14. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2019

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.");
   }
}
```

14. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2019

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.

Rešitve bodo objavljene na `http://rtk.ijs.si/`.

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 letovišče). 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 praviloma.

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.

14. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2019

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.

Rešitve bodo objavljene na `http://rtk.ijs.si/`.

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.

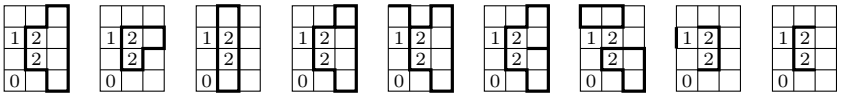
3. Ograje

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.

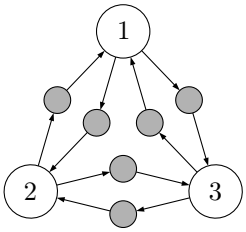
Predpostavimo lahko, da se v besedilu nahaja vsaj pet besed „in“ ali „ter“ (npr.: „... in ... in ... ter ... in ... ter ...“). Lahko tudi predpostavimo, da vrstice niso daljše od 100 znakov. Število vrstic dokumenta ni omejeno. Da ne zapletamo po nepotrebnem, lahko predpostaviš, da besede med seboj loči 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...

(*Nadaljevanje na naslednji strani.*)

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.

14. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2019

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. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku in na ocenjevalnem strežniku), 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 ti lahko prinese 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-ju:

```
#include <stdio.h>
int main()
{
  int i, j; scanf("%d %d", &i, &j);
  printf("%d\n", 10 * (i + j));
  return 0;
}
```

- V C++:

```
#include <iostream>
using namespace std;
int main()
{
  int i, j; cin >> i >> j;
  cout << 10 * (i + j) << '\n';
}
```

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

- V pythonu:

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

- 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));
  }
}
```

14. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2019

NALOGE ZA TRETJO SKUPINO

Rešitve bodo objavljene na `http://rtk.ijs.si/`.

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 nek 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.

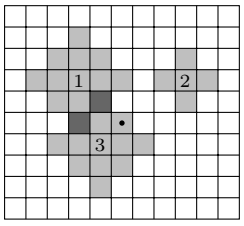
Primer vhoda:

```
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 nek 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 nek 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.

14. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2019

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 prograh 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 najprej 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 = ""
while not NaKoncu():
  # Preberimo naslednjo vrstico, prejšnjo pa si začasno še zapomnimo.
  prejsnja = vrstica; vrstica = Vrstica()
  # 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};
  Z1 := do − od − vsota L[od, ..., do];
  (* Z1 je število zamikov v eno smer — desno, če je i < zadnji, sicer pa levo. *)
  Z2 := k − 1 − Z1;              (* število zamikov v drugo smer *)
  r := r + min{Z1, Z2} + 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 r 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 pogledamo, 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.[1]

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

[1] 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.

hitro rešimo vse testne primere na našem tekmovanju.[2]

```
#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});
  }

  // 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 };
      }
```

[2]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.

```
        // 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 zasukamo 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).[3] 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 $(g - k \cdot c) \bmod 10$.[4] 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 dobimo $D_t = \{(a+b) \bmod 10 \ : \ a \in D_{t-1}, b \in D\}$.

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,

[3] 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).

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

da smo našli najboljšo rešitev: račun dobrodelne organizacije moramo vpisati v $m - t$ največjih nakazil na seznamu. Seznam lahko preprosto uredimo po zneskih 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}.

Naloge so sestavili: detektorji — Urban Duh; jezero — Primož Gabrijelčič; pekarna — Matija Grabnar in Primož Gabrijelčič; nadzor — Tomaž Hočevar; 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 — Mark Martinec; stolpci in vrstice — Polona Novak; transakcijski računi — Jure Slak; smučarski užitki — Jasna Urbančič; razmazani seznam, anagramska razdalja — Janez Brank.

Vprašanja, pripombe, komentarji, popravki ipd. v zvezi z nalogami in rešitvami so dobrodošli: ⟨*janez@brank.org*⟩.