (x, x, x, x, x, \textsc{false})$, s pripadajočo vrednostjo 0. Za veččlena števila oblike $s = (s_1 \dots s_k)$ pa lahko, podobno kot pri podnalogi (d), računamo $g(s)$ iz $g((s_1 \dots s_{k-1}))$ in $g(s_k)$. Preden zapišemo psevdokodo za izračun

funkcije g, še opomba glede dela s slovarji: pri naštevanju vsebine slovarja bomo predpostavili, da nam slovar vrača pare $\langle K, m\rangle$, pri čemer je K nek ključ, m pa njemu pripadajoča vrednost; pri dodajanju novega takega para $\langle K, m\rangle$ v slovar pa bomo predpostavili, da če ključ K v slovarju že obstaja s pripadajočo vrednostjo m', se pri takem dodajanju ta pripadajoča vrednost le popravi na $\max\{m, m'\}$.

funkcija $g(s)$:
  if je s ena sama črka z vrednostjo x:
    return slovar z enim samim ključem, $(x, x, x, x, x, \text{FALSE})$,
        in s pripadajočo vrednostjo 0;
  (* *Sicer je s oblike $(s_1 \dots s_k)$. Najprej si oglejmo možnost, da pobrišemo morebitne oklepaje okrog s_1.* *)
  $h := g(s_1)$;
  if je s_1 veččleno število (torej ne ena sama črka):
    $h' :=$ prazen slovar;
    za vsak $\langle (v, v', p, z, z', b), m\rangle \in h$:
      (* *Ena možnost je, da oklepajev okrog s_1 ne pobrišemo.* *)
      dodaj v h' par $\langle (v, v, v, v, v, \text{FALSE}), m\rangle$;
      (* *Lahko pa jih pobrišemo.* *)
      dodaj v h' par $\langle (v, v', p, z, z', b), m + 1\rangle$;
    $h := h'$;
  (* *Preglejmo zdaj še ostale člene.* *)
  for $i := 2$ **to** k:
    (* *Na tem mestu je $h = g((s_1 \dots s_{i-1}))$. Izračunajmo $h' = g((s_1 \dots s_i))$.* *)
    $h' :=$ prazen slovar;
    za vsak $\langle (v_1, v_1', p_1, z_1, z_1', b_1), m_1\rangle \in h$
    in vsak $\langle (v_2, v_2', p_2, z_2, z_2', b_2), m_2\rangle \in g(s_i)$:
      (* *Imamo „levi del“, ki je nastal iz $s_1 \dots s_{i-1}$ po morebitnem brisanju raznih oklepajev, in „desni del“, ki je podobno nastal iz s_i.* *)
      (* *Če je $b_2 =$ TRUE, ima s_i zunanji par oklepajev. Tedaj je ena možnost ta, da te oklepaje okrog s_i pobrišemo in ga potem pritaknemo k levemu delu. Takrt pride prvi člen desnega dela, p_2, tik za zadnji člen levega dela, z_1 (ali z_1'), tako da moramo preveriti, če ju je treba odšteti. Zadnji člen staknjene celote pa je potem zadnji člen števila s_i, to je z_2 ali z_2'.* *)
      if b_2 **then**
        if $z_1 < p_2$ **then** $v_3 := v_1 - 2z_1 + v_2'$; $z_3 := z_2'$;
            else $v_3 := v_1 + v_2$; $z_3 := z_2$;
        if $z_1' < p_2$ **and** b_1 **then** $v_3' := v_1' - 2z_1' + v_2'$; $z_3' := z_3'$;
            else $v_3' := v_1' + v_2$; $z_3' := z_2$;
        dodaj v h' par $\langle (v_3, v_3', p_1, z_3, z_3', \text{TRUE}), m_1 + m_2 + 1\rangle$;
      (* *Druga možnost pa je, da desni del, torej s_i, pritaknemo k levemu, ne da bi mu pobrisali zunanje oklepaje (če jih sploh ima). Takrat moramo preveriti, če je treba s_i odšteti od zadnjega člena levega dela, to je od z_1 (ali z_1'). Zadnji člen staknjene celote pa je potem kar s_i sam.* *)
      if $z_1 < v_2$ **then** $v_3 := v_1 - 2z_1 + v_2$; $z_3 := \infty$;

```
                else v3 := v1 + v2; z3 := v2;
    if z1' < v2 and b1 then v3' := v1' − 2z1' + v2; z3' := ∞;          (⋆)
                       else v3' := v1' + v2; z3' := v2;
    dodaj v h' par ⟨(v3, v3', p1, z3, z3', TRUE), m1 + m2⟩;
  h := h';
return h;
```

Zakaj potrebujemo v pogoju v vrstici (⋆) poleg „**if** $z_1' < v_2$“ še „**and** b_1“? Takrat računamo vrednost staknjene celote z dodatno predpostavko, da prvega člena te celote ne smemo odšteti od drugega. S pogojem „**if** $z_1' < v_2$“ smo preverili, če je zadnji člen levega dela (z_1') takrat manjši od prvega člena desnega dela (v_2 — tu ima desni del tako ali tako en sam člen, saj mu nismo brisali morebitnih zunanjih oklepajev); takrat bi bilo torej načeloma treba z_1' odšteti od v_2; toda če je $b_1 = \text{FALSE}$, to pomeni, da ima levi del en sam člen, tako da je njegov zadnji člen hkrati tudi prvi člen celote, za tega pa smo rekli, da ga ne smemo odšteti od drugega. Del „**and** b_1“ v pogoju imamo torej zato, da pravilno obdelamo tudi takšne primere. To je tudi razlog, da smo komponento b dodali v naše ključe.

Če je v našem prvotnem s več enakih podštevil — na primer, v `((cc)(cc))` se podštevilo `(cc)` pojavlja dvakrat — bo funkcija g na vsakem od njih vrnila enak slovar, tako da utegne biti koristno, če si že izračunane vrednosti $g(s)$ nekje shranjujemo in pred izračunom preverimo, če smo g za prav enako (pod)število kdaj prej že izračunali.

(f) To podnalogo lahko rešujemo z dinamičnim programiranjem, podobno kot (d), vendar je pri alternirajočih marsovskih številih problem precej lažji. Prej smo morali pri tvorbi števil oblike $(s_1 \dots s_k)$ za vsak člen s_i računati vse možne vrednosti, da smo lahko potem razmišljali o tem, kdaj je en člen manjši od naslednjega in ga je treba zato odštevati. Pri alternirajočih številih pa je to, ali nek člen odštevamo ali prištevamo, odvisno le od njegovega položaja v številu, ne pa tudi od vrednosti sosednjih členov.

Naj bo torej $s[1..n]$ naš začetni niz n črk. Naj bo $f(i,j)$ največja, $g(i,j)$ pa najmanjša vrednost izmed vseh marsovskih števil, ki jih lahko dobimo z dodajanjem oklepajev v podniz $s[i..j]$. Pri $i = j$ je stvar trivialna, $f(i,j) = g(i,j) = x$, če je x vrednost črke $s[i]$. Pri daljših podnizih lahko razmišljamo takole: $s[i..j]$ moramo razbiti na levi del $s[i..r]$ in desni del $s[r+1..j]$ (za nek r z območja $i \le r < j$). Iz levega dela bo nastal po dodajanju oklepajev člen $\tilde{s}_1$, iz desnega pa zaporedje členov $\tilde{s}_2\tilde{s}_3 \dots \tilde{s}_\ell$ za nek $\ell > 1$; na koncu bomo oboje staknili skupaj, zavili v še en par oklepajev in tako dobili število $\tilde{s} = (\tilde{s}_1 \dots \tilde{s}_\ell)$. Naj bo x_i vrednost člena $\tilde{s}_i$. Vrednost števila $\tilde{s}$ je potem $x_1 - x_2 + x_3 - \dots + (-1)^\ell x_\ell$, kar je naprej enako $x_1 - (x_2 - x_3 + \dots + (-1)^{\ell-1} x_\ell)$; izrazu v oklepajih recimo y — vidimo lahko, da je to ravno vrednost števila $(\tilde{s}_2 \dots \tilde{s}_\ell)$ (oz. števila $\tilde{s}_\ell$, če je $\ell = 2$). Vrednost našega števila $\tilde{s}$ je torej $x_1 - y$; če hočemo, da bo največja možna, moramo vzeti največji možni x_1 in najmanjši možni y; tako dobimo vrednost $f(i,r) - g(r+1,j)$. Da bomo dobili največjo vrednost sploh, moramo vzeti maksimum po vseh r; tako dobimo $f(i,j) = \max\{f(i,r) - g(r+1,j) \,:\, i \le r < j\}$. Podoben razmislek nas pripelje tudi do $g(i,j) = \min\{g(i,r) - f(r+1,j) \,:\, i \le r < j\}$. Obe funkciji lahko računamo sistematično od krajših podnizov proti daljšim, že izračunane vrednosti pa sproti pišemo v tabelo, da jih bomo imeli pri roki, ko jih bomo spet potrebovali. Zapišimo

to rešitev s psevdokodo:

for $i := 1$ **to** n:
  naj bo x vrednost črke $s[i]$;
  $f[i,i] := x$; $g[i,i] := x$;
for $d := 2$ **to** n **do for** $i := 1$ **to** $n - d + 1$:
  $j := i + d - 1$;
  $f[i,j] := -\infty$; $g[i,j] := \infty$;
  for $r := i$ **to** $j - 1$:
    $f[i,j] := \max\{f[i,j], f[i,r] - g[r+1,j]\}$;
    $g[i,j] := \min\{g[i,j], g[i,r] - f[r+1,j]\}$;
return $f[1,n]$;

Časovna zahtevnost te rešitve je le $O(n^3)$ in je zdaj — za razliko od podnaloge (d) — neodvisna od tega, koliko različnih vrednosti marsovskih števil je mogoče dobiti iz danega niza. Podobno kot pri (d) bi lahko tudi to rešitev poskusili še malo izboljšati z opažanjem, da če isti podniz nastopa na več mestih v nizu s, bodo tudi pripadajoče vrednosti funkcij f in g vsakič enake, zato jih ni treba računati vsakič znova. Lahko bi si pravzaprav zgradili sufiksno drevo niza s in potem računali f in g za vsako vozlišče tega drevesa.

(g) Preden se lotimo te podnaloge, vpeljimo zapis $[s]$, ki nam bo predstavljal vrednost alternirajočega marsovskega števila s. Velja torej $[s] = x$, če je s ena sama črka z vrednostjo x; potem pa, če so $s_1, \ldots, s_k$ (za $k > 1$) alternirajoča marsovska števila, velja $[(s_1 \ldots s_k)] = \sum_{i=1}^{k} (-1)^{k-1}[s_i]$. Poleg tega pa definirajmo naš zapis $[\cdot]$ še za primere, ko v oglatih oklepajih ni eno samo marsovsko število, ampak stik dveh ali več števil: takrat naj bo $[s_1 \ldots s_k] = [(s_1 \ldots s_k)]$.

Naloga zdaj sprašuje, kako iz s dobiti z brisanjem oklepajev čim večjo možno vrednost. Za primere, ko je s ena sama črka, je stvar trivialna, saj oklepajev ni in jih tudi ne moremo brisati. Recimo zdaj, da imamo s oblike $s = (s_1 \ldots s_k)$. Z brisanjem oklepajev nastane iz s_i niz $\tilde{s}_i$, pri čemer opozorimo na to, da $\tilde{s}_i$ sam po sebi ni nujno veljavno marsovsko število, ker smemo pri s_i pobrisati tudi zunanji par oklepajev — takrat nastane iz njega tak $\tilde{s}_i$, ki je zaporedje dveh ali več marsovskih števil; ker pa bo vse skupaj še vedno vključeno v s, ki zunanje oklepaje ima, bo število, ki na ta način nastane iz s kot celote, še vedno veljavno.

Tako torej iz s nastane $\tilde{s} = (\tilde{s}_1 \ldots \tilde{s}_k)$. Pišimo $\tilde{t} = \tilde{s_1} \ldots \tilde{s}_{k-1}$. Koliko členov ima $\tilde{t}$, v splošnem ne moremo reči; lahko jih je več kot $k - 1$, če smo pobrisali zunanje oklepaje okrog kakšnega s_i (in so tako členi bivšega s_i zdaj postali neposredno členi $\tilde{t}$-ja samega). Če ima $\tilde{t}$ sodo mnogo členov, je potem $[\tilde{s}] = [\tilde{t}] + [\tilde{s}_k]$, če ima liho mnogo členov, pa velja $[\tilde{s}] = [\tilde{t}] - [\tilde{s}_k]$. V obeh primerih bomo največjo vrednost $\tilde{s}$-ja dobili tako, da bomo vzeli največjo vrednost $\tilde{t}$-ja, pri $\tilde{s}_k$ pa hočemo v prvem primeru vzeti največjo, v drugem pa najmanjšo vrednost (ker jo v tem drugem primeru odštevamo). Podobno razmišljamo tudi, če nas znanima najmanjša vrednost $\tilde{s}$-ja.

Glede največje oz. najmanjše vrednosti $\tilde{t}$-ja pa torej zdaj vidimo, da nas bosta zanimali posebej za primere, ko ima $\tilde{t}$ sodo mnogo členov, in posebej za primere, ko ima liho mnogo členov. Na to, koliko členov ima, namreč lahko vplivamo (z brisanjem oklepajev okrog posameznih s_i), in preizkusiti moramo obe možnosti (sodo in liho število členov), da bomo res našli največjo oz. najmanjšo možno vrednost $\tilde{s}$-ja.

Za vsako podštevilo našega prvotnega s-ja — recimo za t — bomo torej želeli izračunati četverico vrednosti $K(t) = (f_S, f_L, g_S, g_L)$, ki nam povedo, da je med števili, ki jih je mogoče dobiti z brisanjem oklepajev iz t-ja in imajo sodo (oz. liho) mnogo členov, največja možna vrednost f_S (oz. f_L), najmanjša pa g_S (oz. g_L). Zapišimo zdaj psevdokodo funkcije K:

funkcija $K(s)$:
  if je s ena sama črka z vrednostjo x **then return** $(-\infty, x, \infty, x)$;
  (* *Sicer je* s *oblike* $(s_1 \dots s_k)$ *za nek* $k > 1$.
    Najprej poglejmo, kaj lahko dobimo iz s_1. *)
  $(f_S, f_L, g_S, g_L) := K(s_1)$;
  (* *Nato postopoma dodajajmo ostale člene.* *)
  for $i := 2$ **to** k:
    (* *V* (f_S, f_L, g_S, g_L) *imamo zdaj* $K(t)$ *za* $t = (s_1 \dots s_{i-1})$ *— oziroma za* $t = s_1$, *če je* $i = 2$. *Radi pa bi, recimo v* $(f''_S, f''_L, g''_S, g''_L)$, *izračunali* $K(u)$ *za* $u = (s_1 \dots s_i)$. *)
    $(f'_S, f'_L, g'_S, g'_L) := K(s_i)$;
    $f''_S := \max\{f_S + f'_S, f_L - g'_L\}$; $g''_S := \min\{g_S + g'_S, g_L - f'_L\}$;
    $f''_L := \max\{f_S + f'_L, f_L - g'_S\}$; $g''_L := \min\{g_S + g'_L, g_L - f'_S\}$;
    (* *Shranimo rezultate v* f_S, f_L, g_S *in* g_L. *)
    $(f_S, f_L, g_S, g_L) := (f''_S, f''_L, g''_S, g''_L)$;
  return (f_S, f_L, g_S, g_L);

Kako smo v gornji psevdokodi prišli do formul za f''_S, f''_L, g''_S in g''_L? Oglejmo si primer. Označimo s $\tilde{t}$, $\tilde{s}_i$ in $\tilde{u}$ števila, ki nastanejo z brisanjem oklepajev iz t, s_i in u. Kdaj ima lahko $\tilde{u}$ sodo mnogo členov?

(1) Ena možnost je takrat, ko imata tako $\tilde{t}$ kot $\tilde{s}_i$ sodo mnogo členov; takrat je $[\tilde{u}] = [\tilde{t}] + [\tilde{s}_i]$, največja možna vrednost takega $\tilde{u}$ je $f_S + f'_S$, najmanjša pa $g_S + g'_S$.

(2) Druga možnost pa je, da imata tako $\tilde{t}$ kot $\tilde{s}_i$ liho mnogo členov; takrat je $[\tilde{u}] = [\tilde{t}] - [\tilde{s}_i]$, največja možna vrednost takega $\tilde{u}$ je $f_L - g'_L$, najmanjša pa $g_L - f'_L$.

Če to dvoje združimo, vidimo, da je največja možna vrednost $\tilde{u}$-ja, če mora imeti sodo mnogo členov, enaka $\max\{f_S, f'_S, f_L - g'_L\}$, najmanjša pa $\min\{g_S + g'_S, g'_L f'_L\}$. Tako smo dobili formuli za f''_S in g''_S. Za primere, ko ima $\tilde{u}$ liho mnogo členov, je razmislek podoben in ga lahko prepustimo bralcu za vajo.

Kakorkoli že, zdaj ko imamo funkcijo K, lahko odgovor, po katerem sprašuje naša naloga, dobimo tako, da izračunamo $K(s)$ in vrnemo $\max\{f_S, f_L\}$ iz četverice, ki nam jo je vrnila $K(s)$.

(h) Doslej smo se pri alternirajočih marsovskih številih izognili potrebi po temu, da bi morali računati množico *vseh* možnih vrednosti števil, ki jih je mogoče dobiti z dodajanjem ali brisanjem oklepajev; tukaj pa se temu ne bo dalo izogniti. To je posledica dejstva, da se pri brisanju oklepajev lahko vrednost posameznih delov števila močno spremeni, pa je na koncu rezultat še vseeno enak kot pred brisanjem. Oglejmo si primer; namesto črk bomo pisali kar njihove vrednosti, mednje pa bomo zaradi preglednosti postavili vejice. Alternirajoče marsovsko število $s = ((((1, 4), 2), 6), (7, 9))$ ima vrednost -9; vidimo lahko tudi, da je sestavljeno iz dveh členov, $s_1 = (((1, 4), 2), 6)$ z vrednostjo -11 in $s_2 = (7, 9)$ z vrednostjo -2. Če v s pobrišemo vse oklepaje razen zunanjih, dobimo $t = (1, 4, 2, 6, 7, 9)$, ki ima

tudi vrednost -9; toda tiste črke, ki so prej tvorile člen s_1 z vrednostjo -11, tvorijo zdaj štiri člene $1, 4, 2, 6$, ki bi, če bi jih obravnavali kot eno število, dali vrednost $1 - 4 + 2 - 6 = -7$, ne pa -11.

Tako torej vidimo, da ko se ukvarjamo z nekim podštevilom našega prvotnega s-ja, ne moremo zares vedeti, kakšno vrednost naj z brisanjem oklepajev naredimo iz njega, saj je to odvisno tudi od tega, kaj bomo naredili z ostalimi podštevili s-ja. Zato bomo morali ravnati podobno kot pri podnalogi (e) in pripraviti množico vseh možnih vrednosti alternirajočih marsovskih števil, ki jih je mogoče dobiti z brisanjem oklepajev iz s.

Je pa še en manjši zaplet. Recimo, da imamo $s = (s_1 s_2)$ in da smo iz s_1 z brisanjem oklepajev naredili $\tilde{s}_1$, iz s_2 pa $\tilde{s}_2$. Če ima s_1 več členov, smemo pobrisati zdaj tudi njegove zunanje oklepaje, tako da je mogoče, da $\tilde{s}_1$ ni alternirajoče marsovsko število, ampak stik več takih števil. To pa pomeni, da pri izračunu vrednosti števila $\tilde{s} = (\tilde{s}_1 \tilde{s}_2)$ ne moremo vnaprej vedeti, ali naj jo računamo kot $[\tilde{s}] = [\tilde{s}_1] - [\tilde{s}_2]$ ali kot $[\tilde{s}] = [\tilde{s}_1] + [\tilde{s}_2]$, saj je to odvisno od tega, ali je $\tilde{s}_1$ stik liho ali sodo mnogo števil. Koristno je torej posebej hraniti množico vrednosti za vsakega od teh dveh primerov.

Definirajmo torej funkcijo g takole: za poljubno alternirajoče marsovsko število s naj bo $g(s)$ slovar, v katerem je prisoten ključ (v, b) za vsako število $\tilde{s}$, ki ga je mogoče dobiti iz s z brisanjem oklepajev; pri tem je v vrednost števila $\tilde{s}$, logična vrednost b pa pove, ali je število členov v $\tilde{s}$ liho; pripadajoča vrednost h ključu (v, b) v slovarju pa pove največje število parov oklepajev, ki jih je mogoče pobrisati iz s, da še dosežemo kakšno število s tem konkretnim (v, b).

funkcija $g(s)$:
  if je s ena sama črka z vrednostjo x **then**
    vrni slovar, ki vsebuje le ključ (x, TRUE) s pripadajočo vrednostjo 0;
  (* *Sicer je* s *oblike* $(s_1 \dots s_k)$.
    Poglejmo najprej, kaj se zgodi, če pobrišemo oklepaje okrog s_1. *)
  $h := g(s_1)$;
  if s_1 ni ena sama črka:
    $h' :=$ prazen slovar;
    za vsak $\langle (v, b), m \rangle \in h$:
      (* *Ena možnost je, da oklepajev okrog* s_1 *ne pobrišemo.* *)
      dodaj v h' par $\langle (v, \text{TRUE}), m \rangle$;
      (* *Lahko pa jih pobrišemo.* *)
      dodaj v h' par $\langle (v, b), m + 1 \rangle$;
    $h := h'$;
  (* *Preglejmo zdaj še ostale člene.* *)
  for $i := 2$ **to** k:
    (* *Na tem mestu je* $h = g((s_1 \dots s_{i-1}))$. *Izračunajmo* $h' = g((s_1 \dots s_i))$. *)
    $h' :=$ prazen slovar;
    za vsak $\langle (v_1, b_1), m_1 \rangle \in h$ in vsak $\langle (v_2, b_2), m_2 \rangle \in g(s_i)$:
      (* *Ena možnost je, da oklepaje okrog* s_i *pustimo* (*če sploh obstajajo*). *)
      if b_1 **then** $v_3 := v_1 - v_2$ **else** $v_3 := v_1 + v_2$;
      dodaj v h' par $\langle (v_3, \textbf{not } b_1), k_1 + k_2 \rangle$;
      (* *Lahko pa oklepaje okrog* s_i *pobrišemo. Na vrednost staknjene celote to*

```
        sicer ne vpliva in ta ostane v_3. *)
      if s_i ni ena sama črka:
        dodaj v h' par ⟨(v_3, b_1 xor b_2), k_1 + k_2 + 1⟩;
    h := h';
  return h;
```

Pri delu s slovarji smo uporabljali enako notacijo kot v rešitvi podnaloge (e).

Ko znamo tako računati funkcijo g, moramo za alternirajoče marsovsko število s, o katerem govori naša podnaloga, izračunati slovar $g(s)$, pregledati v njem tiste ključe (v, b), pri katerih je v enak vrednosti števila s, in vrniti največjo med njihovimi pripadajočimi vrednostmi v slovarju — tisto je potem največje število oklepajev, ki jih lahko pobrišemo, če naj bo na koncu vrednost števila enaka kot na začetku.

Naloge so sestavili: podjetniške hobotnice — Boris Horvat; planinci — Vid Kocijan; zid, šikaku, pretakanje tekočine — Mitja Lasič; najkrajše poti — Matjaž Leonardis in Vid Kocijan; barvanje ograje, krojač — Matija Lokar; dvojno zaokrožanje — Mark Martinec; kolovodja — Jure Slak; kontrolne vsote — Patrik Zajec; volilni kolegij, marsovska števila — 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. e-knjiga

- Objavili smo dve rešitvi: eno, ki bere vhodno besedilo po znakih, in eno, ki ga bere po vrsticah. Oboje je enako dobro. Prav tako je enako dobro tudi, če rešitev prebere celo vhodno besedilo naenkrat v pomnilnik. Rešitev sme tudi predpostaviti neko zgornjo mejo za dolžino posamezne vrstice (npr. da je vrstica dolga največ 100 znakov ali kaj podobnega).

- Če rešitev vhodnega besedila sploh ne bere, ampak predpostavi, da ga kar dobi v nekem seznamu ali tabeli, naj se ji zaradi tega odšteje največ tri točke.

- Format izpisa pri tej nalogi ni posebej predpisan, pomembno je le, da je iz njega razvidno število črk in število samoglasnikov v prebranem vhodnem besedilu.

- Iz primerov v besedilu naloge je razvidno, da se v vhodnem besedilu lahko pojavljajo tako male kot velike črke. Če rešitev deluje samo za male ali pa samo za velike črke, naj se ji zaradi tega odšteje 6 točk.

- Rešitev si lahko pomaga s funkcijami iz standardne knjižnice svojega programskega jezika in naj se ji tega ne šteje v slabo, če deluje pravilno. Tak primer je npr. klic funkcije `toupper` v naših C-jevski rešitvi, da pretvori morebitne male črke v velike; še en primer bi bil, če bi za preverjanje, ali je znak samoglasnik, uporabili `strchr`.

- Rešitev, ki preverja, ali je znak črka, tako, da ima 26 ali celo 52 pogojnih stavkov (ali pa pogojev v enem velikem pogojnem stavku), naj se zaradi tega odšteje 2 točki.

- Pri tej nalogi ni mišljeno, da bi se morala rešitev kaj ukvarjati s tem, kako so znaki v vhodni datoteki zakodirani (npr. ASCII, UTF-8 itd.), oz. se sploh kakorkoli zavedati te problematike.

- Besedilo naloge se dá razumeti tudi tako, kot da je treba šteti male črke posebej in velike črke posebej. Rešitve, ki temeljijo na tej interpretaciji, naj se obravnava kot enako dobre, kot če bi štele oboje črke skupaj.

2. Večerja Franca Jožefa

- Verjetno se bo mnogim reševalcem intuitivno zdelo očitno, da je treba pustiti brez večerje po enega gosta v vsakem tretjem stolpcu, pomembno pa je, da njihov odgovor tudi dovolj dobro utemelji, zakaj naj bi bilo to res. Rešitev brez utemeljitve naj dobi največ polovico možnih točk.

- Pri ocenjevanju utemeljitve je dobro imeti v mislih še tole. Ni težko pokazati, da z določeno razporeditvijo tega, katerim gostom ne prinesemo večerje, uspešno preprečimo, da bi kdorkoli začel jesti. Težje pa je pokazati, da ne obstaja noben drug razpored, pri katerem bi ostalo brez krožnika manj gostov, pa vendarle nihče ne bi začel jesti. Toda ravno to slednje moramo pokazati, če se hočemo prepričati, da je naša rešitev optimalna.

- Naša rešitev se z vsako mizo ukvarja le konstantno mnogo časa. Rešitve, ki bi za mizo dolžine d porabile $O(d)$ časa, naj dobijo največ 13 točk, če so drugače pravilne. Rešitve, ki bi porabile eksponentno veliko časa v odvisnosti od d (npr. ker z rekurzijo preizkusijo vse možnosti glede tega, kateri gostje ne dobijo večerje), naj dobijo največ 7 točk, če so drugače pravilne.

3. Robot

- Pri takšnih nalogah mnogo tekmovalcev piše rešitve, v katerih je skoraj enaka izvorna koda razmnožena v štirih kopijah, po eni za vsako smer. To ni elegantno, ni pa tudi samo po sebi narobe. Takšnim rešitvam naj se odšteje največ tri točke, če so drugače pravilne.
- Rešitev lahko bere opis poti robota po znakih (kot npr. naša rešitev v C++) ali pa célo naenkrat (kot npr. naša rešitev v pythonu); oboje je enako dobro.
- Zaradi morebitnih drobnih napak pri branju vhodnih podatkov naj se rešitvi odšteje največ tri točke.
- V naši rešitvi smo smer popravljali s formulami oblike `(smer + 1) % 4` in podobno, da smo poskrbeli, da `smer` ostane na območju od 0 do 3. Enako dobro je tudi, če tekmovalčeva rešitev za to poskrbi tudi kako drugače, na primer s pogojnim stavkom (`smer += 1; if (smer >= 4) smer -= 4` ipd.). Če pa rešitev nikakor ne upošteva dejstva, da je smer robota cikličen pojem, naj se ji zaradi tega odšteje pet točk.

4. Čokolada

- Naloga pravi, naj bo rešitev čim bolj učinkovita. Rešitve s časovno zahtevnostjo $O(n^3)$ ali slabšo naj dobijo največ 15 točk. Rešitve s časovno zahtevnostjo $O(n^2)$ naj dobijo največ 18 točk.
- Naloga pravi, da si mora Metka izbrati eno ali več zaporednih vrstic. Če bi rešitev pomotoma dovolila tudi, da si Metka izbere 0 vrstic (kar lahko včasih pripelje do boljših rešitev, npr. če je v vsaki vrstici čokolade več rozin kot lešnikov), naj se ji zaradi tega odbije tri točke.
- Če rešitev šteje vrstice od 0 do $n-1$ namesto od 1 do n, naj se ji zaradi tega ne odbija točk (če je drugače pravilna).

5. Golombovo ravnilo

- Ker je dolžina ravnila (s tem pa tudi največje možno število oznak na njem) pri tej nalogi še kar velika, pričakujemo rešitve s časovno zahtevnostjo manj kot $O(n^3)$. Naša rešitev ima časovno zahtevnost $O(n^2)$, vse točke pa lahko dobijo tudi rešitve v času $O(n^2 \log n)$, na primer če rešitev shranjuje že videne razdalje v drevo (npr. razred `map` v C++) ali pa v seznam, ki ga na koncu uredi s quicksortom ali čim podobnim. Rešitve s časovno zahtevnostjo $O(n^3)$ naj dobijo največ 15 točk, take z zahtevnostjo $O(n^4)$ ali slabšo pa največ 10 točk, če so drugače pravilne.

- Pri tej nalogi ni mišljeno, da bi bil poudarek na branju vhodnih podatkov. Za drobne napake pri branju vhodnih podatkov naj se rešitvi odšteje največ 2 točki.

- Format izpisa pri tej nalogi ni posebej določen; glavno je, da program ugotovi, ali je ravnilo Golombovo in ali je popolno, ni pa pomembno, kako to izpiše.

- Lastnosti, po katerih sprašuje naloga (ali je ravnilo Golombovo in ali je popolno), sta neodvisni (ravnilo ima lahko eno, drugo, obe ali nobene od teh dveh lastnosti). Rešitvi, ki preverja le eno od teh dveh lastnosti, ne pa obeh, naj se odbije 7 točk.

Težavnost nalog

Državno tekmovanje ACM v znanju računalništva poteka v treh težavnostnih skupinah (prva je najlažja, tretja pa najtežja); na tem šolskem tekmovanju pa je skupina ena sama, vendar naloge v njej pokrivajo razmeroma širok razpon zahtevnosti. Za občutek povejmo, s katero skupino državnega tekmovanja so po svoji težavnosti primerljive posamezne naloge letošnjega šolskega tekmovanja:

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM
1. e-knjiga	lažja naloga v prvi skupini
2. Večerja	težka naloga v prvi ali lažja v drugi skupini
3. Robot	težja naloga v prvi ali lahka v drugi skupini
4. Čokolada	srednje težka naloga v drugi ali lažja v tretji skupini
5. Golomb	srednje težka naloga v drugi ali lahka v tretji skupini

Če torej na primer nek tekmovalec reši le eno ali dve lažji nalogi, 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.

Zadnja leta na državnem tekmovanju opažamo, da je v prvi skupini izrazito veliko tekmovalcev v primerjavi z drugo in tretjo, med njimi pa je tudi veliko takih z zelo dobrimi rezultati, ki bi prav lahko tekmovali tudi v kakšni težji skupini. Mentorjem zato priporočamo, naj tekmovalce, če se jim zdi to primerno, spodbudijo k udeležbi v zahtevnejših skupinah.

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 eno prvo in tri druge, v drugi skupini pa eno prvo in štiri tretje nagrade. 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 335) in smo jih letos podelili enajst. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot pred leti 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 (letos smo podelili 172 bronastih priznanj).

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	Matej Kralj	1	ZRI + Gimnazija Vič	18	17	16	20	18	89
2Z	2	Tomaž Čuk	3	Vegova Ljubljana	18	20	15	15	15	83
2Z		Filip Štamcar	9	ZRI	20	10	17	19	17	83
2Z	4	Anže Hočevar	1	Gimnazija Vič	20	16	14	13	19	82
3S	5	Tadej Strah	3	Gimnazija Vič	19	13	18	14	17	81
3S	6	Mihael Golob	2	Vegova Ljubljana	15	10	15	20	20	80
S	7	Tadej Tomažič	2	Vegova Ljubljana	17	10	20	14	17	78
S	8	Staš Horvat	3	II. gimnazija Maribor	18	16	11	20	12	77
S	9	Blaž Košir	4	Gimnazija Vič	12	14	15	18	16	75
S	10	Kristjan Komloši	2	ŠC Kranj, Str. gimn.	19	16	0	20	15	70
S		Tadej Kraševec	4	ŠC Novo mesto, SEŠTG	18	6	16	10	20	70
S	12	Jan Zorko	4	ŠC Celje, SŠ za KER	12	18	13	19	7	69
S	13	Dominik Foschini	4	SERŠ Maribor	13	19	0	15	20	67
S	14	Luka Peršolja	2	Gimnazija Vič	14	14	5	17	16	66
S	15	Ana Meta Dolinar	4	Gimnazija Bežigrad	13	18	11	19	4	65
S	16	David Panić	4	SŠTS Šiška	5	10	13	17	18	63
S	17	Jaka Velkaverh	2	Gimnazija Vič	16	8	14	17	7	62
S	18	Iztok Bajcar	3	Gimnazija Vič	10	10	13	16	12	61
S	19	Nik Tomažič	3	SERŠ Maribor	5	8	15	19	12	59
S		Anja Laharnar	2	STPŠ Trbovlje	18	9	15	14	3	59
S		Ella Potisek	2	ZRI + Gimnazija Vič	19	18	7	15	0	59
S	22	Alen Cigler	2	ŠC Celje, SŠ za KER	15	0	11	17	15	58
S	23	Luka Gole	4	ŠC Novo mesto, SEŠTG	15	0	13	12	15	55
S	24	Nina Sangawa Hmeljak	3	Gimnazija Vič	15	10	13	15	1	54
S		Vili Perše	3	STŠ Koper	13	10	11	16	4	54
S	26	Gašper Irman	4	ŠC Velenje, ERŠ	18	0	15	11	9	53
S		Andraž Bajec	4	ŠC Novo mesto, SEŠTG	17	5	10	18	3	53
S		Lovro Lotrič	9	ZRI	14	17	6	13	3	53
S		Anže Kocjančič	3	ŠC Novo mesto, SEŠTG	9	9	4	14	17	53
S	30	Jure Dolar	3	ŠC Celje, Gimn. Lava	16	7	6	16	6	51
S		Luka Lah	3	ŠC Velenje, ERŠ	10	9	6	10	16	51
S	32	Miha Krumpestar	4	SŠ Domžale	3	11	1	15	20	50
S		Klemen Klopčič	1	Gimnazija Bežigrad	15	8	1	19	7	50
S		Nik Vodovnik	9	ZRI	16	7	15	12	0	50
S	35	Mateja Žvegler	4	ŠC Celje, SŠ za KER	6	19	1	16	7	49
S	36	Nejc Cifer	2	Vegova Ljubljana	16	4	6	5	15	46
S		Jan Wulf	3	SERŠ Maribor	12	5	3	19	7	46
S	38	Žan Koren Kern	4	STPŠ Trbovlje	12	8	7	5	13	45
S		Jure Brenčič Jazbec	4	SŠTS Šiška	9	6	12	16	2	45
S	40	Matic Dremelj	9	ZRI	18	0	5	15	6	44
S		Anej Batagelj	1	Gimnazija Bežigrad	12	10	14	8	0	44

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
S	42	Igor Zevnik	2	Vegova Ljubljana	10	0	13	15	4	42
S		Luka Pavčnik	3	ŠC Velenje, ERŠ	7	0	17	10	8	42
S	44	Aljoša Vertot	2	SPTŠ Murska Sobota	10	12	0	16	3	41
S		Tevž Bajželj	3	ŠC Kranj, STŠ Kranj	15	8	17	0	1	41
S		Klemen Šuštar	3	STPŠ Trbovlje	15	0	1	5	20	41
S	47	Anže Goršek	2	ŠC Velenje, ERŠ	13	18	0	7	0	38
S		Tilen Juričan	9	ZRI	18	5	2	10	3	38
S		Tilen Anzeljc	2	Vegova Ljubljana	17	10	0	6	5	38
S		Andraž Šošterič	1	I. gimnazija v Celju	18	9	4	7	0	38
S	51	Rok Hladin	1	I. gimnazija v Celju	5	5	8	11	8	37
	52	Alen Petek	3	ŠC Celje, Gimn. Lava	16	0	0	17	0	33
		Norbert Velušček	2	ZRI	10	3	0	10	10	33
		Matjaž Vuherer	3	ŠC Celje, Gimn. Lava	20	0	0	11	2	33
		Matic Brovč	5	SŠTS Šiška	12	10	0	11	0	33
	56	Tomaž Bizjak	3	ŠC Celje, SŠ za KER	7	10	0	5	10	32
	57	Valentin Romih	1	I. gimnazija v Celju	2	10	7	7	4	30
	58	Timotej Gregorič	4	SŠTS Šiška	8	4	0	10	7	29
		Aleksander Grobelnik	3	ŠC Celje, SŠ za KER	16	3	0	0	10	29
	60	Jure Čufer	4	ŠC N. mesto, SEŠTG	5	10	7	6	0	28
		Martin Belušič	9	ZRI	7	10	1	7	3	28
	62	Matic Kovač	9	ZRI	7	8	0	12	0	27
	63	Marko Vrečer	3	ŠC Celje, SŠ za KER	3	7	0	5	10	25
	64	Timotej Petrovčič	4	Gimnazija Vič	10	5	1	8	0	24
	65	Miha Govedič	1	II. gimnazija Maribor	9	0	10	4	0	23
		Sandi Pečečnik	4	ŠC Velenje, ERŠ	7	0	0	13	3	23
		Domen Hribernik	2	ŠC Celje, SŠ za KER	9	8	1	5	0	23
		Žan Hozjan	4	SPTŠ Murska Sobota	12	0	0	9	2	23
		Žiga Zajc	3	ŠC Kranj, STŠ Kranj	13	0	0	10	0	23
	70	Žan Starašinič	2	ŠC N. mesto, SEŠTG	6	3	1	9	3	22
		Katja Schrader	9	ZRI	0	20	2	0	0	22
		Enej Lah	1	Gimnazija Bežigrad	10	3	0	9	0	22
		Matija Pilko	2	I. gimnazija v Celju	7	10	0	5	0	22
	74	Tim Cesar Ažman	5	ŠC Kranj, STŠ Kranj	3	2	0	16	0	21
		Samo Pungaršek Pritržnik	3	ŠC Velenje, ERŠ	3	7	0	10	1	21
		Nik Jukič	2	STPŠ Trbovlje	7	0	1	13	0	21
		Matic Pristavnik Vrešnjak	3	SŠ Domžale	12	0	0	9	0	21
	78	Nikola Brković	1	Gimnazija Bežigrad	10	6	0	3	0	19
		Mitja Klajnšek	9	OŠ L. Pliberška	0	0	9	10	0	19
	80	Luka Grošelj	4	SŠTS Šiška	3	0	0	15	0	18
	81	Nikita Galuh Kapušin	3	ŠC N. mesto, SEŠTG	3	0	3	11	0	17
		Eva Juvanc	2	ZRI + Gimnazija Vič	14	0	0	3	0	17
	83	Domen Jurkovič	1	Gimnazija Vič	0	0	1	13	0	14

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
84	Gašper Podbregar	2	STPŠ Trbovlje	5	0	1	7	0	13
	Tina Poštuvan	3	Gimnazija Vič	8	5	0	0	0	13
	Anja Rupnik	1	Gimnazija Vič	0	3	1	4	5	13
	Tilen Sapač	2	SPTŠ Murska Sobota	4	0	0	9	0	13
	Tim Novak	3	ŠC Kranj, STŠ Kranj	7	2	0	4	0	13
89	Aljaž Frühvirt	2	SPTŠ Murska Sobota	1	0	0	10	0	11
	Noel Srimac Gajič	1	British Int. School	3	2	1	5	0	11
	Mihael Černjak	1	Gimnazija Murska Sobota	5	1	1	4	0	11
92	Goran Časar	2	SPTŠ Murska Sobota	1	0	0	8	0	9
93	Martin Brandner	1	II. gimnazija Maribor	1	4	0	3	0	8
	Amadeja Rek	1	II. gimnazija Maribor	5	0	0	3	0	8
95	Maj Andrejč	4	Gimnazija Murska Sobota	0	0	0	7	0	7
	Luka Pirš	1	II. gimnazija Maribor	2	0	0	5	0	7
97	Štefan Tot	2	SPTŠ Murska Sobota	3	0	0	2	0	5
	Aljaž Ugovšek	2	Gimnazija Poljane	0	0	0	5	0	5
99	Aljaž Marn	9	ZRI	4	0	0	0	0	4
100	Jaka Zupanc	1	II. gimnazija Maribor	1	0	0	2	0	3
101	Aleks Rakuša	2	SPTŠ Murska Sobota	1	0	0	1	0	2
	Žan Mencigar	4	SPTŠ Murska Sobota	2	0	0	0	0	2
103	Timotej Zadravec	2	SPTŠ Murska Sobota	1	0	0	0	0	1
104	Blaž Gomboši	1	Gimnazija Murska Sobota	0	0	0	0	0	0
	Tilen Jug	1	Gimnazija Murska Sobota	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	Miha Zupan	4	Gimnazija Bežigrad	18	20	13	17	19	87
2Z	2	Matija Likar	1	II. gimnazija Maribor	20	12	19	19	12	82
2Z	3	Luka Pepelnjak	4	Gimnazija Vič	18	13	12	20	18	81
3S	4	Žan Žnidar	3	Gimnazija Kranj	20	15	10	20	15	80
3S		Aljaž Medič	3	ŠC Kranj, Str. gimn.	20	15	15	15	15	80
3S		Miha Krajnc	4	STPŠ Trbovlje	20	14	8	18	20	80
3S	7	Aleksander Piciga	3	Gimnazija Vič	19	14	11	20	12	76
S	8	Miha Frangež	4	SERŠ Maribor	20	15	0	18	20	73
S	9	Rok Štular	2	Gimnazija Bežigrad	7	15	12	18	19	71
S	10	Miha Meglič	3	ŠC Kranj, Str. gimn.	18	12	4	20	16	70
S		Maks Popovič	3	Vegova Ljubljana	20	12	8	15	15	70
S	12	Miha Marinko	4	Gimnazija Vič	20	17	14	17	0	68
S	13	Luka Železnik	4	II. gimnazija Maribor	10	12	9	20	16	67
S	14	Peter Kosem	4	Gimnazija Vič	20	14	0	14	16	64
S		Gregor Kržmanc	3	Gimnazija Vič	19	15	5	20	5	64
S	16	Lenart Arvo Kos	3	Vegova Ljubljana	20	15	8	8	10	61
S		Jakob Kreft	4	Gimnazija Poljane	18	9	6	18	10	61
S	18	Gregor Kovač	3	ZRI	18	12	12	10	8	60
S	19	Matic Conradi	4	I. gimnazija v Celju	10	9	10	15	15	59
S	20	Filip Trplan	1	Gimnazija Vič	20	10	6	15	6	57
S		Job Petrovčič	3	Gimnazija Vič	20	14	8	10	5	57
S	22	Adrijan Mladenić Grobelnik	1	ZRI	18	13	8	5	11	55
S	23	Janez Koprivec	3	Gimnazija Vič	20	15	8	1	12	56
	24	Vid Urh	3	Škof. klas. gimn. Lj.	18	12	8	15	0	53
	25	Domen Ogorevc	2	Gimnazija Vič in ZRI	20	11	6	15	0	52
	26	Blaž Čerenak	1	I. gimnazija v Celju	18	12	13	2	5	50
		David Ošlaj	3	Škof. klas. gimn. Lj.	17	15	3	15	0	50
		Luka Skeledžija	3	Gimnazija Vič	20	13	0	17	0	50
		Domen Kralj	3	Vegova Ljubljana	14	12	7	12	5	50
	30	Lucijan Semprimožnik	4	I. gimnazija v Celju	17	12	12	8	0	49
	31	Mark David Longar	2	Gimnazija Poljane	10	12	8	12	5	47
	32	Aleksandar Georgiev	4	Vegova Ljubljana	10	13	0	12	10	45
	33	Nejc Strelec	4	ŠC Ptuj, ERŠ	10	10	5	3	15	43
		Jan Hrastnik	3	Gimnazija Vič	5	12	1	17	8	43
	35	Domen Kastelic	2	ZRI	19	15	7	1	0	42
	36	Tilen Šket	1	I. gimnazija v Celju	9	12	5	3	12	41
		Matej Remc	3	Škof. klas. gimn. Lj.	14	12	10	5	0	41
	38	Jonas Lasan	4	ŠC Kranj, Str. gimn.	18	15	1	3	3	40
		Janez Ignacij Jereb	2	ZRI	7	11	10	12	0	40
	40	Tjaž Eržen	3	Gimnazija Kranj	9	12	6	12	0	39
	41	Maj Fontana Korošec	4	SEŠG Maribor	18	8	12	0	0	38
	42	Luka Žibert	4	Gimnazija Kranj	12	11	13	0	0	36
	43	Gregor Sevčnikar	2	ŠC Ravne, SŠ Ravne	7	13	1	9	1	31
	44	Tadej Logar	4	ŠC Ravne, SŠ Ravne	14	12	0	3	0	29
	45	Grega Potočnik	2	ŠC Ravne, SŠ Ravne	3	15	5	5	0	28
	46	Jan Konečnik	3	ŠC Velenje, ERŠ	10	0	0	0	8	18

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Jakob Schrader	2	ZRI + Gimnazija Vič	100	32	94	100	100	426
1Z	2	Benjamin Bajd	1	ZRI	97	32	100	74	97	400
2Z	3	Matevž Miščič	4	Gimnazija Vič	100	65		77		242
2Z	4	Lan Sevčnikar	2	II. gimnazija Maribor	100	29		34	77	240
3S	5	Tevž Lotrič	3	Gimnazija Kranj	100	0	97		17	214
3S	6	Bor Grošelj Simić	4	Gimnazija Vič	100	35	30	37		202
S	7	Jernej Jezeršek	4	Vegova Ljubljana	97	82		20		199
S	8	Jon Mikoš	4	Gimnazija Vič	97	26	34	37		194
S	9	Luka Horjak	3	I. gimnazija v Celju	94	0	79	0		173
S	10	Patrik Žnidaršič	2	Gimnazija Vič	100	32	30	10		172
S	11	Matija Kocbek	3	ZRI + I. gim. v Celju	100		34			134
S	12	Lenart Bučar	3	Gimnazija Bežigrad	90	9	27	0		126
S	13	Luka Lonec	2	II. gimnazija Maribor	97					97
	14	Andraž Pevcin	4	ŠC Celje, Gim. Lava	87	0				87
	15	Lan Vukušič	4	Gimnazija Tolmin	41	32	0			73
	16	Luka Toplak	4	Vegova Ljubljana	47					47
	17	Gašper Pistotnik	4	ŠC Celje, Gim. Lava	34	0				34
	18	Miha Korenjak	4	Gimnazija Bežigrad	8	15			0	23
	19	Jan Vasiljević	4	Gimnazija Tolmin	14	0				14
	20	Urh Primožič	4	Škof. klas. gimn. Lj.		0	10			10
	21	Daniel Blažič	2	Gimnazija Vič	0					0
		Gašper Čopi	4	Gimnazija Tolmin	0	0				0
		Lovro Drofenik	2	I. gimnazija v Celju	0		0			0
		Matej Urbas	4	Gimnazija Tolmin	0					0
		Urban Mišmaš	4	Vegova Ljubljana	0			0		0
		Gregor Žunič	4	Gimnazija Bežigrad	0	0				0

VRSTNI RED ŠOL

Da bi spodbudili šole k čim večji udeležbi in čim boljšim rezultatom v vseh treh skupinah, smo začeli leta 2018 objavljati tudi vrstni red šol v neke vrste skupnem seštevku. Posamezni šoli prinesejo točke najboljši štirje tekmovalci iz te šole v prvi skupini, najboljši trije v drugi in najboljša dva v tretji skupini. Točke šole so enake vsoti točk njenih tekmovalcev. Točke, ki jih prispeva tekmovalec k vsoti, se izračuna tako, da se delež točk (od vseh možnih točk), ki jih je ta tekmovalec dosegel na tekmovanju, pomnoži z utežjo za skupino, v kateri je tekmoval. Utež za prvo skupino je 100, za drugo skupino 200 in za tretjo skupino 300.

Mesto	Šola	Točke
1	Gimnazija Vič	1177,8
2	Vegova Ljubljana	796,6
3	I. gimnazija v Celju	627,2
4	II. gimnazija Maribor	616,2
5	Gimnazija Bežigrad	586,4
6	ŠC Kranj, Strokovna gimnazija	450
7	Gimnazija Kranj	438,4
8	STPŠ Trbovlje	326
9	SERŠ Maribor	318
10	Škofijska klasična gimnazija Ljubljana	294
11	ŠC Novo mesto, SEŠTG	231
12	Gimnazija Poljane	221
13	ŠC Velenje, ERŠ	220
14	ŠC Celje, SŠ za KER	208
15	ŠC Celje, Gimnazija Lava	189,6
16	ŠC Ravne na Koroškem, Srednja šola	176
17	SŠTS Šiška	170
18	ŠC Kranj, STŠ Kranj	98
19	SPTŠ Murska Sobota	88
20	ŠC Ptuj, ERŠ	86
21	SEŠG Maribor	76
22	SŠ Domžale	71
23	STŠ Koper	54
24	Gimnazija Tolmin	52,2
25	OŠ Ludvika Pliberška Maribor	19
26	Gimnazija Murska Sobota	18
27	British International School of Ljubljana	11

NAGRADE

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

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Matej Kralj	telefon Samsung Galaxy A8 in ovitek
1	2	Filip Štamcar	telefon Huawei P20 in ovitek
1	2	Tomaž Čuk	telefon Samsung Galaxy A8 in ovitek
1	2	Anže Hočevar	telefon Samsung Galaxy J6+ in ovitek
1	3	Tadej Strah	telefon Samsung Galaxy J6+ in ovitek
1	3	Mihael Golob	miška Razer Death Adder Elite
2	1	Miha Zupan	telefon Huawei P20 in ovitek Dasgupta *et al.: Algorithms*
2	2	Matija Likar	telefon Samsung Galaxy A8 in ovitek Dasgupta *et al.: Algorithms*
2	2	Luka Pepelnjak	telefon Samsung Galaxy A8 in ovitek Dasgupta *et al.: Algorithms*
2	3	Žan Žnidar	telefon Samsung Galaxy A8 in ovitek
2	3	Aljaž Medič	telefon Samsung Galaxy J6+ in ovitek
2	3	Miha Krajnc	telefon Samsung Galaxy J6+ in ovitek
2	3	Aleksander Piciga	miška Razer Death Adder Elite
3	1	Jakob Schrader	telefon Huawei P20 in ovitek Raspberry Pi 3 model B Steven S. Skiena: *The Algorithm Design Manual*
3	1	Benjamin Bajd	telefon Samsung Galaxy A8 in ovitek Raspberry Pi 3 model B Steven S. Skiena: *The Algorithm Design Manual*
3	2	Matevž Miščič	telefon Samsung Galaxy A8 in ovitek Raspberry Pi model B Steven S. Skiena: *The Algorithm Design Manual*
3	2	Lan Sevčnikar	telefon Samsung Galaxy A8 in ovitek
3	3	Tevž Lotrič	telefon Samsung Galaxy J6+ in ovitek
3	3	Bor Grošelj Simič	telefon Samsung Galaxy J6+ in ovitek
Off-line naloga — Kolesarji			
	1	Gregor Kikelj	Raspberry Pi 3 model B
	4	Miha Zupan	Raspberry Pi 3 model B

SODELUJOČE ŠOLE IN MENTORJI

British International School of Ljubljana	Demjan Vester, Rok Andree
II. gimnazija Maribor	Aleksander Kelenc, Mitja Osojnik, Mirko Pešec
Gimnazija Bežigrad	Gregor Anželj, Andrej Šuštaršič, Jurij Železnik
Gimnazija Kranj	Zdenka Vrbinc
Gimnazija Murska Sobota	Romana Zver
Gimnazija Poljane	Janez Malovrh, Boštjan Žnidaršič
Gimnazija Tolmin	Jernej Cvek, Špela Čopi
Gimnazija Vič	Klemen Bajec, Marina Trost
Osnovna šola Ludvika Pliberška Maribor	Saša Silič
I. gimnazija v Celju	Luka Zlatečan
Srednja ekonomska šola in gimnazija Maribor (SEŠG)	Aleksander Kelenc, Mitja Osojnik
Srednja elektro-računalniška šola Maribor (SERŠ)	Vida Motaln, Slavko Nekrep, Mitja Osojnik, Branko Potisk
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Simon Horvat, Igor Kutoš, Dominik Letnar
Srednja šola Domžale	Tadej Trinko
Srednja šola tehniških strok Šiška	Peter Krebelj, Maruša Perič Vučko, Boris Ribaš
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Srednja tehniška šola Koper	Andrej Florjančič
Šolski center Celje, Gimnazija Lava	Karmen Kotnik
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Dušan Fugina
Šolski center Kranj, Srednja tehniška šola	Miha Baloh
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG)	Albert Zorko, Simon Vovko

Šolski center Ptuj, Elektro in računalniška šola (ERŠ)	Zoltan Sep, Franc Vrbančič
Šolski center Ravne na Koroškem, Srednja šola Ravne	Gorazd Geč, Zdravko Pavleković
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Škofijska klasična gimnazija Šentvid	Helena Starc Grlj
Vegova Ljubljana	Marko Kastelic, Nataša Makarovič, Darjan Toth, Aljaž Vulčini

Zavod za računalniško izobraževanje (ZRI), Ljubljana

OFF-LINE NALOGA — KOLESARJI

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. Opis naloge in testne primere smo objavili decembra 2018, nekaj mesecev po razpisu za tekmovanje v znanju; tekmovalci so imeli čas do 22. marca 2019 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Opis naloge

Dani so izidi več kolesarskih dirk. Na vsaki od njih je nastopilo istih n kolesarjev in znano je, v kakšnem vrstnem redu so prišli na cilj. Kolesarji dobijo na vsaki tekmi točke glede na mesto, na katero so se uvrstili. Tvoja naloga je sestaviti navidezno kolesarsko ekipo, ki jo sestavlja k kolesarjev. Pred prvo dirko si lahko izbereš poljubnih k kolesarjev, pred vsako naslednjo dirko pa lahko največ m kolesarjev svoje navidezne ekipe zamenjaš z drugimi. Cilj je maksimizirati skupno število točk, ki jih dosežejo kolesarji v tvoji navidezni ekipi. (Pri vsakem kolesarju štejejo njegove točke na vseh tistih dirkah, pri katerih je bil v tvoji navidezni ekipi.)

Primer: recimo, da imamo pet kolesarjev (označeni so s številkami od 1 do 5), tri dirke in da na posamezni dirki kolesar dobi 100 točk za prvo mesto, 80 za drugo, 60 za tretje, 40 za četrto in 30 za peto. Recimo, da so bili izidi posameznih dirk takšni, kot jih kaže spodnja tabela:

Mesto		1.	2.	3.	4.	5.
Točke		100	80	60	40	30
Izidi posameznih dirk	prva	5	3	2	4	1
	druga	3	4	5	1	2
	tretja	1	2	5	3	5

Recimo, da bi radi sestavili navidezno ekipo treh kolesarjev, pri čemer smemo po vsaki dirki največ enega zamenjati. Nekaj možnih scenarijev:

- Pred prvo dirko izberemo kolesarje 1, 2, 3 in nikoli nobenega ne zamenjamo. Tako dobimo $(30 + 60 + 80) + (40 + 30 + 100) + (100 + 80 + 40) = 560$ točk.

- Pred prvo dirko izberemo kolesarje 1, 2, 4; pred drugo dirko ne zamenjamo nobenega, pred tretjo pa zamenjamo kolesarja 1 s kolesarjem 3. Tako dobimo $(30 + 60 + 40) + (40 + 30 + 80) + (80 + 40 + 30) = 430$ točk.

- Pred prvo dirko izberemo kolesarje 2, 3, 5; pred drugo dirko zamenjamo kolesarja 2 s kolesarjem 1; pred tretjo dirko zamenjamo kolesarja 3 s kolesarjem 2. Tako dobimo $(60 + 80 + 100) + (40 + 100 + 60) + (100 + 80 + 60) = 680$ točk.

Možnih je seveda še veliko drugih scenarijev. Izkaže se, da je tisti s 680 točkami najboljši možni, tisti s 430 točkami pa najslabši možni pri teh vhodnih podatkih.

Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogah v prejšnjih letih. Pripravili smo 27 testnih primerov, pri vsakem testnem primeru smo razvrstili tekmovalce po številu točk njegove navidezne kolesarske ekipe, nato pa je prvi tekmovalec (tisti, čigar navidezna kolesarska ekipa je dosegla največ točk) dobil 10 točk, drugi 8, tretji 7 in tako naprej po eno točko manj za vsako naslednje mesto (osmi dobi dve točki, vsi nadaljnji pa po eno). Na koncu smo za vsakega tekmovalca sešteli njegove točke po vseh 27 testnih primerih.

Največji testni primeri so imeli do 1000 kolesarjev in 1000 dirk, večina pa je bila manjših. Velikost ekipe je bila pri večini primerov omejena na 20 kolesarjev ali še manj, pri nekaj primerih pa je bila ekipa velika 50 ali 100 kolesarjev. Šest testnih primerov je bilo dovolj majhnih, da bi se dalo optimalno rešitev najti z dinamičnim programiranjem. Uvrstitve kolesarjev na posamezni dirki smo generirali naključno: za vskega kolesarja i smo najprej izbrali povprečni čas vožnje μ_i iz enakomerne porazdelitve na intervalu $[240, 300]$ (recimo, da merimo čase v minutah), nato pa smo za posamezne dirke generirali čase tega kolesarja po normalni porazdelitvi $N(\mu_i, \sigma^2)$, pri čemer je bila $\sigma = 15$ ali 30 enaka za vse kolesarje. Tako so torej nekateri kolesarji v povprečju boljši od drugih, vendar je na vsaki dirki prisoten tudi element naključnosti, ki lahko vpliva na vrstni red. Nekaj testnih primerov pa je namesto naključnih rezultatov vsebovalo resnične rezultate z dirke Tour de France 2018.

Letos je svoje rešitve pri off-line nalogi poslalo kar dvanajst tekmovalcev, od tega šest srednješolcev in pet študentov. Končna razvrstitev je naslednja:

Mesto	Ime	Letnik	Šola	Točke
1	Gregor Kikelj	1	FMF	270
2	Tim Poštuvan	1	FRI + FMF	210
3	Matija Bolko	3	FMF	192
4	Miha Zupan	4	Gimn. Bežigrad	183
5	Samo Kralj	5	FMF	166
6	Jakob Drusany	3	ERŠ Nova Gorica	131
7	Franci Obid	1	ŠC Nova Gorica, VSŠ	89
8	Vid Urh	3	Škof. klas. gimnazija	85
	Bartolomej Kozorog	3	ŠC Nova Gorica	85
10	Andrej Golčer	–	—	55
11	Uroš Koritnik	2	FRI	54
12	Matija Kocbek	3	I. gimn. v Celju	36

Rešitev

Naloge se lahko lotimo na različne načine, odvisno od števila kolesarjev in velikosti ekipe. Pri majhnih testnih primerih lahko poiščemo optimalno rešitev z dinamičnim

programiranjem. Naj bo $f(t, A)$ največje število točk, ki jih lahko dosežemo v prvih t dirkah, če hočemo po t-ti dirki imeti ekipo A. To lahko računamo po naraščajočih vrednostih t:

$$f(t, A) = \mathit{točke}(t, A) + \max_B f(t-1, B),$$

pri čemer $\mathit{točke}(t, A)$ pomeni število točk, ki jih v t-ti dirki dosežejo kolesarji iz množice A, maksimum v drugem členu vsote pa računamo po vseh tistih B, ki imajo k kolesarjev in ki se od A razlikujejo v največ m kolesarjih (tako smo upoštevali omejitev, da smemo pred vsako dirko zamenjati največ m kolesarjev). Robni primer je prva dirka, $t = 1$, kjer ta drugi člen odpade. Na koncu pa, po zadnji dirki — recimo d-ti dirki — nam je vseeno, kakšna je takrat naša ekipa, zato takrat vrnemo $\max_A f(d, A)$, pri čemer gre A po vseh ekipah s k kolesarji. Možnih ekip je $\binom{n}{k}$, zato je ta rešitev obvladljivo hitra le pri majhnem številu kolesarjev (majhnem n).

Pri večjih testnih primerih lahko kolesarje izbiramo s požrešnim algoritmom. Pred vsako dirko vržemo iz ekipe tistih m kolesarjev, ki bodo na njej dosegli najmanj točk (izmed vseh kolesarjev, ki so bili trenutno v naši ekipi), nato pa vanjo vzamemo m tistih kolesarjev (izmed vseh, ki jih zdaj ni v naši ekipi), ki bodo na prihajajoči dirki dosegli največ točk. V rešitev lahko poskusimo dodati tudi nekaj inercije tako, da poleg trenutne dirke gledamo še točke posameznega kolesarja na naslednjih nekaj dirkah, vendar jih pri tem utežimo z utežjo, manjšo od 1.

Zmagovalna rešitev pa je temeljila na predelavi naloge v problem celoštevilskega linearnega programiranja (*integer linear programming*). To je sicer v splošnem NP-težak problem in zanj ne poznamo učinkovitega algoritma, ki bi zagotavljal optimalno rešitev v polinomskem času, vendar pa obstajajo zanj algoritmi in knjižnice, ki v sprejemljivo kratkem času dovolj dobro rešijo marsikak praktično zanimiv problem celoštevilskega linearnega programiranja. Tako se je izkazalo tudi pri naši nalogi.

Za vsakega kolesarja i in vsako dirko t vpeljimo celoštevilsko spremenljivko a_{it}, ki nam pove, ali je ta kolesar na tisti dirki v naši navidezni ekipi ($a_{it} = 1$) ali ne ($a_{it}) = 0$).

Za vsaka i in t imejmo omejitev $0 \le a_{it} \le 1$. S tem poskrbimo, da ima a_{it} res vedno vrednost ali 0 ali 1.

Za vsako dirko t imejmo omejitev $\sum_i a_{it} = k$. S tem poskrbimo, da je v naši navidezni ekipi na vsaki dirki res točno k kolesarjev.

Poskrbeti moramo še za to, da se pred posamezno dirko ne bo zamenjalo več kot m kolesarjev naše ekipe. To lahko naredimo na primer tako, da za vsaka i in t vpeljemo še eno celoštevilsko spremenljivko, b_{it}, ki nam pove, ali je ta kolesar na tisti dirki prišel v ekipo ($b_{it} \ge 1$) ali ne ($b_{it} = 0$). Pri tem naj slednja možnost (da ni prišel v ekipo) pokrije tako primere, ko je bil že od prej v ekipi in je tudi ostal v njej, kot primere, ko ga po novem ni v ekipi (prej pa je lahko bil ali pa tudi ne). Potem lahko za vsako dirko t vpeljemo omejitev $\sum_i b_{it} \le m$, ki bo zagotovila, da v ekipo pred to dirko pride največ m kolesarjev. Tega, da bo enako število (drugih) kolesarjev ekipo tudi zapustilo, pa nam ni treba posebej preverjati, saj bo to zagotovljeno že z eno od prej omenjenih omejitev, ki poskrbijo, da je velikost ekipe vedno točno k kolesarjev.

Vprašanje je le še, kako poskrbeti, da bodo imele spremenljivke b_{it} primerne vrednosti, kot smo si jih zamislili v prejšnjem odstavku, torej nekaj v stilu $b_{it} =$

$\max\{0, a_{it} - a_{i,t-1}\}$. V ta namen je dovolj že, če za vsaka i in t vpeljemo omejitvi $b_{it} \geq 0$ in $b_{it} \geq a_{it} - a_{i,t-1}$. V primerih, ko kolesar i vstopi v ekipo, bo $a_{it} - a_{i,t-1} = 1$ in obe omejitvi skupaj bosta dali $b_{it} \geq 1$; v ostalih primerih pa bo $a_{it} - a_{i,t-1}$ bodisi -1 (če kolesar izstopi iz ekipe) bodisi 0 (če ostane v ekipi ali pa ostane zunaj ekipe), tako da bosta obe omejitvi skupaj dali $b_{it} \geq 0$. Vsota $\sum_i b_{it}$ je torej večja ali enaka od števila kolesarjev, ki vstopijo v ekipo, zato bo omejitev $\sum_i b_{it} \leq m$ uspešno poskrbela, da v ekipo pred dirko t ne bo vstopilo več kot m kolesarjev.

V okviru doslej naštetih omejitev za spremenljivke a_{it} in b_{it} moramo zdaj maksimizirati kriterijsko funkcijo oblike $\sum_i \sum_t w_{it} a_{it}$, pri čemer je w_{it} konstanta, ki pove, koliko točk doseže kolesar i na dirki t. Omenjena dvojna vsota je torej skupno število točk, ki jih doseže naša navidezna kolesarska ekipa na vseh dirkah skupaj.

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, `tekmovanja.acm.si/upm`) 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 od 29. novembra do 1. decembra 2019 v Pragi), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bo od 21. do 26. junija 2020 v Moskvi).

Na letošnjem UPM je sodelovalo 42 ekip s skupno 122 tekmovalci, ki so prišli s treh slovenskih univerz, nekaj pa je bilo celo srednješolcev (in ena osnovnošolka!). Tabela na naslednjih dveh straneh prikazuje vse ekipe, ki so se pojavile na vsaj enem krogu tekmovanja.

	Ekipa	Št. rešenih nalog*	Čas
1	Žiga Željko (FRI + FMF), Žan Knafelc (FDV), Tim Poštuvan (FRI + FMF)	24	26:28:31
2	Urban Duh, Gregor Kikelj, Andraž Maier (FMF)	23	24:46:38
3	Jakob Schrader, Daniel Blažič, Patrik Žnidaršič (Gim. Vič)	21	27:04:09
4	Bor Grošelj Simić (Gim. Vič), Benjamin Bajd, Tevž Lotrič (Gim. Kranj)	20	21:20:30
5	Boshko Koloski, Bozhidar Stevanoski, Ilija Tavchioski (FRI)	19	24:20:12
6	Aljaž Eržen, Marko Rus, Žiga Vene (FRI + FMF)	18	20:33:59
7	Mitja Žalik, Vid Keršič, Niko Uremović (FERI)	18	27:45:09
8	Miha Rajter, Domen Vreš (FRI + FMF), Timen Stepišnik Perdih (FRI)	17	27:49:16
9	Vid Drobnič, Matej Marinko, Žiga Patačko Koderman (FMF)	16	21:07:52
10	Matija Kocbek, Luka Horjak, Lovro Drofenik (I. gimn. v Celju)	16	25:53:51
11	Žiga Šmelcer (FE), Žiga Gradišar (FMF), Lojze Žust (FRI)	15	17:30:16
12	Bor Brecelj (FRI + FMF), Zala Erič, Miha Benčina (FRI)	14	14:48:41
13	Martin Domajnko, Tilen Koren, Jakob Kordež (FERI)	13	17:54:01
14	Žan Hafner Petrovski, Tine Makovecki, Tinka Pirc (FMF)	13	17:58:04
15	Sandi Režonja (FRI)	13	20:45:18
16	Jan Rozman, Ines Meršak, Matic Oskar Hajšen (FMF)	12	10:15:51
17	Andrej Kolar-Požun, Jan Jezeršek, Miha Rot (FMF)	12	14:26:57
18	Rok Strah, Gašper Golob, Jakob Zmrzlikar (FMF)	12	15:35:14
19	Tadej Petrič, Nejc Zajc, Žan Bajuk (FMF)	12	17:54:17
20	Sebastian Mežnar, Maja Gornik, Anže Alič (FRI + FMF)	11	14:58:41
21	Aljaž Žel, Urban Prah, Samo Miklavc (FERI)	11	22:01:07
22	Žan Magerl, Domen Grzin (FRI), Urban Cör (FRI + FMF)	10	9:45:36
23	Tomaž Tomažič (—), Tadej Tomažič (Vegova), Blaž Blokar (FRI)	10	14:40:23
24	Eva Silađi, Vladimir Bošković, Mirza Redžić (FAMNIT)	10	15:34:29
25	Tilen Jesenko, Goran Tubić, Gašper Moderc (FAMNIT)	9	10:24:05
26	Anna Sidorova (FERI)	8	9:19:21
27	Maruša Lekše, Anja Pirnat, Iris Ulčakar (FMF)	8	12:02:33
28	Vili Perše, Jani Bangiev, Aleš Špeh (STŠ Koper)	7	14:22:42
29	Robi Novak, Mihael Baketarić (FERI), Peter Bernad (FNM)	5	7:33:12
30	Gregor Brantuša, Marko Hostnik, Tim Štuhec (FRI + FMF)	5	12:35:46
31	Dario Gavranović (F. za upravo), Tomaž Martinčič, Tim Lunar (FRI)	4	9:36:29

* 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
32	Davor Ornik, Urban Knupleš, Janko Gruden (FERI)	3	2:31:37
33	Mirza Krbezlija, Hasan Mahmutagić, Arber Avdullahu (FAMNIT)	3	3:31:30
34	Arsen Matej Golubovikj, Predrag Zvezdakoski, Hristijan Marinkovski (FAMNIT)	3	3:42:21
35	Jaka Vrhovec, Janez Justin (FRI + FMF)	3	6:08:20
36	Damjan Dimitrov, Aleksandr Kobrin, Daniil Baldouski (FAMNIT)	2	2:12:20
37	Ella Potisek (Gim. Vič), Urša Mati Djuraki (Gim. Bežigrad), Katja Schrader (OŠ Majde Vrhovnik)	2	2:40:47
38	Jani Kaukler, Marko Kužner, Luka Kobale (FERI)	2	4:10:27
39	Rok Kos (FRI + FMF), Benjamin Benčina (FMF)	2	4:35:13
40	Martin Jurkovič, Frenk Dragar, Rene Ratej (FRI)	2	4:48:07
41	Jaka Basej (FMF), Đorđe Jovanović, Sara Veber (FRI + FMF)	2	4:54:06
42	Matevž Rom (FRI + FMF), Domen Šlejkovec (FRI)	0	0:00:00
	Veno Lan Banovšek, Luka Dragar (FRI), Aljaž Škof (FMF)	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 5 kot predstavnice Univerze v Ljubljani, ekipi 7 in 13 kot predstavnici Univerze v Mariboru in ekipa 24 kot predstavnica Univerze na Primorskem. V konkurenci 63 ekip s 25 univerz iz 6 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
30	Žan Knafelc, Tim Poštuvan, Žiga Željko	7	13:39
33	Urban Duh, Gregor Kikelj, Andraž Maier	7	22:16
43	Vid Keršič, Mitja Žalik, Niko Uremović	4	9:14
45	Boshko Koloski, Bozhidar Stevanoski, Ilija Tavchioski	4	11:59
47	Tilen Koren, Martin Domajnko, Jakob Kordež	2	2:20
59	Vladimir Bošković, Mirza Redžić, Eva Silađi	1	1:40

Na srednjeevropskem tekmovanju je bilo 12 nalog, od tega so jih najboljše ekipe rešile po enajst.

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 169–176.

Letnik: ☐ 8. r. OŠ ☐ 9. r. OŠ ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Kako si izvedel(a) za tekmovanje?
☐ od mentorja ☐ na spletni strani (kateri? _____)
☐ od prijatelja/sošolca ☐ drugače (kako? _____)
Kolikokrat si se že udeležil(a) kakšnega tekmovanja iz računalništva pred tem tekmovanjem? _____
Katerega leta si se udeležil(a) 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(a) 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(a) 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(a) v programu uporabiti naslednje podatkovne strukture:

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

Ali bi znal(a) 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++, 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(a), da bi objavljali deklaracije (tudi) v kakšnem drugem programskem jeziku? Če da, v katerem? _____

V rešitvah nalog trenutno objavljamo izvorno kodo v C++ (v 1. skupini pa tudi v pythonu).
— Ali razumeš C++ (oz. python) dovolj dobro, da si lahko kaj pomagaš z izvorno kodo v naših rešitvah? ☐ da ☐ ne
— Ali bi raje videl(a), 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#, javi in pythonu. 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(a) prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal(a) 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#), dict, set (v pythonu)	☐	☐	☐
map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#)	☐	☐	☐
priority_queue (v C++), PriorityQueue (v javi), heapq (v pythonu)	☐	☐	☐

[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

Zakaj si se odločil(a) sodelovati na tem tekmovanju? _____

Česa si se na tem tekmovanju naučil(a)? _____

Ali je udeležba na tem tekmovanju potrdila ali spremenila kakšne od tvojih pogledov na računalništvo? _____

Ali bi udeležbo na tem tekmovanju priporočil(a) tudi drugim dijakom/dijakinjam? Zakaj oz. zakaj ne? _____

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 85 tekmovalcev prve skupine, 36 tekmovalcev druge skupine in 17 tekmovalcev tretje skupine. Vprašanja so bila pri letošnji anketi enaka kot lani, novost je bilo le nekaj vprašanj na predzadnji strani ankete o tem, zakaj so se odločili sodelovati na tekmovanju, ali so se na njem naučili česa novega, kako je tekmovanje vplivalo na njihove poglede na računalništvo in ali bi udeležbo priporočili tudi drugim.

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. 170. 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.

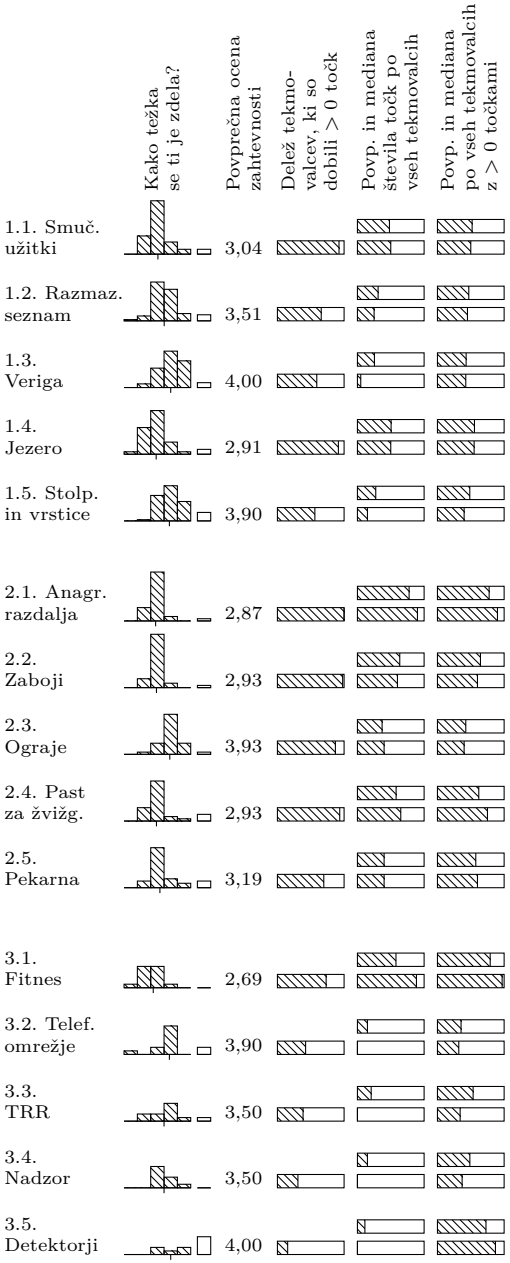
V povprečju so se zdele tekmovalcem v vseh skupinah naloge še kar težke, vendar so številke podobne kot v prejšnjih letih, le v prvi skupini so se jim zdele malo težje. Če pri vsaki nalogi pogledamo povprečje mnenj o zahtevnosti te naloge (1 = prelahka, 3 = primerna, 5 = pretežka) in vzamemo povprečje tega po vseh petih nalogah, dobimo: 3,47 v prvi skupini (v prejšnjih letih 3,32, 3,11, 3,31, 3,41, 3,40), 3,17 v drugi skupini (prejšnja leta 3,19, 3,51, 3,65, 3,33, 3,44) in 3,52 v tretji skupini (prejšnja leta 3,59, 3,73, 3,43, 3,61, 3,19).

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 ponavadi (šibka) negativna korelacija; letos je bila precej močna, podobno kot lani in predlani ($R^2 = 0{,}71$; v prejšnjih letih 0,67, 0,70, 0,39, 0,56, 0,14, 0,52, 0,21, 0,11, pred tem okoli 0,4).

V prvi skupini so tekmovalci kot težki ocenili predvsem nalogi 1.3 (veriga) in 1.5 (stolpci in vrstice), kar je zanimivo, ker se naloge na temo obdelave besedila in nizov ljudem običajno ne zdijo posebej težke. Res pa je, da smo na tekmovanju nalogo, podobno nalogi 1.3, pred leti že imeli (2010.2.2, reka presledkov) in da so jo tudi takrat ocenjevali kot razmeroma težko. V drugi skupini se jim je zdela najtežja naloga 2.3 (ograje), kjer je mogoče algoritem v rešitvi res malo bolj zapleten kot pri ostalih, v tretji skupini pa nalogi 3.2 (telefonsko omrežje) in 3.5 (detektorji). Algoritem pri nalogi 3.5 sicer ni posebej zapleten, je pa z njo še kar nekaj dela in tudi besedilo ima dolgo.

Kot najlažji so tekmovalci v prvi skupini ocenili nalogi 1.4 (jezero) in 1.1 (smučarski užitki); v drugi skupini nalogo 2.1 (anagramske razdalje), ki so jo tudi najuspešneje reševali; in v tretji skupini nalogo 3.1 (fitnes).

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 171. Nad razumljivostjo besedil ni veliko pripomb, le v prvi skupini malo več kot prejšnja leta. Kot težje razumljive so ocenili predvsem naloge 1.2 (razmazani seznam), 1.3 (veriga) in 2.5 (pekarna).



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

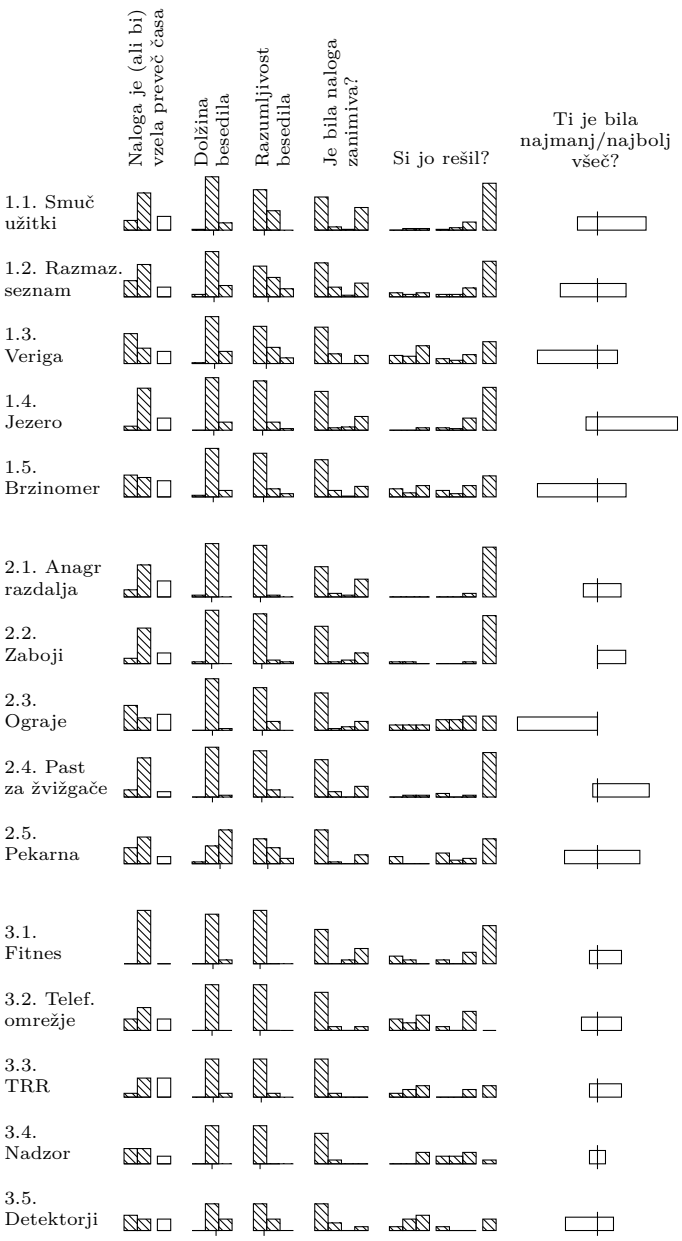
Pomen stolpcev v vsaki vrstici:

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?

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, približno enako kot v prejšnjih letih ali še malo bolj. Pri tem še najbolj odstopata nalogi 2.5 (pekarna) in 3.5 (detektorji), ki imata tudi res daljše besedilo, kot je to običajno. Mnenj, da je kakšno besedilo prekratko, je bilo letos zelo malo.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta, le v prvi skupini so malo nižje. Pripomb, da jim je neka naloga že znana, je bilo letos približno toliko kot lani; največ jih je bilo pri nalogi 1.4 (jezero). Kot bolj zanimive izstopajo 3.3 (transakcijski računi), 2.3 (ograje) in 1.4 (jezero), kot manj zanimivi pa 1.2 (razmazani seznam) in 1.3 (veriga).

Pripomb, da bi naloga vzela preveč časa, je bilo malo več kot ponavadi, še posebej v prvi skupini. Največ takih pripomb je bilo pri nalogah 1.3 (veriga), 2.3 (ograje) in 3.5 (detektorji). Pri zadnjih dveh je z rešitvijo res nekaj več dela, pri 1.3 pa ne toliko. So pa to bolj ali manj iste naloge, ki so se jim zdele tudi najzahtevnejše.

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, sta bili v prvi skupini najmanj priljubljeni nalogi 1.3 (veriga) in 1.5 (stolpci in vrstice), najbolj pa 1.4 (jezero). V drugi skupini kot nepriljubljena izrazito izstopa 2.3 (ograje), najbolj všeč jim je bila 2.4 (past za žvižgače), pri nalogi 2.5 (pekarna) pa je bilo precej glasov tako za „najbolj všeč“ kot za „najmanj všeč“. V tretji skupini je bila največ tekmovalcem najmanj všeč naloga 3.5 (detektorji), glasovi za „najbolj všeč“ pa so precej razpršeni med vse naloge.

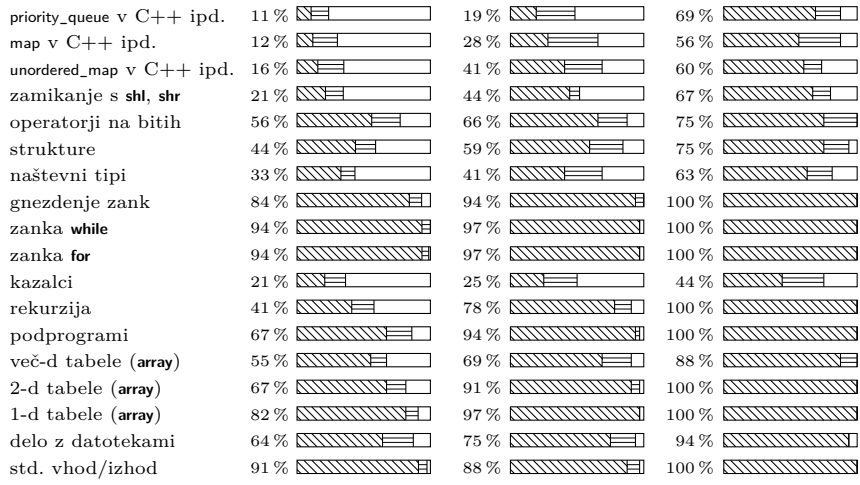


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

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

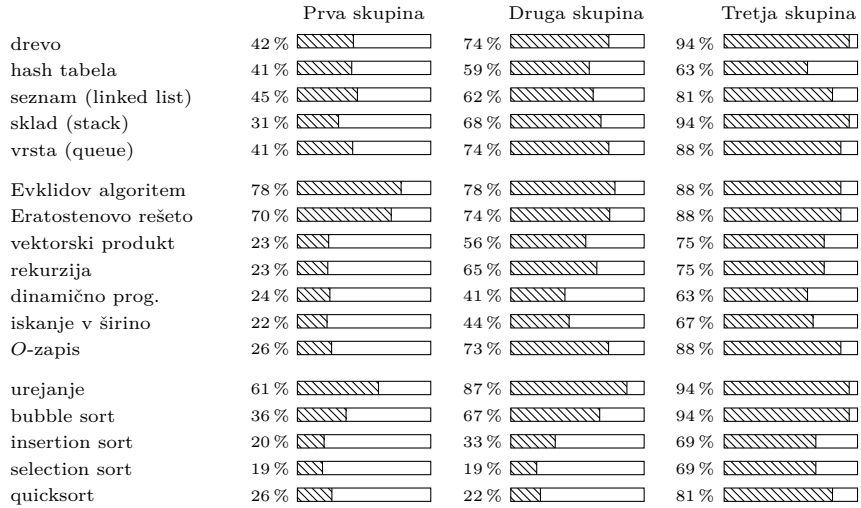


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

podatkovno strukturo. Zato jih v anketah zadnjih nekaj let sprašujemo, če te reči poznajo in bi jih znali uporabiti v svojih programih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let. V povprečju sicer v vseh treh skupinah pravijo, da znajo malo manj kot v lanski anketi. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: kazalci, naštevni tipi in operatorji na bitih, v prvi in drugi skupini tudi strukture in rekurzija. Kazalce pozna letos še manj ljudi kot lani (kar sicer najbrž ni čudno, saj jih veliko dela v jezikih, kjer s kazalci nimajo veliko opravka).

Uporaba programskih jezikov

Na splošno so razmerja med različnimi jeziki podobna kot v prejšnjih letih. V prvi skupini je letos z občutno prednostjo najpogostejši jezik python, sledi mu C++, nato pa java in C#. Več kot lani je bilo v prvi skupini uporabnikov C-ja. V drugi spremembi letos prvič python izrazito prevladuje (doslej smo imeli veliko uporabnikov pythona večinoma le v prvi skupini), sledi mu C++ in nato java in C. V tretji skupini je C++ daleč najpogostejši, precej pa je bilo tudi uporabnikov pythona (čeprav vedo, da so načeloma na slabšem, ker časovne omejitve niso prilagojene počasnosti izvajanja pythonovskih programov). Javascripta in basica ni letos uporabljal nihče, pascal pa le dva v prvi skupini.

Podobno kot prejšnja leta se je tudi letos pojavilo nekaj tekmovalcev, 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

	Leto in skupina																	
	2019			2018			2017			2016			2015			2014		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	2						4			$\frac{1}{3}$	3		5	2		$2\frac{1}{2}$	2	1
C	10	4	$\frac{1}{2}$	5	4	$\frac{1}{2}$	4	3	$2\frac{1}{2}$	$4\frac{1}{3}$	1	2	3		1	$3\frac{1}{2}$	6	
C++	$21\frac{1}{2}$	$7\frac{1}{2}$	18	$18\frac{1}{2}$	13	11	23	10	$15\frac{1}{2}$	28	8	9	27	9	$9\frac{1}{2}$	19	$4\frac{1}{2}$	$10\frac{1}{2}$
java	15	5	1	$21\frac{1}{2}$	$8\frac{1}{2}$	4	28	3	2	24	6	5	22	6	$3\frac{1}{2}$	23	2	$1\frac{1}{2}$
PHP			–			–			–			–	3		–	2		–
basic			–			–									1		1	–
C#	12	2		11	6		7	6		12	5	1	16	5		12	$1\frac{1}{2}$	2
python	$36\frac{1}{2}$	$26\frac{1}{2}$	$6\frac{1}{2}$	38	11	$\frac{1}{2}$	42	11	–	$29\frac{1}{3}$	12	–	26	1	–	16	6	–
javascript			–		$\frac{1}{2}$	–			–	1		–	1		–	1		–
julia			–			–	1		–			–			–			–
batch			–			–			–			–			–	1		–
psevdokoda	5	1	–	3	1	–	5		–	5		–	6	1	–	10		–
nič	3			2		1				2	3		4	1		4	2	

Število tekmovalcev, ki so uporabljali posamezni programski jezik.

Nekateri uporabljajo po več različnih jezikov (pri različnih nalogah) in se štejejo delno k vsakemu jeziku. (V letu 2019 je en tekmovalec uporabljal C in C++, trije pa python in C++.) „Nič“ pomeni, da tekmovalec ni napisal nič izvorne kode (niti psevdokode, pač pa morda rešitve v naravnem jeziku). 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“.

---

stilu „nato bi s stavkom **if** preveril, ali je spremenljivka x večja od spremenljivke y“). Podobno kot prejšnja leta smo tudi letos pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“ (kjer je bilo to primerno).

Podobno kot v prejšnjih letih je v anketi nekaj tekmovalcev napisalo, da dobro poznajo tudi PHP in/ali javascript, vendar ju na tekmovanju letos ni uporabljal nihče.

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže gornja tabela. Glede štetja C in C++ v tej tabeli je treba pripomniti, da je razlika med njima majhna in včasih pri kakšnem krajšem kosu izvorne kode že težko rečemo, za katerega od obeh jezikov gre. Je pa po drugi strani videti, da se raba stvari, po katerih se C++ loči od C-ja, sčasoma povečuje; zdaj že veliko tekmovalcev na primer uporablja `string` namesto **`char *`** in tip `vector` namesto tradicionalnih tabel (*arrays*). Novosti, po katerih se zadnje različice C++ (od vključno C++11 naprej) razlikujejo od C++98, je letos uporabljalo kar precej tekmovalcev, še posebej v tretji skupini (npr. ranged **`for`**, **`auto`** v novem pomenu, metoda `emplace_back`).

Pri pythonu zdaj praktično vsi uporabljajo python 3 in ne python 2; je pa res, da je pri tako preprostih programih, s kakršnimi se srečujemo na našem tekmovanju, razlika večinoma le v tem, ali `print` uporabljajo kot stavek ali kot funkcijo.

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, C#, pythonu in javi. Delež tekmovalcev, ki pravijo, da deklaracije razumejo, je letos v prvi skupini malo nižji kot lani (70/80), podobno tudi v drugi (29/31), vendar je še vedno visok. Kot običajno so pri vprašanju, ali bi želeli deklaracije še v kakšnem jeziku, nekateri tekmovalci navedli jezike, v katerih deklaracije že imamo, na primer javo ali C#; originalna predloga sta bila

PHP in ruby. V vsakem primeru pa se poskušamo zadnja leta v besedilih nalog izogibati deklaracijam v konkretnih programskih jezikih in jih zapisati bolj na splošno, na primer „napiši funkcijo `foo(x, y)`“ namesto „napiši funkcijo `bool foo(int x, int y)`“.

V rešitvah nalog objavljamo od 2017 izvorno kodo v C++, pri prvi skupini pa tudi v pythonu. Tekmovalce smo v anketi vprašali, če razumejo C++ (ali, v prvi skupini, python) dovolj, da si lahko kaj pomagajo s izvorno kodo v rešitvah, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Večina je s C++ (oz. pythonom) zadovoljna (63/77 v prvi skupini, 22/31 v drugi, 16/16 v tretji); ta delež je v drugi skupini malo nižji kot lani (kar je najbrž povezano s tem, da letos več tekmovalcev v drugi skupini uporablja python), v prvi pa precej višji, mogoče zato, ker smo lani v anketi pozabili omeniti, da objavljamo rešitve za prvo skupino tudi v pythonu. Zanimivo vprašanje je, ali bi s kakšnim drugim jezikom dosegli večji delež tekmovalcev (koliko tekmovalcev ne bi razumelo rešitev v javi? ali v pythonu?). Med jeziki, ki bi jih radi videli namesto (ali poleg) C++, jih največ omenja python in (malo manj) javo. Do nadaljnjega bomo zato še naprej objavljali rešitve nalog za prvo skupino tudi v pythonu in ne le v C++.

Letnik

Običajno so tekmovalci zahtevnejših skupin večinoma v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot prejšnja leta; v prvi skupini so tekmovalci v povprečju malo mlajši kot lani in s tem bližje dolgoletnemu povprečju. Letos je nastopilo tudi nekaj osnovnošolcev, in sicer vsi v prvi skupini.

	Št. tekmovalcev po letnikih						Povprečni
Skupina	9	1	2	3	4	5	letnik
prva	10	20	27	25	21	2	2,4
druga		5	7	18	16		3,0
tretja		1	6	4	15		3,3

Druga vprašanja

Podobno kot prejšnja leta 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, je podobno kot prejšnja leta najpogostejši odgovor, da so se naučili programirati sami (takih sta približno dve tretjini); sledijo tisti, ki so se tega naučili v šoli pri pouku (takih je slaba tretjina). Približno ena tretjina tekmovalcev pa se je naučila programirati (tudi) na krožkih in tečajih.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni, vendar jih je manj kot lani. Med tistimi, ki niso, so mnenja precej razdeljena, najpogostejše kombinacije pa so „več časa, manj nalog“, „več časa, enako nalog“ in „enako časa, manj nalog“.

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, razen zaradi tehničnih težav, ki so letos v prvi in drugi

	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	80	0	4	3	51	38	19	10	7	24	5	43	2	23	44	37	28	21	38	35	34	34	38
II	33	0	1	2	29	10	12	3	5	10	1	18	1	6	21	16	16	13	17	14	8	14	14
III	12	0	3	1	10	7	8	1	0	5	2	6	0	8	4	5	3	6	1	6	8	3	3

skupini onemogočile pisanje rešitev na računalnikih, tako da so morali vsi tekmovalci v teh dveh skupinah reševati na papir.

CVETKE

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

(1.1) Nekdo res ne mara indeksiranja od 1 naprej:

```
# v prvi element prog bomo dali prazno vrednost, ker se je nekdo spomnil,
# da je dobra ideja, da elemente naslavljamo s števili od 1 dalje.
# Očitno tekmovalne naloge sestavljajo Lua programerji.
```

Pri nalogah na računalniških tekmovanjih (ne le pri nas) je sicer indeksiranje od 1 naprej zelo pogosto. To ima najbrž več zveze s podobnimi navadami v matematiki (in konec koncev v vsakdanjem življenju) kot s kakšnim konkretnim programskim jezikom.

(1.1) Rešitev z nezaupanjem v zagotovila iz besedila naloge:

```
if p > 1000: # p naj ne bi bil večji od 1000, ampak zihr je zihr
  raise Exception('Navodila naloge so se mi zlagala!')
```

(1.1) Podprogram, ki poskuša izračunati skupni užitek po več vožnjah po neki progi z dano začetno oceno:

```
def sestzm(ocena, kok):
  for i in range(kok):
    ocena += (ocena / 10) * 9
```

Vidimo lahko, da v resnici prvotno oceno le pomnoži z 1,9 ob vsaki vožnji, tako da se smučarjev užitek tu eksponentno hitro povečuje namesto zmanjšuje. Podobno, a manj dramatično, narašča užitek tudi pri naslednji rešitvi:

```
if j in proge:
  proge[j] /= 0.9
```

(1.1) Zanimiv nov prispevek k izrazoslovju:

> Najprej sem deklariral vse spremenljivke in pa tudi 2 večvrednostni spremenljivki. [...] S **for** zanko najprej shranim vse začetne ocene prog in nastavim vse vrednosti večvrednostne spremenljivke stprog na 0.

Pogled na njegovo izvorno kodo pokaže, da „večvrednostna spremenljivka“ tu pomeni tabelo ali polje (*array*). Lahko si predstavljamo tabele z večvrednostnim kompleksom, kako vihajo nos nad preprostimi skalarnimi spremenljivkami :)

(1.1) Za ljubitelje Prešernove poezije:

> Kaj pa tebe treba je bilo
> računalnik — dete milo
> Meni neizkušeni materi
> Kateri ne ležijo nobeni programi

(1.1) Impresivno nepotrebna vpeljava dodatne spremenljivke:

```
int h;
for (int i = 0; i < stevilo_prog; i++) { h = i;
   map[++h] = 0;
}
```

S h-jem kasneje ničesar ne počne, tako da bi čisto lahko naredil preprosto „**for** (**int** i = 1; i <= stevilo_prog; i++) map[i] = 0;“.

(1.2) Predrzen komentar na koncu ene od rešitev:

```
// naloga ne zahteva izpisa seznama B
```

V resnici naloga zahteva, da elemente B-ja izpisujemo, in sicer že sproti med branjem A-ja (to, da so ga izpisovali šele na koncu, je bila tudi sicer pogosta napaka pri tej nalogi).

(1.2) Verjetno najbolj neučinkovita rešitev letos: za vsako celo število se v zanki zapelje po celem A in preverja, ali je treba to število dodati v množico B ali ne:

```
for (int j = 0; j < Max; j++)
   for (int i = 0; i < A.size(); i++)
      if ((A[i] − m) <= j && j <= (A[i] + v))
         B.push_back(j);
```

V nadaljevanju programa ima še podobno zanko za negativna števila od -1 do Min. Števil Min in Max sicer nikjer ne izračuna, ampak očitno je mišljeno, da moramo vzeti $\min(A) - m$ in $\max(A) + v$.

(1.2) Nezaupanje do unarnega minusa:

```
cin << manjši << večji;   // sprejmem m in v
manjši = 0 − manjši;      // manjši ima predznak − za lažje računanje
```

(1.2) Nekaj zanimivih sintaktičnih razširitev pythona. Za vsak element A-ja naredi naslednje:

```
b.list(A[stevec1] − m <= A[stevec1] + v)    # naredimo seznam b,
                                            # ki ustreza podanim kriterijem
if b2(max) >= b(min):    # če je največji element seznama b2 večji od najmanjšega v b,
   print(b > b2(max)     # izpiši le večje
else: print(b)           # drugače izpiši cel seznam b
b2 = b
```

(1.2) Precej nekoristen pogoj:

```
y = x − m;
z = x + v;
if (y <= x && x <= z) {
```

(1.3) Lepa hibridna sintaksa za pretvarjanje tipov, ki deluje tako v C/C++ kot v pythonu (program je sicer v slednjem):

```
count = (int) (ponovljena[1])
```

(1.3) Rešitev z veliko seznami:

```
# V 9 if zankah ločim črke vsake vrstice in
# jih shranim v liste
L0 = []
L1 = []
L2 = []
...
L80 = []
# ustvarim 80 novih listov, saj je v vsaki vrstici
# največ 80 črk
```

Sezname očitno obvlada, zakaj je torej definiral 81 spremenljivk, namesto da bi imel en seznam z 81 elementi?

(1.4) Nekaj ustvarjalnosti pri neskončnih zankah:

```
neskončno = true;
while (neskončno == true) // neskončen while, ker se bool nikoli ne spremeni
{
```

In pri nekem drugem tekmovalcu:

```
for stevec in range(∞):    # neskončna zanka
```

(1.4) Iz ene od rešitev v pythonu:

```
global globine # GLOBGLOB
```

(1.4) Tekmovalci včasih poskušajo implementirati funkcije, za katere naloga pravi, da so že dane. Tule je en dober letošnji prispevek na to temo:

```
int GlobinaVode() // funkcija, ki pove globino jezera
{
  int globina;
  return globina; // vrne globino
}
```

(1.5) Precej tekmovalcev si pri pretvorbi črk v števila od 0 do 25 pomaga s slovarjem ali seznamom (eden celo z linked listo!), ker najbrž ne vedo, da bi lahko računali preprosto `c - 'A'` (v C/C++) ali `ord(c) - ord('A')` (v pythonu). Težje pa je razumeti, zakaj nekdo, ki to ve, vseeno uporabi slovar:

```
abc = { a: 1, b: 2, c: 3, ..., z: 26 }
```

in v komentar na koncu rešitve napiše:

> [...] če je `crka` številska vrednost črke (jaz sem to rešil z dictom, lahko pa tudi z `ord(črka) - 64`).

(1.5) Boleč način za pretvorbo črk v števila:

```
switch (vnos[a]) {
  case "A": b += 1;
  case "B": b += 2;
  case "C": b += 3;
  case "D": b += 4;
  ———— se nadaljuje...
  case "Z": b += 26;
}
```

Ker je pozabil stavke break, bi se na primer pri znaku Z vrednost b povečala kar za $1 + 2 + \ldots + 26 = 351$.

(1.5) V tej rešitvi so koordinate spremenile spol:

```
nezgosceno.append(koordinat)  # seznam vseh koordinatov
```

(1.5) Zelo nenavadne predstave o tem, kako pretvarjati več-črkovne oznake stolpcev v številke:

> Če je črka, potem izpišem v številčno obliko (upam, da je A = 1, Z = 26) in jo pripišem spremenljivki stznaka, ki ima seštete vse vrednosti (torej AA = 1 + 1 = 2).

(1.5) Marsikdo si je pri pretvarjanju črk v števila pomagal z nizom, ki vsebuje vse črke abecede. Nekateri so si tak niz pripravili takole:

```
abeceda = "".join([chr(c) for c in range(ord('A'), ord('Z') + 1)])
```

S tem ni nič narobe, zanimivo je le to, da je s tem približno dvakrat toliko tipkanja, kot če bi napisal celo abecedo kar kot string literal. (Poleg tega je tak niz že na voljo v pythonovi standardni knjižnici: string.ascii_uppercase.)

(1.5) Nekomu dobri stari ASCII s svojimi 128 znaki ni bil dovolj:

```
// v ascii tabeli so velike črke od 122 do 148
```

Kasneje ima tudi tole lepo sintaktično razširitev:

```
if (znaki[i] == 'A' || 'B' || 'C' || ... || 'Z')
```

(1.5) Rešitev, ki se res potrudi biti čim bolj potratna ne le s pomnilnikom, ampak tudi s časom:

```
string [,] tabela = new string[10000, 20000];
// vnos v tabelo
for (int i = 0; i < tabela.GetLength(0); i++)
   for (int j = 0; j < tabela.GetLength(1); j++)
      for (char k = 'a'; k < 'z'; k++)
         string[i, j] = k;
```

Končni rezultat je torej ta, da imamo veliko tabelo, polno znakov 'y'.

(1.5) Neugodno strukturiran pogojni stavek:

```
if i in stolpci:
   indeks = stolpci.index(i)
elif i in stolpci and crke[crke.index(i) + 1] in stolpci:
   indeks = stolpci.index(i) + stolpci[i + 1]
```

Namen pogoja pri **elif**-u je sicer najbrž ta, da preveri, če je trenutni znak (i) vhodnega niza crke črka in naslednji tudi, toda ker ne pozna indeksa znaka i v nizu crke, ga poskuša najti s crke.index — kar bo seveda narobe, če se isti znak v vhodnem nizu pojavi večkrat.

(1.5) Najbolj mazohistična rešitev letos: za pretvorbo črk v števila ima 26 zajetnih stavkov **if**, po enega za vsako črko.

```
if (znak1 == 'A' || znak1 == 'a') {
   do {
      vsota = 1;
      if (stevec > 1) {
         vsota += 26;
      }
      stevec--;
   } while (stevec >= 1);
} else if (znak1 == 'B' || znak1 == 'b') {
```

in tako naprej do Z. Vse to je napisal ročno (slabih 8 strani), saj tekmovalci v 1. in 2. skupini letos zaradi tehničnih težav večino časa niso mogli delati na računalnikih.

(1.5) Ta rešitev pripiše črkam vrednosti tako, da po abecedi padajo namesto naraščajo:

Če je črka le ena, se izpiše število, ki ga dobi, ko izračuna (`"Z"` − (črka)).

(2.1) Pri tej nalogi je bilo precej rešitev, ki z izračunom rezultata kompliciraj bolj, kot bi bilo treba. Na primer:

```
koraki = min(manjkajoče, odvečne) + abs(manjkajoče − odvečne)
```

Rezultat je seveda enak max(manjkajoče, odvečne). Še en podoben primer pri nekem drugem tekmovalcu:

```
vmesnaRazdalja −= spremenjenih
ostanek −= spremenjenih    # odštejem št., ki se jih spremeni, tako da vem,
                           # koliko jih dodati/izbrisati
razdalja = spremenjenih + max([vmesnaRazdalja, ostanek])   # samo ena od teh dveh
                                                           # je prava
return razdalja # to-do
```

Lahko se torej z odštevanjem in prištevanjem spremenjenih sploh ne bi ukvarjal in bi že na začetku vrnil max(vmesnaRazdalja, ostanek).

(2.1) Impresivno zapleten način, kako ugotoviti dolžino daljšega izmed seznamov s in t:

```
op = 0
while len(s) > len(t): op++; s.pop()      # preveri za manjkajoče črke
while len(s) < len(t): op++; t.pop()      # preveri za odvečne črke
while len(s) == len(t) and len(s) > 0:    # preveri za napačne črke
   op++; s.pop(); t.pop()
return op
```

Rešitev je bila sicer pravilna, le neučinkovita (najprej je niza s in t predelal v seznama, nato pobrisal iz njiju črke, ki so prisotne v obeh, potem pa je izvedel gornji fragment kode).

(2.1) Komentar po inicializaciji precej spremenljivk:

```
# verjetno bi se dal narest s pol manj spremenljivkami, a ne vidim razloga zakaj ne
```

(2.2) Zakaj bi preverjali enakost, če lahko zakompliciramo:

če je ((položaj števca s v seznamu x) + 1) deljeno z (dolžino seznama x + 1) enako 1

(2.3) Še en primer zloglasne „**if** zanke“:

Da preverimo, ali je ograja sestavljena iz le enega zaključenega cikla, mora imeti vsak del ograje le dva sosednja dela. To je mogoče preveriti z daljšo **if** zanko.

(Ta kriterij je tudi sicer nekam sumljiv; lahko imamo na primer več ločenih ciklov, pa ima vsak del ograje vendarle le dva soseda.)

(2.3) Zanimiva sintaktična inovacija v pythonu. To, kar bi sicer moral narediti z običajno zanko **for**, je poskušal prevaliti kar na list comprehension:

```
sum = 0
[sum += x for x in mreza[i][j][1] ]    // seštejemo št. ograj polja
```

To je sicer sintaktična napaka, kajti v list comprehensionu mora biti izraz (*expression*), prireditve pa so v pythonu stavki (*statements*) in ne izrazi.

(2.4) Nekdo ni bil navdušen nad zgodbico pri tej nalogi:

```
// ne podpiram boja proti transparentnosti
```

(2.4) Modrovanje v komentarju na koncu ene od rešitev:

Zakaj, le zakaj sem si bil izbral tak način za določanje in-ov in ter-ov? [...] Lenoba, vseh grdob grdoba (dejansko citat iz Svetega pisma, mislim da nekje iz Knjige Modrosti).

Videti je sicer, da gre bolj za ljudski rek, ki ga je populariziral Slomšek (najdemo ga npr. v njegovem *Malem berilu*, Dunaj 1853, str. 170).

(2.4) Veliko zank za malo haska:

```
for (int j = 0; j < 1; j++) {
for (int g = 0; g < 1; g++) {
for (int k = 0; k < 1; k++) {
for (int p = 0; p < 1; p++) {
for (int o = 0; o < 1; o++) {
```

(2.4) Čudovita kombinacija besed „spremenljive“ in „spremenjene“ (mislil je sicer očitno slednje):

```
if (Math.Pow(2, 5 − spremenljeneBesede − 1) < število) {
   beseda = "in";
   stevilo −= (int) Math.Pow(2, 5 − spremenljeneBesede − 1);
   spremenljeneBesede++;
}
```

(2.5) Posrečeno ime razreda, v katerem je bila glavna funkcija ene od rešitev:

```
public class ConcurrencyNightmare {
   public static void main(String[] args) {
```

(2.5) Eden od tekmovalcev je poskušal izziv, ki ga pošilja drugima dvema računalnikoma, v vsaki iteraciji povečati za 1, obenem pa poskrbeti, da ne bi naraščal v nedogled. To slednje mu je uspelo malo predobro:

```
izziv += 1; izziv %= 1;
```

(2.5) Komentar na začetku ene od rešitev:

```
# mislim, da je ta naloga prikrit napad na pogodbene programerje
# (kar je izredno nesramno (in hkrati zabavno))
# po temeljitem premisleku sem se odločil, da grem raje v psevdokodo
```

(2.5) Rešitev brez veliko potrpljenja do nedelujočih računalnikov:

```
    v primeru, da ga računalniki serjejo, jih z veseljem ugašam;
```

(3.2) Pri tej nalogi je imelo veliko rešitev težave s prekomerno porabo pomnilnika, npr. ker so v ogromni tabeli označevale, katera polja so pokrita. Tole je eden od bolj ekstremnih primerov:

```
#define dno 100000000
int mreza[2 * dno][2 * dno];
```

Ta rešitev se ni niti prevedla, ker bi bila globalna spremenljivka `mreza` prevelika. Isti tekmovalec je poskušal tudi z `dno` $= 10$ in z `dno` $= 10000$, kar se je prevedlo, vendar se je program potem vsakič sesul, ker so bile nekatere koordinate oddajnikov večje od 10000 (kaj šele od 10).

(3.2) Še ena izredno neučinkovita rešitev pri tej nalogi: najprej z zanko po oddajnikih določi najmanjšo in največjo x- in y-koordinato pokritih polj, nato pa ima dve gnezdeni zanki po gromozanskem pravokotniku, ki je očrtan vsem pokritim poljem; za vsako polje tega pravkotnika potem z zanko po oddajnikih preverja, ali je pokrito:

```
for (int i = xmin; i <= xmax; i++)
  for (int j = ymin; j <= ymax; j++)
    for (int k = 0; k < n; k++)
      if (abs(i − a[k][0]) + abs(j − a[k][1]) <= a[k][2]) {
        res++;
        break;
      }
```

(3.2) Najgloblje gnezdenje template argumentov letos:

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

(3.3) Za ljubitelje nepotrebnih zavitih oklepajev:

```
  for (int i = 0; i<n; i++) {
    if (racuni[i][10] == '0') { racun[i] = 0; nekaj[0]++; }
    else { if (racuni[i][10] == '1') { racun[i] = 1; nekaj[1]++; }
    else { if (racuni[i][10] == '2') { racun[i] = 2; nekaj[2]++; }
    ⋮
    else { if (racuni[i][10] == '8') { racun[i] = 8; nekaj[8]++; }
    else { if (racuni[i][10] == '9') { nekaj[9]++; }
  }}}}}}}}
}}
```

(3.4) Boleče neučinkovita rekurzivna rešitev:

```
long long f(int x, int y, int n, int a[][3]) {
  ⋮
  if (f(x, z, n, a) + f(z, y, n, a) < min) {
    min = f(x, z, n, a) + f(z, y, n, a);
```

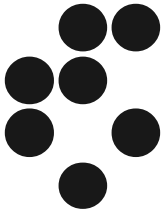
Ker v funkciji ni nikakršnega pomnjenja že izračunanih rezultatov, se bosta klica iz predzadnje vrstice v zadnji vrstici izvedla še enkrat.

(*.*) Letos je bilo precej primerov čudaških imen spremenljivk; spodnji primeri so od štirih različnih tekmovalcev iz vseh treh skupin:

```
long long a, mavricij, vlad;

for (Povezava pnevmatika: ograje) {

var lep, urejen, negovan, [...]: integer;

int grozdje = 0;
int jabolka = (m * (racuni[0][racuni[0].length() − 1] − '0')) % 10;
```

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. 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

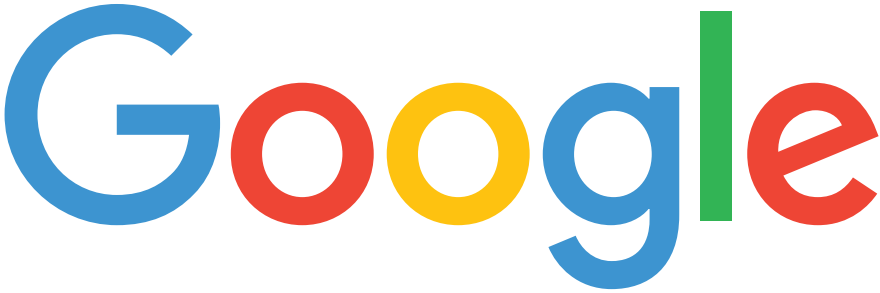
Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih 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.

GENERALNI POKROVITELJ



ZLATI POKROVITELJ

Quintelligence

Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurenčne prednosti. Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja. IKT (informacijsko-komunikacijska tehnologija) je postavila temelje za nemoten pretok in hranjenje podatkov in informacij. S primernimi metodami je potrebno na osnovi teh informacij izpeljati ustrezne analize in odločitve. Nivo upravljanja in delovanja se tako seli iz informacijske logistike na mnogo bolj kompleksen in predvsem nedeterminističen nivo razvoja in uporabe metodologij. Tako postajata razvoj in uporaba metod za podporo obvladovanja znanja (knowledge management, KM) vedno pomembnejši segment razvoja.

Podjetje Quintelligence je in bo usmerjeno predvsem v razvoj in izvedbo metod in sistemov za pridobivanje, analizo, hranjenje in prenos znanja. S kombiniranjem delnih — problemsko usmerjenih rešitev, gradimo kompleksen in fleksibilen sistem za podporo KM, ki bo predstavljal osnovo globalnega informacijskega centra znanja.

Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.

BRONASTI POKROVITELJ