13. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2018

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.");
   }
}
```

13. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2018

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 (npr. kopiraj in prilepi v Notepad in shrani v datoteko). **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. Collatz++

Collatzevo zaporedje je zaporedje naravnih števil, v katerem je vsak člen izračunan iz prejšnjega po naslednjem pravilu: če je prejšnji člen — recimo mu n — deljiv z 2, je naslednji člen enak $n/2$, sicer pa je naslednji člen enak $3n + 1$. Ustavimo se, ko pridemo do števila 1.

Konkretno zaporedje, ki ga na ta način dobimo, je odvisno od tega, s katerim številom začnemo. Na primer, Collatzevo zaporedje z začetnim členom 15 je:

$$15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.$$

Gojmir je napisal funkcijo, ki poišče takšno število k z območja od vključno a do vključno b, da Collatzevo zaporedje z začetkom k nekje doseže maksimalno vrednost. Npr. na območju od 30 do 50 je takšno število $k = 31$, saj je pripadajoče zaporedje:

$$31, 94, 47, 142, 71, 214, \ldots, 6154, 3077, \mathbf{9232}, 4616, 2308, 1154, \ldots, 5, 16, 8, 4, 2, 1.$$

To zaporedje doseže maksimum 9232. Če izberemo kot začetni člen katerokoli drugo celo število z območja od 30 do 50, tega ne bomo presegli.

Kmalu pa je opazil, da ima njegova funkcija pomanjkljivost, saj $k = 41$ prav tako doseže enak maksimum. Ker je izčrpan in ne more več programirati, potrebuje tvojo pomoč. **Napiši podprogram** oz. funkcijo `PoisciVse(a, b)`, ki poišče *vsa* takšna števila. (Tvoja funkcija lahko rezultat vrne v obliki seznama, tabele, vektorja ali česa podobnega, lahko pa ga izpiše na standardni izhod ali v kakšno datoteko, karkoli ti je lažje.)

Primer: `PoisciVse(30, 50)` mora najti števila 31, 41 in 47 — če se namreč Collatzevo zaporedje začne z enim od teh treh števil, bo sčasoma doseglo vrednost 9232; in te vrednosti ne bo nikoli doseglo (ali preseglo), če se začne s katerimkoli drugim številom od 30 do 50.

2. Alfa Bravo

Palček Godrnjavček je postal vodja tajne službe, ki skrbi za varnost Sneguljčice, njegova naloga pa je, da sprejema podatke agentov na terenu in jim daje navodila, kako naj ukrepajo. Da bi se palčki zaščitili pred napakami pri sporazumevanju, so uvedli fonetično abecedo, ki posamezni črki priredi besedo.

Črka	Beseda	Črka	Beseda	Črka	Beseda
A	`ALFA`	J	`JULIET`	S	`SIERRA`
B	`BRAVO`	K	`KILO`	T	`TANGO`
C	`CHARLIE`	L	`LIMA`	U	`UNIFORM`
D	`DELTA`	M	`MIKE`	V	`VICTOR`
E	`ECHO`	N	`NOVEMBER`	W	`WHISKY`
F	`FOXTROT`	O	`OSCAR`	X	`X-RAY`
G	`GOLF`	P	`PAPA`	Y	`YANKEE`
H	`HOTEL`	Q	`QUEBEC`	Z	`ZULU`
I	`INDIA`	R	`ROMEO`		

Godrnjavček te prosi, da **napišeš program**, ki bo znal vnešeno besedilo odkodirati. Kot vhod torej dobi zaporedje kodnih besed iz gornje tabele (ločene bodo s presledki; če ti je lažje, pa lahko predpostaviš, da je vsaka beseda v svoji vrstici), izpiše pa naj pripadajoče zaporedje črk. Pri tem naj bo odporen tudi na manjše napake pri kodiranju: namesto prave kodne besede se lahko v vhodnih podatkih pojavi taka beseda, ki se od prave razlikuje v največ enem znaku (je pa zagotovo še vedno enako dolga kot prava kodna beseda). Tako se lahko na primer zgodi, da namesto besede `LIMA` dobimo `RIMA` ali `LINA` ali `LQMA` in tako naprej, vse te besede pa ravno tako predstavljajo črko L.

Tvoj program lahko podatke bere s standardnega vhoda in piše na standardni izhod, lahko pa namesto tega bere iz datoteke `vhod.txt` in piše v datoteko `izhod.txt` (karkoli ti je lažje).

Primer: če na vhodu dobimo

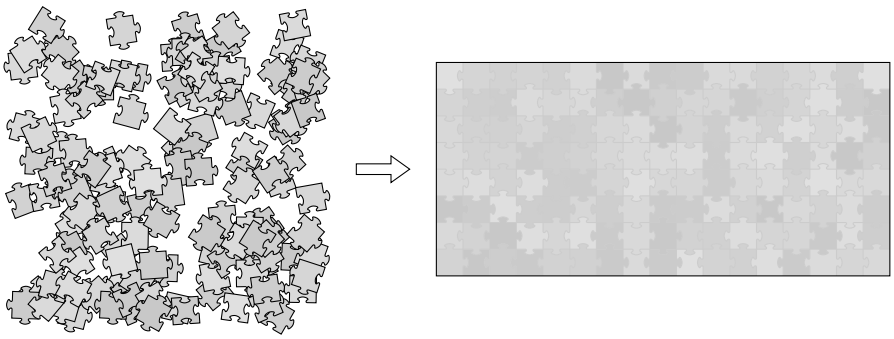
```
SIERRA RODEO ERHO CHARLIF MOVEMBER OSQAR
```

moramo izpisati:

```
SRECNO
```

3. Sestavljanka

Gojmir je na polici opazil škatlo s sestavljanko, ki ima 136 koščkov (glej sliko).



Gojmir ve, da sestavljanko dobimo tako, da sliko razrežemo s (skoraj) vodoravnimi in (skoraj) navpičnimi črtami, tako da dobimo (skoraj) kvadratne koščke. Opazil pa je, da slika, ki jo tvori sestavljanka, v resnici ni kvadratna. Takole čez palec je ocenil, da je razmerje med višino in širino slike približno $3 : 7$.

Koliko koščkov v resnici pride po višini in koliko po širini? Pravokotnik iz 136 koščkov bi lahko bil oblike 1×136 ali 2×68 ali 4×34 ali 8×17 in tako naprej. Vprašanje je torej, pri katerem od teh pravokotnikov je razmerje med višino in širino najbližje tistemu, ki ga je ocenil Gojmir.

Napiši podprogram oz. funkcijo Sestavljanka(steviloKosckov, razmerje), ki bo to izračunal v splošnem primeru, za poljubno število koščkov in poljubno razmerje. Gojmir določi razmerje višine in širine dokaj natančno, vendar ne povsem natančno.

Primer: če pokličemo Sestavljanka(136, 0.428) (to je primer od zgoraj — 0,428 je približno 3/7), mora funkcija vrniti ali izpisati rezultat 8×17 (8 vrstic, 17 stolpcev). Pri tej sestavljanki je razmerje med višino in širino enako $8/17 \approx 0{,}471$, kar je od vseh možnih pravokotnih sestavljank s 136 koščki še najbližje iskanemu razmerju 0,428.

4. Pisalni stroj

Imamo pisalni stroj, ki deluje tako, da se ob vsakem izpisu znaka nujno pomakne za eno mesto v desno. (Eden od teh znakov je tudi presledek, ki sicer ne izpiše ničesar, nas pa ravno tako premakne za eno mesto v desno.) Poleg tega obstaja tudi tipka za pomik na začetek trenutne vrstice (*carriage return*). Drugih možnosti za premik v levo nimamo. Radi bi napisali neko vrstico tako, da bo ponekod na istem mestu več črk ena čez drugo (npr. da kaj podčrtamo z znakom `_`, prekrižamo z znakom `x`, dodamo kakšno naglasno znamenje nad črko in podobno). Recimo, da bo vrstica na koncu dolga n znakov, pri čemer hočemo na mestu i natipkati a_i znakov. **Opiši postopek**, ki iz teh podatkov (torej števil $n, a_1, a_2, \ldots, a_n$) izračuna, kolikšno je najmanjše število pritiskov na tipke (med pritiske šteje tudi pomik na začetek vrstice), ki jih potrebujemo, da natipkamo to vrstico. Pri tem ni pomembno, kje v vrstici se nahajamo na koncu tipkanja. Tvoja rešitev naj bo čim bolj učinkovita, da bo delovala hitro tudi v primerih, ko so n in števila a_i zelo velika.

Primer. Recimo, da hočemo natipkati takšno vrstico dolžine $n = 6$:

`äbç de`

Tabela a bi bila torej v tem primeru takšna:

i	1	2	3	4	5	6
a_i	2	1	3	0	2	1

Minimalno potrebno število pritiskov na tipke je v tem primeru 16. Primerno zaporedje (ni pa edino) tipk je: `a`, `b`, `c`, presledek, `d`, `e`, pomik na začetek, ¨, presledek, vejica, presledek, `_`, pomik na začetek, presledek, presledek in `^`.

5. Brzinomer

V avtomobilu za prikaz trenutne hitrosti vozila skrbi instrument (brzinomer), ki je lahko digitalen (prikazuje številke) ali pa analogen (fizični kazalec instrumenta se pomika po skali in s svojo lego kaže izmerjeno hitrost). A tudi prikazovalniki s kazalcem in skalo so doživeli svojo prenovo in niso več preprosti analogni merilniki neke fizikalne veličine, ampak gre za prikazovalnike, kjer je kazalec pritrjen na os miniaturnega koračnega elektromotorčka, pomike tega pa krmili avtomobilski računalnik glede na hitrost, ki jo izmerijo tipala hitrosti.

Naš prikazovalnik ima skalo v obsegu od 0 km/h do 250 km/h. Koračni motorček lahko pomika kazalec v vsakem koraku le za 1 km/h navzgor ali navzdol, ali pa kazalca ne premakne. Če je kazalec že v najnižji legi, ukaz za pomik navzdol ne stori ničesar (kazalec ostane v najnižji legi, t.j. kaže na 0 km/h na skali). Podobno tudi pri najvišji legi: ukaz za pomik navzgor ne stori ničesar, kazalec ostane pri najvišji legi, t.j. kaže na 250 km/h na skali.

Premik koračnega motorčka za en korak (oz. kazalca, pritrjenega nanj, za 1 km/h) je sicer precej hiter, vendar vseeno zahteva določen čas. Da ne bi avtomobilski računalnik skušal premikati kazalca hitreje, kot to koračni motorček zmore, skrbi ura, ki se proži nekajkrat v sekundi in jamči, da je takrat koračni motorček pripravljen sprejeti nov ukaz za pomik za en korak.

Napiši podprogram oz. funkcijo Premik, ki jo bo ura periodično klicala (približno stokrat v sekundi, točna vrednost intervala ni pomembna). Kot argument bo funkcija ob vsakem klicu prejela celo število med 0 in 300, ki predstavlja trenutno hitrost avtomobila v km/h. Glede na svoje védenje o trenutni legi kazalca brzinomera naj funkcija vrne vrednost -1, $+1$ ali 0, kar bo povzročilo pomik kazalca za en korak navzdol ali navzgor ali pa pomika v tem časovnem intervalu ne bo. Funkcija naj skrbi, da bo lega kazalca brzinomera čim bolj verno sledila dejanski hitrosti avtomobila:

```
function Premik(Hitrost: integer): integer;  { v pascalu }
int Premik(int hitrost);                     /* v C/C++ in podobnih jezikih */
def Premik(hitrost): ...                     # v pythonu; „hitrost“ bo tipa int
```

Upoštevaj, da se lahko hitrost avtomobila med dvema zaporednima klicema tvoje funkcije spremeni tudi za več kot 1 km/h. V tem primeru bo sicer kazalec merilnika lahko za krajši čas zaostajal s pravim prikazom, a mora pravo lego po najboljših močeh čim prej ujeti. Pri hitrostih nad prikazovalnim območjem (t.j. nad 250 km/h) naj kazalec tiči v svoji najvišji legi.

Po potrebi lahko deklariraš poljubne globalne spremenljivke in jim tudi določiš začetne vrednosti.

Upoštevaj tudi, da ob zagonu avtomobilskega računalnika ne vemo točno, kje je obtičal kazalec brzinomera — lahko bi se npr. zgodilo, da je bil motor avtomobila (in računalnik) ugasnjen, še preden se je avtomobil dokončno ustavil na domačem dvorišču. Da zagotovimo znano lego kazalca, si lahko pomagamo z informacijo iz drugega odstavka, ki zagotovi, da z 250 koraki navzdol zagotovo dosežemo spodnjo lego (torej prikaz 0 km/h) ne glede na začetni položaj kazalca. Če ti to dela preglavice, lahko zapišeš, da tvoj program predpostavlja ob zagonu ničelno lego kazalca, a za to boš prikrajšan za polovico točk.

13. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2018

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.

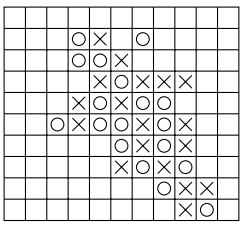
Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi 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 (npr. kopiraj in prilepi v Notepad in shrani v datoteko). **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. Križci in krožci

Imamo igralno ploščo z mrežo m vrstic in n stolpcev. V mrežo na prazna polja rišeta izmenično dva igralca: en igralec postavlja križce, drugi pa krožce. Zmaga tisti, ki prvi postavi 5 svojih znakov skupaj, torej da je 5 enakih znakov zaporedno v isti vrstici, stolpcu ali diagonali.



Primer mreže, v kateri je zmagal igralec, ki postavlja krožce.

Napiši podprogram (funkcijo) ali del programa, ki za dano tabelo $m \times n$ znakov ugotovi, kdo je zmagovalec oziroma, da ni zmagovalca, če ni nihče postavil 5 svojih znakov skupaj v vrsto, stolpec ali diagonalo. Vsak element tabele je eden od znakov `'x'` (križec), `'o'` (krožec) in `' '` (presledek, ki predstavlja prazno polje). Zagotovljeno je, da je zmagovalec lahko največ en igralec (lahko pa tudi ni zmagovalca).

2. Popravljanje testov

Profesor Golob pazi razred n učencev, ki pišejo test iz matematike. Na robu katedra ima označeno mesto, kamor morajo učenci odložiti test, ko zaključijo z delom. Test vedno odložijo na vrh kupa, če ta obstaja. Po tem, ko je na kupu vsaj en test, začne prof. Golob popravljati teste. Nov test vedno vzame z vrha in po popravljanju ga odloži drugam. Primer zaporedja oddajanja in popravljanja bi lahko predstavili z nizom `ABCXDXXRXXHXQX`, kjer `X` pomeni, da je profesor vrhnji test vzel s kupa, druge črke pa, da je neka oseba test odložila na kup. V zgornjem primeru bi prof. Golob po vrsti popravljal teste `CDBRAHQ`.

Učencem je obljubil, da jim bo teste vrnil v enakem vrstnem redu, kot so jih oddajali. Zgolj iz zaporedja `CDBRAHQ` vrstnega reda oddajanja ne more rekonstruirati, verjame pa, da bi lahko pravilno rekonstruiral vrstni red, če bi vedel še, koliko testov je bilo na kupu vsakič, ko je vzel test s kupa. Če bi torej imel zapis oblike `???C?DB?RA?H?Q`, ki za vsak `X` v zgornjem zapisu pove, kateri test je vzel s kupa, znak `?` pa predstavlja, da je nek učenec oddal test, bi lahko rekonstruiral niz `ABCXDXXRXXHXQX`.

Napiši program ali podprogram (funkcijo), ki sprejme niz oblike `???C?DB?RA?H?Q` in vrne ali izpiše niz oblike `ABCXDXXRXXHXQX`. Če vhodni niz ni mogel nastati iz nobenega zaporedja veljavnih oddaj testov, potem to tudi sporoči (vhodni niz štejemo za neveljavnega tudi, če zaporedja oddaj iz njega ni mogoče rekonstruirati). Zaželeno je, da je tvoja rešitev čim bolj učinkovita, torej da bi delovala hitro tudi za velike n (npr. $n = 10^8$).

Nekaj primerov:

Vhod: `?A`
Izhod: `AX`

Vhod: `???`
Izhod: *neveljaven vhod*

Vhod: `???C?DB?RA?H?Q`
Izhod: `ABCXDXXRXXHXQX`

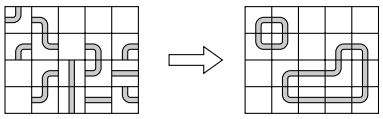
Vhod: `??QWE?`
Izhod: *neveljaven vhod*

Vhod: `??????DC??GBAA??WQBA`
Izhod: `ABAACDXXBGXXXXQWXXXX`

3. Cevi

Na pravokotni mreži dimenzij $v \times s$ (pri čemer sta v in s manjša ali enaka 100) je vsaka celica prazna ali pa vsebuje enega od dveh tipov cevi. Prvi tip je zavita cev  (tip L), ki povezuje dve sosednji stranici celice. Drugi tip je ravna cev  (tip I), ki povezuje nasprotni stranici celice. Z eno potezo lahko zavrtimo celico (oz. cev v njej) za 90 stopinj v poljubno smer. S takimi potezami želimo povezati vse cevi tako, da ne bo nikjer kakšnega odprtega konca cevi. **Opiši** in dobro utemelji **postopek**, ki izračuna najmanjše število potez oz. ugotovi, da to ni mogoče.

Primer: mrežo na spodnji sliki lahko primerno uredimo v 11 potezah.



4. Palačinke

Mihec je navdušen nad palačinkami, zato si jih navadno speče ogromno. Poleg palačink pa ga skoraj še bolj navdušuje prelaganje le-teh. Prelaga jih na poseben način. Pred vsakim prelaganjem si izmisli poljubno naravno število k (med 1 in številom palačink), nato prime kup, sestavljen iz zgornjih k palačink, ga obrne na glavo in nanj (v enakem vrstnem redu, kot so bile do sedaj) postavi ostanek kupa. Tokrat je malo pretiraval v količni palačink, pa še vsaka je malo drugačna, zato te je prosil za pomoč pri simuliranju igre.

Opiši postopek ali napiši program (karkoli ti je lažje), ki simulira potek obračanja palačink, pri čemer palačinke zaradi preprostosti označimo z naravnimi števili od 1 do n. V prvi vrstici standardnega vhoda se nahaja naravno število $n > 1$, ki predstavlja število palačink. Na začetku so palačinke zložene na kupu tako, da je na dnu palačinka številka 1, potem pa si po vrsti sledijo palačinke do vrhnje, ki je označena s številko n. S standardnega vhoda nato do konca beri cela števila (od vključno 0 do vključno n), ki so zapisana vsaka v svoji vrstici. Če je to število večje od 0, potem ustrezno obrni Mihčev kup palačink, če pa je to število enako 0, potem Mihca zanima celotno zaporedje palačink na kupu od spodaj navzdol in ga ustrezno izpiši.

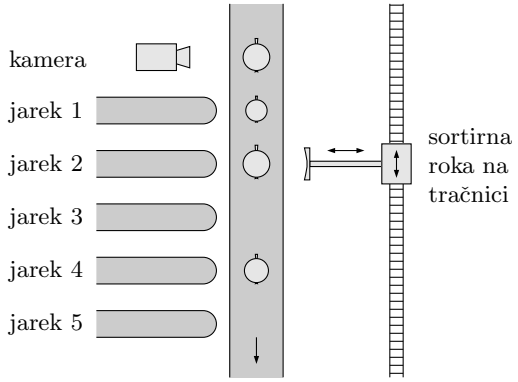
Zaželeno je, da je tvoj postopek čim bolj učinkovit. Pri tem lahko predpostaviš, da so števila k v povprečju majhna v primerjavi z n in da bo zahtev po izpisu celotnega zaporedja malo v primerjavi s številom obračanj.

Primer vhoda:	Pripadajoči izhod:

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

5. Jabolka

Imamo tekoči trak, po katerem prihajajo jabolka. Na začetku traku imamo kamero, ki prepoznava kvaliteto jabolka in ga oceni s številko od 1 do 5. Za kamero je 5 jarkov in sortirna roka, ki lahko jabolko odrine v jarek, pred katerim se nahaja. Vsak jarek pelje v drug zabojček, v katerem se nabirajo sortirana jabolka.



Tekoči trak se premika v diskretnih korakih, vedno po en jarek naprej. V enem takem koraku se jabolka na traku premaknejo za en jarek naprej; tisto jabolko, ki je bilo v prejšnjem koraku pred kamero, se premakne do prvega jarka; pred kamero pa lahko pride naslednje jabolko (ni pa nujno, ker so lahko med jabolki na traku tudi presledki — glej sliko).

Napiši program, ki teče v neskončni zanki in skrbi za to, da sortirna roka odrine vsako jabolko v pravi jarek (torej v tisti jarek, čigar številka je enaka oceni jabolka). Predpostavi, da so že na voljo naslednji podprogrami oz. funkcije, ki jih lahko tvoj program kliče za upravljanje s trakom:

- **int** PremakniTrak() premakne trak za en jarek naprej in vrne celo število z oceno novega jabolka (od 1 do 5), ki je zdaj (po tem premiku) pred kamero. Če pred kamero ni jabolka, funkcija vrne 0. Med dvema zaporednima klicema funkcije PremakniTrak je trak pri miru.
- **void** PremakniRoko(**int** k) premakne sortirno roko pred jarek k (parameter k mora biti od 1 do 5).
- **void** SproziRoko() sproži iztegovanje roke. Če je pred roko jabolko, se bo zvalilo v jarek. Če pred roko ni jabolka, se ne bo zgodilo nič hudega. Podprogram se vrne šele, ko je roka spet v skrčenem položaju in pripravljena na morebitni naslednji premik.

Še deklaracije v drugih programskih jezikih:

```
{ V pascalu: }
function PremakniTrak: integer;
procedure PremakniRoko(k: integer);
procedure SproziRoko;

# V pythonu:
def PremakniTrak(): ...      # vrne int
def PremakniRoko(k): ...     # parameter „k“ mora biti int
def SproziRoko(): ...
```

13. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2018

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.

Po prijavi se bo pred teboj pojavilo namizje XFCE. Nekaj morda uporabnih bližnjic je že na namizju, če pa se ti zdi, da bi ti prav prišlo še kakšno orodje, lahko klikneš na ikono miši v zgornjem levem kotu zaslona. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali python, mi pa jih bomo preverili s prevajalniki Free-Pascal, GNUjevima `gcc` in `g++` 4.7.2 (pozor: ta verzija skoraj v celoti podpira C++11, novejših različic standarda C++ pa ne), prevajalnikom za javo iz JDK 7, s prevajalnikom Mono 2.10 za C# in z interpreterjema za python 2.7 in 3.2. Za delo lahko uporabiš Lazarus (IDE za pascal), `gcc`/`g++` (GNU C/C++ — command line compiler), `javac` (za javo), Eclipse in druga orodja.

Na spletni strani `http://www.putka.si/tasks/slo_tekme/rtk2018/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. 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 več testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. Izjemi sta druga naloga, kjer je testnih primerov 20 in za pravilen odgovor pri posameznem testnem primeru dobiš 5 točk, in peta naloga, kjer je možno dobiti tudi delne točke za delno pravilne odgovore.

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 M točk.

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));
  }
}
```

13. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2018

NALOGE ZA TRETJO SKUPINO

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

1. Buteljke

Skupina prijateljev je sklenila, da si bodo poenostavili življenje in si za rojstni dan ne bodo več podarjali domiselnih daril, saj takšno načrtovanje terja čas in energijo. Od zdaj naprej bo vsak obdarovanec dobil za darilo buteljko vina.

Ker so Gorenjci, so kmalu ugotovili, da lahko buteljko, ki so jo dobili v dar, prihranijo in jo poklonijo ob naslednjem rojstnem dnevu enemu od prijateljev (lahko tudi tistemu, ki jim jo je poklonil). Posamezni človek podari posameznemu prejemniku natanko eno buteljko (če ga sploh obdaruje).

Napiši program, ki izračuna, koliko je minimalno število buteljk, ki jih morajo vsi skupaj kupiti, da bo sistem deloval na dolgi rok, če za vsakega med njimi vemo, kdaj ima rojstni dan in koga vse obdaruje. (Predpostavimo, da z načrtom začnejo ob optimalnem trenutku.)

Da bo prišla rešitev prav tudi skopuhom na drugih planetih, boš pri tej nalogi dobil kot vhodni podatek tudi število dni v letu, recimo mu d. Dnevi so predstavljeni z zaporednimi številkami od 1 do d.

Vhodni podatki: na vhodu dobiš enega ali več testnih primerov. V prvi vrstici je celo število T (zanj velja $1 \leq T \leq 3$). Sledi T testnih primerov, pred vsakim pa je še prazna vrstica.

Posamezni testni primer je opisan takole: v prvi vrstici so tri cela števila, ločena s po enim presledkom: število dni v letu d, skupno število prijateljev n in število vseh obdarovanj k (veljalo bo $1 \leq n \leq d \leq 10^5$ in $1 \leq k \leq 10^6$). Sledi n vrstic s podatki o rojstnih dnevih: i-ta od teh vrstic vsebuje število r_i, to je številka dneva, ko ima oseba i rojstni dan. (Veljalo bo $1 \leq r_i \leq d$. Nikoli se ne bo zgodilo, da bi imela dva človeka rojstni dan na isti dan.) Sledi še k vrstic, od katerih i-ta vsebuje dve različni števili a_i in b_i (zanju velja $1 \leq a_i \leq n$ in $1 \leq b_i \leq n$), ločeni s presledkom; to pomeni, da oseba a_i za rojstni dan obdaruje osebo b_i.

Izhodni podatki: za vsak testni primer iz vhodnih podatkov izpiši po eno vrstico. V tej vrstici izpiši minimalno število buteljk, ki jih morajo (pri tistem testnem primeru) kupiti vsi skupaj; ali pa niz „`Pijanci so med nami!`“ (brez narekovajev), če sistem v tej družbi ne bo deloval.

Primer vhoda:

```
2

100 3 2
53
22
90
1 2
2 3

100 3 3
53
22
90
1 2
2 3
3 1
```

Pripadajoči izhod:

```
Pijanci so med nami!
2
```

Komentar: v vhodnih podatkih sta dva testna primera; v obeh nastopajo trije prijatelji, ki imajo rojstne dneve na 53., 22. in 90. dan v letu. V prvem primeru oseba 1 obdaruje osebo 2, ta pa osebo 3; tu sistem ne deluje. Drugi primer je tak kot prvi, le da poleg tega še oseba 3 obdaruje osebo 1. Zdaj je dovolj, če kupijo dve buteljki in si ju podajajo na veke vekov.

2. Pravokotnik

Podane imamo točke v ravnini. Nekatere so pobarvane modro, nekatere pa rdeče. Zanimajo nas pravokotniki, ki ustrezajo vsem naslednjim pogojem:

- Stranice pravokotnika morajo biti vodoravne ali navpične.
- Nobena modra točka ne sme ležati zunaj pravokotnika (lahko pa ležijo na njegovem robu ali znotraj pravokotnika).
- Nobena rdeča točka ne sme ležati znotraj pravokotnika (lahko pa ležijo na njegovem robu ali zunaj pravokotnika).

Napiši program, ki za dani nabor točk izračuna ploščino največjega takega pravokotnika (ko rečemo „največjega“ pravokotnika, mislimo s tem na tistega z največjo ploščino). Pri tej nalogi bodo testni primeri sestavljeni tako, da vsaj en tak največji pravokotnik gotovo obstaja.

Vhodni podatki: v prvi vrstici sta dve celi števili, m (število modrih točk) in r (število rdečih točk), ločeni s presledkom. Veljalo bo $1 \leq m \leq 10^4$ in $1 \leq r \leq 10^4$. Sledi m vrstic, ki vsebujejo koordinate modrih točk; i-ta od teh vrstic vsebuje dve celi števili x_i in y_i, ločeni s presledkom, to sta koordinati i-te modre točke. Nato sledi še r vrstic, ki na enak način podajajo koordinate rdečih točk. Za koordinate vseh točk velja $|x_i| \leq 10^9$ in $|y_i| \leq 10^9$.

Izhodni podatki: izpiši eno samo celo število, in sicer ploščino največjega pravokotnika, ki ustreza pogojem iz besedila naloge. *Nasvet:* ker je lahko ta ploščina večja od 2^{32}, je koristno pri tej nalogi uporabljati 64-bitne celoštevilske tipe (npr. **long long** v C/C++, **long** v C# in javi, `int64` v pascalu).

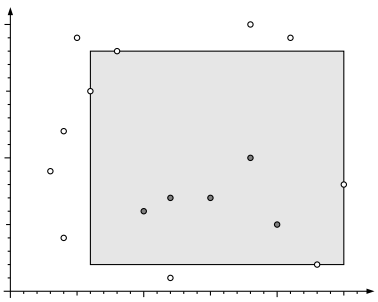
Primer vhoda:

```
5 11
10 6
12 7
18 10
20 5
15 7
25 8
12 1
18 20
3 9
4 4
23 2
21 19
4 12
6 15
8 18
5 19
```

Pripadajoč izhod:

```
304
```

Naslednja slika kaže razpored točk iz primera na levi in največji primerni pravokotnik (s ploščino $19 \cdot 16 = 304$):



3. Tekoči trak

Načrtujemo dolgo tovarniško linijo z n vzporednimi tekočimi trakovi, dolgimi po L metrov. Tekoči trakovi se gibljejo z različnimi hitrostmi, prvi z 1 m/s, sosednji z 2 m/s, naslednji 3 m/s in tako naprej do zadnjega, ki se giblje s hitrostjo n m/s. Izdelek začne svojo pot na začetku prvega traku, po vsakem prevoženem metru (razen po zadnjem) pa lahko izdelek premaknemo na enega izmed sosednjih trakov, tako da se mu hitrost poveča ali zmanjša za 1 m/s. Dodatno imamo na nekaterih dolžinskih odsekih poti hitrostne omejitve, saj morajo izdelek zaznati različni senzorji. **Napiši program**, ki ugotovi, kako naj se izdelek premika po trakovih, da bo prepotoval celotno tovarniško linijo v čim krajšem času, pri čemer mora potovanje začeti in zaključiti na najpočasnejšem tekočem traku (to pomeni, da moramo prvi meter, torej od $x = 0$ do $x = 1$, in zadnji meter, torej od $x = L - 1$ do $x = L$, prevoziti po prvem traku).

Vhodni podatki. V prvi vrstici so podana tri naravna števila, ločena s po enim presledkom: število tekočih trakov n, dolžina linije L in število omejitev m (veljalo bo $1 \leq n \leq 10^9$, $1 \leq L \leq 10^9$ in $1 \leq m \leq 10^3$). Sledi m vrstic, od katerih vsaka vsebuje tri cela števila s_i, e_i in v_i (ločena s po enim presledkom), ki predstavljajo omejitev, da izdelek na območju od točke s_i do e_i (merjeno v metrih od začetka linije) ne sme iti hitreje kot v_i (lahko pa gre točno s hitrostjo v_i). Za vsako od teh omejitev velja $1 \leq v_i \leq n$ in $0 \leq s_i < e_i \leq L$.

V 60 % testnih primerov bo veljalo tudi $n \leq 100$, $L \leq 200$ in $m \leq 200$.

Izhodni podatki: pot izdelka si lahko predstavljamo kot zaporedje trojic oblike l_i d_i t_i, ki pomenijo, da je izdelek prevozil interval od x-koordinate l_i do x-koordinate d_i v celoti po traku t_i. (Seveda mora za vsako tako trojico veljati $0 \leq l_i < d_i \leq L$ in $1 \leq t_i \leq n$.) Tiste izmed teh trojic, pri katerih je $d_i - l_i > 1$, izpiši na izhod, vsako v svojo vrstico, pri čemer naj bodo števila l_i, d_i in t_i ločena s po enim presledkom. Vrstice naj bodo urejene naraščajoče po l_i. Tistih trojic, pri katerih je $d_i = l_i + 1$, pa sploh ne izpisuj.

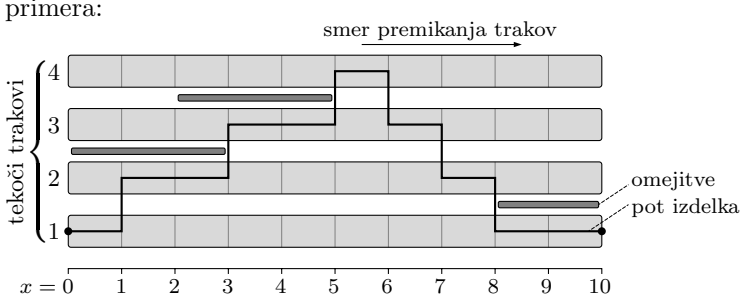
Primer vhoda:

```
4 10 3
0 3 2
8 10 1
2 5 3
```

Pripadajoči izhod:

```
1 3 2
3 5 3
8 10 1
```

Skica gornjega primera:



Komentar: izdelek potuje s hitrostjo 1 m/s na prvem metru, nato potuje dva metra s hitrostjo 2 m/s, nato potuje dva metra s hitrostjo 3 m/s, nato en meter s hitrostjo 4 m/s, nato en meter s hitrostjo 3 m/s, en meter s hitrostjo 2 m/s in dva metra s hitrostjo 1 m/s. Kot zahteva opis naloge, smo na izhod izpisali le tiste od teh korakov, kjer je izdelek potoval vsaj dva metra skupaj po istem traku.

4. Knjige

Na polici imaš od leve proti desni razporejenih več različno visokih knjig. Nekatere boš s police pospravil drugam, da se na njih ne bo nabiral prah. Za preostale knjige na polici pa želiš, da (brez preurejanja) najprej naraščajo po višini, nato pa padajo. Lahko tudi samo naraščajo ali samo padajo. **Napiši program**, ki bo izračunal, kakšno je največje število knjig, ki jih lahko pustiš na polici.

Vhodni podatki: prva vrstica vsebuje število knjig na polici; to je celo število n, za katero velja $1 \leq n \leq 10^6$. V drugi vrstici so naštete s presledki ločene višine knjig, kot si sledijo od leve proti desni. Višine knjig so cela števila med vključno 1 in n. Vse knjige so različnih višin.

V 60 % testnih primerov bo veljalo $n \leq 10^4$, v 40 % primerov bo veljalo $n \leq 100$, v 20 % primerov pa celo $n \leq 15$.

Izhodni podatki: izpiši eno samo celo število, in sicer število knjig, po katerem sprašuje naloga.

Primer vhoda:	Pripadajoči izhod:
`11` `9 4 8 1 11 7 3 6 5 10 2`	`7`

Komentar: v optimalni rešitvi ostanejo na polici od leve proti desni knjige višine 4, 8, 11, 7, 6, 5 in 2.

5. Posredne volitve

V neki državi imajo zelo nenavaden volilni sistem. Imajo n državljanov, ki so oštevilčeni z naravnimi števili od 1 do n. Vsak državljan, recimo u, si izbere nekega državljana, recimo $g(u)$, za svojega *zastopnika*. Lahko izbere tudi samega sebe, torej da je $g(u) = u$.

Na začetku dobi vsak državljan eno kroglico, nato pa si v več *fazah* podajajo kroglice med seboj po naslednjem preprostem pravilu: v posamezni fazi izroči vsak državljan vse kroglice, ki jih je imel ob začetku te faze, svojemu zastopniku.

Število kroglic, ki jih ima državljan u po k fazah, označimo s $f_k(u)$. Na začetku, pred prvo fazo, ima vsak eno kroglico, torej je $f_0(u) = 1$ za vsak u.

Pri opisanem prenašanju kroglic se lahko zgodi, da nek državljan ostane brez kroglic, torej da ima $f_k(u) = 0$. Ni se težko prepričati, da v tem primeru tudi v kasnejših fazah ne bo dobil nobene kroglice več. Zato bomo rekli, da je tak državljan *izpadel* iz volitev. Prej ali slej pa se nujno zgodi, da državljani nehajo izpadati. Naj bo K najmanjše tako število, za katero velja, da v fazah od vključno $K + 1$ naprej ne izpade noben državljan več. (Lahko se zgodi celo, da je $K = 0$.)

Volilna komisija namerava po K fazah prenašanja kroglic ta postopek končati; tiste državljane, ki bodo imeli takrat še kakšno kroglico, bodo razglasili za poslance v parlamentu, vpliv posameznega poslanca pri kasnejših glasovanjih v parlamentu pa bo sorazmeren s tem, koliko kroglic ima na koncu K-te faze. Pomagaj volilni komisiji in ji **napiši program**, ki določi K in izračuna število kroglic po koncu K-te faze, torej $f_K(u)$ za vse državljane u.

Vhodni podatki: v prvi vrstici je število državljanov n; to je celo število z območja $1 \leq n \leq 10^5$. Sledi n vrstic, pri čemer u-ta od njih vsebuje le $g(u)$; to je celo število z območja $1 \leq g(u) \leq n$.

V 40 % testnih primerov bo veljalo tudi $n \leq 1000$.

Izhodni podatki: v prvo vrstico izpiši vrednost K, po kateri sprašuje naloga, v drugo vrstico pa po vrsti izpiši vrednosti $f_K(1), f_K(2), \ldots, f_K(n)$, ločene s po enim presledkom.

Če je v tvoji rešitvi kakšna od vrednosti $f_K(u)$ napačna, vrednost K pa je pravilna, dobiš za tisti testni primer polovico vseh možnih točk.

Primer vhoda:	Pripadajoči izhod:
8 5 3 8 1 8 8 3 1	2 3 0 0 0 2 0 0 3

Komentar: v prvi fazi izpadejo državljani 2, 4, 6 in 7, v drugi fazi pa še državljan 3. Po tistem ne izpade nihče več, tako da je $K = 2$.

13. tekmovanje ACM v znanju računalništva za srednješolce

24. marca 2018

REŠITVE NALOG ZA PRVO SKUPINO

1. Collatz

Pomagamo si lahko z dvema gnezdenima zankama; zunanja gre po vseh možnih vrednostih začetnega člena k od a do b, notranja pa pregleda celotno Collatzevo zaporedje z začetnim členom k. Pri tem računa nove člene po formuli iz besedila naloge, ustavi pa se, ko pride do 1. V spremenljivki kNaj si zapomnimo največji člen tega zaporedja. Ko pridemo do konca, ga primerjamo z največjo vrednostjo drugih doslej pregledanih zaporedij (z začetnimi členi od a do $k-1$), ki jo hranimo v spremenljivki naj; če je nova rešitev enako dobra kot najboljše doslej, dodamo trenutni k v seznam rezultatov (spremenljivka v). Če je nova rešitev celo boljša od najboljše doslej, pa dosedanje rezultate zavržemo (izpraznimo vektor v, v katerem smo jih hranili) in si novo rešitev zapomnimo v naj.

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

vector<int> PoisciVse(int a, int b)
{
  int naj = 0;      /* Največja doslej najdena vrednost. */
  vector<int> v;  /* Začetni členi, pri katerih je bila dosežena. */
  /* Preglejmo vse možne začetne člene od a do b. */
  for (int k = a; k <= b; k++)
  {
    int kNaj = k; /* Največja doslej najdena vrednost v k-jevem zaporedju. */
    /* Preglejmo Collatzevo zaporedje z začetkom pri k. */
    for (int n = k; n > 1; n = (n % 2) ? 3 * n + 1 : n / 2)
      if (n > kNaj) kNaj = n;  /* Če je člen n največji doslej, si ga zapomnimo. */
    /* Če je to boljša rešitev od dosedanjih, dosedanje pobrišimo. */
    if (kNaj > naj) { v.clear(); naj = kNaj; }
    /* Dodajmo k med rešitve, če je vsaj tako dober kot najboljše doslej. */
    if (kNaj == naj) v.push_back(k);
  }
  return v;
}
```

Zapišimo to rešitev še v pythonu:

```
def PoisciVse(a, b):
  naj = 0; v = []
  for k in range(a, b + 1):
    n = kNaj = k
    while n != 1:
      n = n // 2 if n % 2 == 0 else 3 * n + 1
      if n > kNaj: kNaj = n
    if kNaj > naj: v.clear(); naj = kNaj
    if kNaj == naj: v.append(k)
  return v
```

Če nočemo hraniti seznama rešitev (kot je npr. vektor v v gornjem podprogramu), bi morali narediti s k-jem dva prehoda od a do b; v prvem prehodu bi določili naj, v drugem pa izpisovali tiste k-je, pri katerih je kNaj == naj.

2. Alfa bravo

Vhodno besedilo berimo po besedah; za vsako prebrano besedo pojdimo z zanko po vseh črkah abecede in primerjajmo našo besedo s kodo tiste črke. Če sta niza enako dolga in se razlikujeta v največ enem istoležnem znaku, je to prava koda in lahko izpišemo pripadajočo črko. Spodnja rešitev hrani kode v tabeli (globalna spremenljivka kode), za primerjanje dveh nizov pa ima podprogram SeUjema. Ta najprej preveri, če sta niza enako dolga, nato pa primerja istoležne znake in šteje neujemanja. Čim opazi drugo neujemanje, odneha in sporoči, da sta niza preveč različna; če pa pride do konca z največ enim neujemanjem, sporoči, da se ujemata.

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

const string kode[] = { "ALFA", "BRAVO", ..., "ZULU" };

bool SeUjema(const string &s, const string &t)
{
  /* Če sta niza različno dolga, se gotovo ne ujemata. */
  int n = s.length(); if (t.length() != n) return false;
  /* Sicer primerjajmo istoležne znake. */
  for (int i = 0, stNeujemanj = 0; i < n; i++)
    /* Če je to že drugo neujemanje, lahko obupamo. */
    if (s[i] != t[i]) if (++stNeujemanj > 1) return false;
  /* Če smo prišli do konca, se dovolj dobro ujemata. */
  return true;
}

int main()
{
  while (true)
  {
    /* Preberimo naslednjo besedo. */
    string s; cin >> s; if (! cin.good()) break;
    /* Primerjajmo jo z vsemi kodami in izpišimo ustrezni znak. */
    for (int c = 0; c < 26; c++)
      if (SeUjema(s, kode[c])) { cout.put('A' + c); break; }
  }
  return 0;
}
```

Oglejmo si to rešitev še v pythonu:

```
kode = ["ALFA", "BRAVO", ..., "ZULU"]

def SeUjema(s, t):
  n = len(s); stNeujemanj = 0
  if len(t) != n: return False
  for i in range(n):
    if s[i] == t[i]: continue;
    stNeujemanj += 1
    if stNeujemanj > 1: return False
  return True

import sys
for vrstica in sys.stdin:
  for beseda in vrstica.split():
    for c in range(len(kode)):
      if SeUjema(beseda, kode[c]):
        sys.stdout.write(chr(ord('A') + c))
        break
```

Če bi bilo kod veliko, bi jih bilo koristno namesto v tabeli hraniti v drevesu po črkah (*trie*), po katerem bi se potem spuščali glede na črke pravkar prebrane besede (niz s), pri

čemer bi imeli v mislih tudi to, da sme enkrat med tem spuščanjem priti do neujemanja. Lahko bi imeli celo po eno tako drevo za vsako dolžino kode in potem gledali le v tisto drevo, v katerem so kode enako dolge kot niz `s`, saj naloga zagotavlja, da do napak pri dolžini kod ne prihaja.

3. Sestavljanka

Označimo število kosov naše sestavljanke z n, želeno razmerje med višino in širino pa z x. Iščemo torej tako izražavo $n = w \cdot h$ (za naravni števili w in h), pri kateri bo h/w čim bližje x. Lahko gremo v zanki po h in pri vsakem preverimo, ali je res delitelj n-ja; če je, potem tudi izračunamo h/w, ga primerjamo z najboljšim doslej in si ga, če je boljši, zapomnimo:

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

void Sestavljanka(int n, double x)
{
  int hNaj; double dNaj;
  for (int h = 1; h <= n; h++)
  {
    /* Ali je h sploh delitelj n-ja? */
    if (n % h != 0) continue;
    /* Izračunajmo w in razliko |h/w − x|. */
    int w = n / h;
    double d = fabs(h / double(w) − x);
    /* Če je najboljša doslej, si jo zapomnimo. */
    if (h == 1 || d < dNaj) hNaj = h, dNaj = d;
  }
  printf("%d * %d\n", hNaj, n / hNaj);
}
```

Še rešitev v pythonu:

```
def Sestavljanka(n, x):
  for h in range(1, n + 1):
    if n % h != 0: continue
    w = n // h
    d = abs(h / w − x)
    if h == 1 or d < dNaj: hNaj = h; dNaj = d
  return (hNaj, n // hNaj)
```

Časovna zahtevnost te rešitve je $O(n)$, ker v zanki pregleda n različnih vrednosti h-ja in ima z vsako od njih $O(1)$ dela. To bi se dalo na razne načine še izboljšati.

Na primer, naša zanka gre po vseh h-jih in pri vsakem najprej preveri, ali je delitelj n-ja. Če h je delitelj n-ja, se dá n izraziti kot $n = h \cdot w$ za nek celoštevilski w. Iz te enakosti takoj sledi, da ne moreta biti h in w oba večja od $\sqrt{n}$ (oz. oba manjša od $\sqrt{n}$), ker bil potem njun produkt večji od n (oz. manjši od n), ne pa enak n. Delitelji n-ja torej vedno nastopajo v parih, pri čemer je eden $\leq \sqrt{n}$, drugi pa $\geq \sqrt{n}$. Torej bi bilo dovolj, če bi naša zanka po h šla le do $\sqrt{n}$, ne pa do n. Pri vsakem h-ju, za katerega bi opazili, da je delitelj n-ja, pa bi takrat preizkusili še n/h (ki je v tem primeru tudi delitelj n-ja). Tako še vedno pregledamo vse delitelje n-ja, časovna zahtevnost našega postopka pa se z $O(n)$ zmanjša na $O(\sqrt{n})$.

Še bolje bi bilo, če bi n najprej razbili na prafaktorje, potem pa ne bi bilo težko kar našteti vseh njegovih deliteljev: če je $n = \prod_i p_i^{r_i}$, so njegovi delitelji vsa števila oblike $\prod_i p_i^{s_i}$ za $0 \leq s_i \leq r_i$. Vsakega od tako dobljenih deliteljev bi vzeli za h in pogledali, pri katerem je razmerje h/w najbližje x.

Lahko pa se sistematičnega pregledovanja h-jev lotimo malo drugače. Ko je h delitelj n-ja in torej velja $n = h \cdot w$ (za nek celoštevilski w), lahko razmerje h/w zapišemo kot $h/(n/h) = h^2/n$. Pogoj, da mora biti h/w čim bližje x (to zahteva naloga), je torej enak pogoju, naj bo h^2/n čim bližje x, to pa je enakovredno pogoju, naj bo h^2 čim

bližje $x \cdot n$. Namesto da pregledujemo h-je v naraščajočem vrstnem redu od 1 naprej, bi bilo torej bolje, če bi jih pregledovali naraščajoče po vrednosti $E(h) := |h^2 - x \cdot n|$. Tako bi se lahko ustavili že kar pri prvem takem h-ju, za katerega bi se izkazalo, da je delitelj n-ja — tisto je potem najboljša rešitev, ki jo iščemo. Funkcija E seveda doseže svoj minimum, $E = 0$, pri $h_0 := \sqrt{x \cdot n}$, kar pa ni nujno celo število, zato bomo začeli s $h = \lfloor h_0 \rfloor$ in $h = \lceil h_0 \rceil$ in se od tam premikali malo navzdol in malo navzgor, kakor pač nanesejo vrednosti funkcije E.

```
void Sestavljanka2(int n, double x)
{
  /* Funkcija napake, ki jo minimiziramo. */
  auto E = [x, n] (const int h) { return fabs(h * h - x * n); };

  double h0 = sqrt(n * x);  /* Tu doseže E svoj minimum. */
  int h1 = int(floor(h0)); if (h1 < 1) h1 = 1;
  int h2 = h1 + 1; if (h2 > n) h2 = n;
  double e1 = E(h1), e2 = E(h2); int h;

  /* Pregledujmo h-je po naraščajoči vrednosti E(h),
     dokler ne najdemo kakšnega takega h, ki deli n. */
  while (true)
    /* Kateri izmed e1 = E(h1) in e2 = E(h2) je manjši? */
    if (e1 <= e2) {
      if (n % h1 == 0) { h = h1; break; }
      h1--; e1 = E(h1); }
    else {
      if (n % h2 == 0) { h = h2; break; }
      h2++; e2 = E(h2); }

  printf("%d * %d\n", h, n / h);
}
```

Ali v pythonu:

```
import math

def Sestavljanka2(n, x):
  def E(h): return abs(h * h - x * n)
  h0 = math.sqrt(n * x)
  h1 = max(1, math.floor(h0))
  h2 = min(h1 + 1, n)
  while True:
    if E(h1) <= E(h2):
      if n % h1 == 0: return (h1, n // h1)
      h1 -= 1
    else:
      if n % h2 == 0: return (h2, n // h2)
      h2 += 1
```

Pokazati je mogoče, da ima ta rešitev pri $x \leq 1$ v najslabšem primeru časovno zahtevnost $O(\sqrt{n})$; če pa je $x > 1$, je časovna zahtevnost v najslabšem primeru lahko kar $O(n)$. Na misel nam lahko pride, da bi takrat namesto za x raje rešili problem za $1/x$ in nato zamenjali širino in višino tako dobljene rešitve, vendar se hitro vidi, da lahko to pripelje do napačnih rezultatov.[1] To, kar bi morali narediti, je, da bi v takem primeru (torej za $x > 1$) šli z zanko po w-jih namesto po h-jih, pregledovali pa bi jih spet po naraščajočem odstopanju razmerja h/w od iskane vrednosti x, torej $\hat{E}(w) = |h/w - x| = |(n/w)/w - x| = |n/w^2 - x|$. Minimum doseže ta funkcija pri $w_0 := \sqrt{n/x}$, mi pa bi se potem v zanki premikali po celoštevilskih w gor in dol od te začetne vrednosti.

4. Pisalni stroj

Poiščimo največji i, pri katerem je $a_i \geq 1$ — z drugimi besedami, to je najbolj desni stolpec, v katerem je treba sploh kaj natipkati. Vsaj enkrat se bo moral torej naš stroj

[1] Na primer: recimo, da je $n = 11$ in $x = 2$. Možni razbitji na sta le dve, 1×11 (razmerje 1/11, kar je $\approx 0{,}09$) in 11×1 (razmerje 11); iskanemu $x = 2$ je očitno bližja prva od teh dveh možnosti. Toda če rešimo problem za razmerje $1/x$ (torej za 1/2), bomo tudi dobili rešitev 1×11, in če jo potem obrnemo, dobimo 11×1, kar pa *ni* pravilna rešitev prvotnega problema (za $x = 2$).

premakniti vse do stolpca i, da bomo lahko tam kaj natipkali; nobenega razloga pa ni, da bi se premaknili kaj dlje v desno, ker od tam naprej ni treba natipkati ničesar več. Ko se premikamo proti levi od začetka vrstice do stolpca i, lahko spotoma natipkamo še po en znak v vseh ostalih stolpcih $j < i$ (če imajo $a_j \geq 1$; kjer pa je $a_j = 0$, se le premaknemo naprej s tipko za presledek).

Če so bili vsi $a_j \leq 1$, smo s tem že natipkali vse, kar potrebujemo, in lahko končamo. Sicer pa se moramo premakniti v levo, da bomo lahko v kakšnem stolpcu natipkali še kaj; in premakniti v levo se ne moremo drugače kot s skokom na začetek vrstice.

Zdaj lahko poiščemo največji i, pri katerem je $a_i \geq 2$ — to je zdaj zadnji tak stolpec, v katerem je treba še kaj natipkati (ker smo doslej v njem natipkali šele en znak in to očitno ni dovolj). Ko se premikamo od začetka vrstice do tega i, lahko natipkamo še po en znak tudi pri vseh drugih stolpcih $j < i$, ki imajo $a_j \geq 2$.

Če so bili vsi $a_j \geq 2$, smo končali, drugače pa poiščemo največji i, pri katerem je $a_i \geq 3$ in tako naprej.

Označimo z b_k najbolj desni stolpec, v katerem je treba natipkati vsaj k znakov (torej: $b_k := \max\{i : a_i \geq k\}$) in naj bo $m := \max_i a_i$ največje število znakov, ki jih je treba natipkati v kakšnem stolpcu. Dosedanji razmislek nam je povedal, da je skupno število pritiskov na tipke enako

$$s := b_1 + 1 + b_2 + 1 + \ldots + b_{m-1} + 1 + b_m.$$

Pri tem členi +1 v tej vsoti predstavljajo pritiske na tipko za vrnitev v začetek vrstice. Po zadnjem (m-tem) prehodu v desno nam je ni treba pritisniti, saj naloga nič ne pravi o tem, da bi morali končati ravno na začetku vrstice.

Vprašanje je zdaj, kako čim ceneje izračunati to vsoto, sploh ker naloga pravi, da so lahko n in števila a_i (zato pa tudi število m) velika. Neugodno bi bilo, če bi naša rešitev porabila $O(m)$ ali celo $O(n \cdot m)$ časa, npr. ker bi m-krat pregledovali celo tabelo, da določimo $b_1, \ldots, b_m$.

Bolje je, če pregledujemo tabelo od desne proti levi. Ko pri tem prvič naletimo na stolpec z $a_i \geq 1$, je številka tega stolpca (torej i) ravno naš b_1. Ko prvič naletimo na stolpec z $a_i \geq 2$, je njegov i ravno naš b_2 in tako naprej. Vsakič, ko na ta način za nek k prvič izvemo za vrednost b_k, bi jo bilo treba prišteti k neki spremenljivki, v kateri se bo tako sčasoma nabrala vsota s.

Da bo postopek učinkovit, moramo biti pozorni še na to, da se lahko več vrednosti b_k pojavi pri istem stolpcu; na primer, če ima najbolj desni stolpec vrednost $a_n = 4$, bomo imeli $b_1 = b_2 = b_3 = b_4 = n$. Namesto da bi vsakega od teh štirih b_k-jev posebej prištevali k s, je bolje, če upoštevamo, da so pač štirje, in prištejemo k s-ju kar $4 \cdot n$. Tako dobimo naslednji postopek:

```
m := 0;    (* največja doslej videna vrednost a_i *)
s := 0;    (* potrebno število pritiskov na tipke *)
for i := n downto 1:
  if a_i > m:
    (* Zdaj smo izvedeli, da je b_{m+1} = b_{m+2} = ... = b_{a_i} = i.
       Prištejmo te člene k vsoti s. *)
    s := s + i · (a_i − m);
    m := a_i; (* To je zdaj novi največji doslej videni a_i. *)
if m > 0:
  s := s + m − 1; (* Pritiski na tipko za vrnitev na začetek vrstice. *)
return s;
```

Časovna zahtevnost te rešitve je le $O(n)$, česa boljšega od tega pa že ne moremo pričakovati, saj gre $O(n)$ časa že samo za branje vhodnih podatkov.

5. Brzinomer

Koristno je imeti globalno spremenljivko s trenutnim položajem kazalca (v spodnji rešitvi je to kazalec). Ko sistem pokliče našo funkcijo Premik in nam pove novo hitrost, jo primerjajmo s trenutnim položajem kazalca in na podlagi tega določimo premik (+1,

-1 ali 0). Preden se vrnemo iz funkcije, še prištejmo zamik k našemu položaju kazalca, da bomo pripravljeni na naslednji klic.

Naloga pravi, da kazalec ne more iti čez 250 in da bo sistem nadaljnje premike navzgor v tem primeru ignoriral. Zato ni posebne koristi od tega, da v takem primeru sploh vrnemo premik $+1$. Temu se lahko izognemo na primer tako, da če ob klicu dobimo hitrost nad 250, jo pred nadaljnjo obdelavo postavimo na 250.

Paziti moramo še na to, da na začetku ne poznamo pravega položaja kazalca. Naloga zato priporoča, naj prvih 250 korakov zahtevamo premike navzdol, tako da bo števec zagotovo prišel na 0. Spodnja rešitev uporablja zato še eno globalno spremenljivko (zacetek), ki pove, koliko od teh začetnih 250 korakov nam je še ostalo. Vsakič jo zmanjšamo za 1, ko pa pade na 0, začnemo z običajnim delovanjem kazalca. (Šlo pa bi tudi brez te dodatne spremenljivke — lahko bi na primer spremenljivko kazalec inicializirali na -250 in bi jo funkcija Premik počasi povečevala za 1, vračala pa premike navzdol; na običajno delovanje pa bi preklopila, ko bi bil kazalec večji ali enak 0.)

```
enum { Max = 250 };    /* najvišji možni položaj kazalca */
int kazalec = 0;       /* trenutni položaj kazalca */
int zacetek = Max;     /* koliko je še ostalo od začetnega premikanja navzdol */

int Premik(int hitrost)
{
  /* Na začetku se premikamo navzdol, da bo kazalec zagotovo prišel na 0. */
  if (zacetek > 0) { zacetek--; return -1; }

  /* Prevelike hitrosti oklestimo na Max. */
  if (hitrost > Max) hitrost = Max;

  /* Določimo premik, s katerim se bo kazalec približal pravi hitrosti. */
  int premik = (hitrost > kazalec) ? 1 : (hitrost < kazalec) ? -1 : 0;

  kazalec += premik; /* Nova vrednost kazalca po tem premiku. */
  return premik;
}
```

Zapišimo to rešitev še v pythonu:

```
Max = 250
kazalec = 0
zacetek = Max

def Premik(hitrost):
  global kazalec, zacetek
  if zacetek > 0: zacetek -= 1; return -1
  if hitrost > Max: hitrost = Max
  premik = 1 if hitrost > kazalec else -1 if hitrost < kazalec else 0
  kazalec += premik
  return premik
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Križci in krožci

Preprosta rešitev je, da gremo z dvema gnezdenima zankama po vseh poljih mreže in za vsako polje preverimo, ali se tam začne zmagovalna peterica enakih znakov. Če torej trenutno polje ni prazno, ampak je na njem križec ali krožec, moramo preveriti, ali se v kakšni smeri naprej od njega pojavijo še štirje enaki znaki. Za to preverjanje torej uporabimo še dve vgnezdeni zanki: eno po smereh in eno, s katero se premikamo naprej v trenutni smeri. Pri tem premikanju se ustavimo, če pademo čez rob mreže, naletimo na napačen znak ali pa naberemo pet enakih znakov.

Smer premikanja lahko opišemo s parom števil (DX[i], DY[i]), ki povesta, kako se v tisti smeri spreminjata x- in y-koordinata. Potrebujemo štiri smeri: vodoravno, navpično in dve diagonalni.

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

char KdoJeZmagal(const vector<string>& a)
{
  int h = a.size(); if (h == 0) return ' ';
  int w = a[0].size();
  const int DX[] = { 1, 0, 1, 1 }, DY[] = { 0, 1, 1, -1 };
  /* Preverimo vsak možni začetni položaj v tabeli. */
  for (int x = 0; x < w; x++) for (int y = 0; y < h; y++)
  {
    int c = a[y][x]; if (c != 'x' && c != 'o') continue;
    /* Na tem mestu je znak c. Ali so v kakšni smeri še štirje taki znaki? */
    for (int smer = 0; smer < 4; smer++)
    {
      /* Z zanko gremo naprej v smeri „smer“, dokler še vidimo znake „c“.
         Števec „d“ pove, koliko smo jih že videli. */
      int d = 1, xx = x, yy = y;
      while (d < 5)
      {
        xx += DX[smer]; yy += DY[smer];
        if (xx < 0 || yy < 0 || xx >= w || yy >= h)
          break; /* Padli smo čez rob tabele. */
        if (a[yy][xx] != c) break; /* Ni pravi znak. */
        d++;
      }
      if (d == 5) return c; /* Če smo našli pet znakov „c“, je ta igralec zmagal. */
    }
  }
  return ' '; /* Če pridemo do sem, ni zmagal nihče. */
}
```

Morebitna drobna izboljšava bi bila, da bi, preden se začnemo iz nekega (x, y) premikati v smeri smer, preverili še, če ni slučajno na sosednjem polju v *nasprotni* smeri že tudi znak c — to bi pomenilo, da kolikor dolgo skupino c-jev bi že našli iz (x, y) v smeri smer, se bo dalo najti še daljšo skupino, ko bomo začeli iz tistega sosednjega polja (in tisto polje bo prej ali slej prišlo na vrsto, saj zunanji dve zanki po x in y sčasoma preizkusita vsa možna začetna polja).

2. Popravljanje testov

Potek dogodkov je najlažje rekonstruirati, če gremo po znakih vhodnega niza od konca proti začetku. Recimo, da poznamo stanje kupa po i-tem vhodnem znaku. Če je i-ti vhodni znak vprašaj (?), to pomeni, da je nek učenec takrat oddal svoj test. Oddal ga je seveda na vrh kupa, torej je to ravno tisti učenec, ki je po tem znaku na vrhu kupa; in pred tem znakom je bil kup tak kot po njem, le brez tega zadnjega testa na vrhu.

Po drugi strani, če je i-ti vhodni znak kaj drugega kot vprašaj, to pomeni, da je profesor vzel ta test z vrha kupa. Pred tem korakom je bil torej kup tak kot po njem, le s tem dodatnim testom na vrhu.

Tako se lahko počasi premikamo od konca niza proti začetku, sproti popravljamo stanje kupa (naša spodnja rešitev ga hrani kar v nizu `kup`) in rekonstruiramo oddaje.

Na začetku tega postopka bi načeloma hoteli vedeti, kakšen je bil kup na koncu vseh korakov iz vhodnega niza. Tega v resnici ne vemo, vemo pa, da so bili takrat na njem le še taki testi, ki jih profesor ni popravil. Zanje nimamo upanja, da bi ugotovili, kateri učenci so jih oddajali. Zato lahko vzamemo, kot da je bil kup na koncu vhodnega niza prazen; nato pa, če bomo med premikanjem nazaj po nizu kdaj naleteli na oddajo (znak `?`), spremenljivka `kup` pa bo kazala, da je bil kup po tej oddaji prazen, bomo vedeli, da gre tu za oddajo takega testa, ki ga profesor ni popravil, zato te oddaje ne moremo rekonstruirati in je vhodni niz neveljaven.

Na koncu našega postopka, torej ko smo prišli na začetek vhodnega niza, moramo preveriti še to, ali je kup takrat prazen. Če ni prazen, to pomeni, da je profesor pobiral s kupa nekakšne teste, ki jih ni nihče oddal, torej je vhodni niz neveljaven.

Spodnji podprogram vhodni niz `s` sprejme po referenci in ga spremeni v izhodnega, vrne pa logično vrednost, ki pove, ali je bil vhodni niz veljaven. (Če ni bil, ostane `s` po vrnitvi iz naše funkcije v nekem delno spremenjenem stanju.)

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

bool Testi(string &s)
{
  string kup;
  for (int i = s.length() - 1; i >= 0; i--)
  {
    /* Na tem mestu velja: po tistem, ko se je zgodilo prvih i + 1 dogodkov (oddaj
       ali popravljanj), je bilo stanje kupa takšno, kot je trenutno v nizu „kup“. */
    if (s[i] == '?')
    {
      /* V tem dogodku je nek učenec oddal svoj test. To je lahko le tisti,
         čigar test je po tem dogodku na vrhu kupa. Če je bil kup po tem
         dogodku prazen, je to nemogoče, torej je vhodni niz neveljaven. */
      if (kup.empty()) return false;
      /* Znak ? v vhodnem nizu zamenjajmo z oznako tega učenca. */
      s[i] = kup.back();
      /* Pred to oddajo je bil kup tak kot po njej, le brez vrhnjega testa. */
      kup.pop_back();
    }
    else
    {
      /* V tem dogodku je profesor pobral test z vrha kupa. Pred njim
         je bil torej kup tak kot po njem, le s tem dodatnim testom na vrhu. */
      kup.push_back(s[i]);
      /* V izhodnem nizu hočemo imeti pobiranja predstavljena z znakom X. */
      s[i] = 'X';
    }
  }
  /* Če kup zdaj ni prazen, to pomeni, da je bil vhodni niz neveljaven
     (profesor je pobiral teste, ki jih ni nihče oddal). */
  return kup.empty();
}
```

3. Cevi

Recimo, da se postavimo v neko celico naše mreže in razmišljamo o tem, kako bi obrnili cev v njej. Če že poznamo stanje zgornje sosede naše celice, nam to predstavlja določeno omejitev za našo celico: naša celica na tisto sosedo s svojim zgornjim robom, tako da, če se v zgornji sosedi en konec cevi konča na tem robu (ki je njen spodnji rob), se mora

tudi v naši trenutni celici en konec cevi končati na tem robu (ki je njen zgornji rob); in po drugi strani, če se v zgornji sosedi na tem robu ne konča noben konec cevi, se tudi v naši trenutni celici ne sme. Drugače namreč cevi ne bi bile pravilno sklenjene, pač pa bi nekje nek konec cevi obvisel v zraku. (Ta razmislek deluje tudi, če zgornje sosede sploh ni, ker naša trenutna celica leži v prvi vrstici; takrat pač dobimo omejitev, da se v naši trenutni celici cev ne sme končati na zgornjem robu).

Podoben razmislek lahko naredimo tudi z levo sosedo, ki nam postavi omejitev glede levega roba naše trenutne celice. Hitro lahko opazimo, da obe omejitvi skupaj že enolično določata, kako mora biti zasukana cev v trenutni celici:

Tip cevi v trenutni celici	Naj se cev navezuje na zgornji oz. levi rob? na nobenega	le zgornjega	le levega	na oba
prazna		×	×	×
tip I	×			×
tip L				

Pri tem križci × označujejo primere, ko je problem nerešljiv — kakorkoli zasukamo trenutno celico, stanje na zgornjem in levem robu ne bo táko, kot ga zahtevata zgornja in leva soseda.

Tako lahko torej tudi za trenutno celico enolično določimo, kako mora biti obrnjena (ali pa vidimo, da je problem nerešljiv). Potem tudi ni težko določiti, koliko korakov potrebujemo, da jo obrnemo v pravo smer; naredimo 0 ali več vrtenj za 90° v levo, razen če vidimo, da bi to zahtevalo tri korake, tedaj pa raje naredimo eno vrtenje za 90° v desno.

Ker se je dosedanji razmislek opiral na to, da za zgornjo in levo sosedo že vemo, kako sta obrnjeni, je koristno mrežo pregledovati sistematično po vrsticah od zgoraj navzdol, v vsaki vrstici pa gremo po celicah od leve proti desni. Tako bomo vedno, preden pridemo do neke celice, že imeli obdelani njeno zgornjo in levo sosedo.

Zapišimo dobljeni postopek še s psevdokodo:

```
n := 0;
for y := 1 to v:
  for x := 1 to s:
    Z := (y > 1) and mreža[x, y − 1] ∈ {  ,  ,  };
    L := (x > 1) and mreža[x − 1, y] ∈ {  ,  ,  };
    (* Z in L povesta, ali na zgornjem oz. levem robu trenutne celice
       cev mora biti ali ne sme biti. *)
    C := vrednost v zgoraj omenjeni tabeli glede na Z, L in tip cevi v celici (x, y);
    if C = ×:
      izpiši, da je problem nerešljiv; return;
    povečaj n za toliko, kolikor vrtenj je treba,
      da trenutna celica pride v stanje C;
    mreža[x, y] := C;
return n;
```

4. Palačinke

Stanje kupa palačink lahko predstavimo s tabelo celih števil, recimo `kup`, v kateri so palačinke navedene od najnižje (`kup[0]`) do najvišje (`kup[n − 1]`). Na začetku inicializiramo elemente tabele preprosto na števila od 1 do n. Ko uporabnik zahteva izpis kupa, moramo iti le z zanko po vrsti čez tabelo in izpisovati vrednosti v njej.

Pri obračanju zgornjih k palačink lahko novo stanje kupa izračunamo v pomožni novi tabeli (recimo `nova`). Zadnjih k elementov stare tabele skopiramo na začetek nove, vendar v obrnjenem vrstnem redu. Nato prvih $n-k$ elementov stare tabele skopiramo na konec nove (v nespremenjenem vrstnem redu). Zdaj lahko vsebino nove tabele vpišemo nazaj v staro ali pa staro zavržemo in odslej namesto nje uporabljamo novo. Oglejmo si primer implementacije te rešitve v C++:

```
#include <cstdio>
#include <vector>
```

```
using namespace std;

int n;
vector<int> kup;

void Obrni(int k)
{
  vector<int> nova{n};
  /* Zgornjih k palačink gre v obrnjenem vrstnem redu na dno novega kupa. */
  for (int i = 0; i < k; i++) nova[i] = kup[n - 1 - i];
  /* Nad njih pride spodnjih n - k palačink prvotnega kupa. */
  for (int i = k; i < n; i++) nova[i] = kup[i - k];
  kup = move(nova);
}

void Izpisi()
{
  for (int i = 0; i < n; i++)
    printf("%d%c", kup[i], i == n - 1 ? '\n' : ' ');
}

int main()
{
  /* Inicializirajmo kup. */
  scanf("%d", &n); kup.resize(n);
  for (int i = 0; i < n; i++) kup[i] = i + 1;
  /* Odgovarjajmo na poizvedbe. */
  for (int k; scanf("%d", &k) == 1; )
    if (k == 0) Izpisi(); else Obrni(k);
  return 0;
}
```

Slabost te rešitve je, da za obračanje k palačink vedno porabimo $O(n)$ časa (in tudi $O(n)$ prostora za pomožno tabelo nova). To je potratno, saj naloga pravi, da je k v povprečju majhen v primerjavi z n.

Boljšo rešitev dobimo, če si tabelo kup predstavljamo kot krožno (*ring buffer*) — najnižja palačinka ni nujno kup[0], ampak kup[z] za nek začetni indeks z, od tam naprej pa si sledijo kup[z + 1], ..., kup[n − 1], kup[0], ..., kup[z − 1]. V splošnem bomo palačinko, ki bi bila po starem na indeksu i, po novem hranili na $(z + i) \bmod n$. (Na to moramo zdaj paziti pri izpisovanju tabele.)

Ta nova interpretacija tabele kup na primer pomeni, da če povečamo z za 1, ne da bi v tabeli kup karkoli spremenili, nam bo tabela po novem predstavljala takšen kup, kot če bi v prejšnjem kupu vzeli najnižjo palačinko in jo premaknili na vrh kupa.

Če torej povečamo z za $n-k$, bo učinek tak, kot da bi vzeli spodnjih $n-k$ palačink in jih premaknili na vrh kupa; preostalih k palačink pa (ki so bile prej na vrhu kupa) pride zdaj na dno. To je že skoraj tisto, kar potrebujemo za obračanje k palačink, kot ga zahteva naša naloga; vse, kar moramo še narediti, je, da obrnemo vrstni red teh k palačink. To nam bo vzelo le $O(k)$ časa, pa tudi pomožne tabele ne potrebujemo:

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

int n, z = 0;
vector<int> kup;

void Obrni(int k)
{
  /* Ciklično zamaknimo tabelo za n - k mest.
     S tem pride zgornjih k palačink na dno kupa. */
  z = (z + n - k) % n;
```

```
  /* Obrnimo vrstni red spodnjih k palačink. */
  for (int i = 0, j = k - 1; i < j; i++, j--)
    swap(kup[(z + i) % n], kup[(z + j) % n]);
}

void Izpisi()
{
  /* Pri izpisu moramo zdaj upoštevati, da je na dnu kupa tista palačinka,
     ki je na indeksu z, od tam pa gremo ciklično po preostanku kupa. */
  for (int i = 0; i < n; i++)
    printf("%d%c", kup[(z + i) % n], i == n - 1 ? '\n' : ' ');
}
```

Funkcija main lahko ostane enaka kot pri prvotni rešitvi, zato je nismo pisali še enkrat.

Še ena rešitev, ki bi tudi porabila le $O(k)$ časa za obračanje palačink, pa je, da namesto tabele (ali vektorja) uporabimo seznam, predstavljen z dvojno verigo členov, povezanih s kazalci (*doubly linked list*). Tu se lahko v zanki sprehajamo od konca seznama nazaj, pobiramo palačinke z njega in jih odlagamo v nek pomožni seznam; ko jih tako naberemo n, pa novi seznam vrinemo na začetek glavnega seznama (kar ustreza temu, da tistih k palačink vrinemo na dno kupa); to vzame le $O(1)$ časa, ker je treba le popraviti nekaj kazalcev med elementi seznamov. Oglejmo si primer takšne rešitve v C++:

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

list<int> kup;

void Obrni(int k)
{
  list<int> novi;

  /* Vzemimo zgornjih k palačink s kupa in jih obrnimo. */
  for (int i = 0; i < k; i++) {
    novi.push_back(kup.back());
    kup.pop_back(); }

  /* Vrinimo jih na dno kupa (v O(1) časa). */
  kup.splice(kup.begin(), novi);
}

void Izpisi()
{
  for (auto i = kup.begin(); i != kup.end(); ++i)
    printf("%s%d", i == kup.begin() ? "" : " ", *i);
  printf("\n");
}

int main()
{
  /* Inicializirajmo kup. */
  int n; scanf("%d", &n);
  for (int i = 0; i < n; i++) kup.push_back(i + 1);

  /* Odgovarjajmo na poizvedbe. */
  for (int k; scanf("%d", &k) == 1; )
    if (k == 0) Izpisi(); else Obrni(k);

  return 0;
}
```

Slabost te rešitve v primerjavi s prejšnjo je, da porabi $O(n)$ dodatnega pomnilnika za hrambo kazalcev med elementi seznama.

Naloga pravi, da bo k v povprečju majhen, zato sta za potrebe našega tekmovanja zadnji dve opisani rešitvi, torej taki, ki porabita za obračanje k palačink $O(k)$ časa,

dovolj dobri. Vseeno pa je zanimivo razmisliti še o rešitvah, ki bi se dobro obnesle tudi pri večjih k.

Spomnimo se naše prve rešitve, ki je celoten kup predstavila s tabelo. Na začetku nastopajo palačinke v tabeli kar po vrsti od 1 do n. Tabelo si lahko predstavljamo kot blok:

$$\langle 1..n \rangle.$$

Po preložitvi k palačink nastaneta dva bloka:

$$\langle n..(n-k+1) \rangle \, \langle 1..(n-k) \rangle.$$

Po naslednji preložitvi, recimo za l, načeloma nastanejo trije bloki. Če je $l \leq n-k$, zadeva ta preložitev le drugega izmed gornjih dveh blokov in dobimo

$$\langle (n-k)..(n-k-l+1) \rangle \, \langle n..(n-k+1) \rangle \, \langle 1..(n-k-l) \rangle.$$

Če pa je $l > n-k$, prizadene ta preložitev oba gornja bloka in dobimo

$$\langle (n-k+1)..(n-k+c) \rangle \, \langle (n-k)..1 \rangle \, \langle n..(n-k+c+1) \rangle \qquad \text{za } c = l-(n-k).$$

Tako lahko nadaljujemo in po vsaki preložitvi se lahko število blokov poveča za 1. Po r preložitvah imamo $r+1$ blokov in zato naslednja preložitev stane $O(r)$ časa. Ko pridemo do $r = \sqrt{n}$, smo vsega skupaj porabili za teh r preložitev že $O(n)$ časa. Zato si lahko zdaj privoščimo porabiti še $O(n)$ časa, da izračunamo novo stanje tabele in si ga zapišemo. Odtlej imamo spet le en blok (par števil pri vsakem bloku zdaj pomeni indeksa v nazadnje izračunano stanje tabele).

Tako nam torej vsaka skupina $\sqrt{n}$ preložitev vzame skupaj $O(n)$ časa, povprečno torej $O(\sqrt{n})$ za vsako preložitev. Ali je to bolje ali slabše od prvotne rešitve, ki je porabila $O(k)$ časa za preložitev dolžine k? To je seveda odvisno od tega, kako velike k smo tam dobivali.

Rešitev z bloki lahko poskusimo še izboljšati. Namesto da imamo bloke v seznamu, jih zložimo v drevo — predstavljajmo si ga kot B-drevo, da bo lažje razmišljati o uravnoteževanju.[2] Bloki se bodo sčasoma drobili, pa recimo, da jih kar mi že na začetku razdrobimo do dolžine 1. Vsak list tako vsebuje eno palačinko p. Vsako vozlišče (tako list kot notranje) vsebuje tudi dolžino d območja, ki ga predstavlja, in zastavico f, ki pove, ali je to območje obrnjeno.

Naj bo $B(v)$ zaporedje palačink, ki ga predstavlja vozlišče v. Definiramo ga takole:

- če je v list: $B(v) := [v.p]$;
- če je v notranje vozlišče z otroki $c_1, \ldots, c_t$:
  - če $v.f =$ **false**: $B(v) := B(c_1) + \ldots + B(c_t)$;
  - če $v.f =$ **true**: $B(v) := \mathsf{reversed}(B(c_1) + \ldots + B(c_t))$.

Kot običajno pri B-drevesu si izberimo neko stopnjo t in vzdržujmo lastnost, da ima vsako notranje vozlišče od t do $2t-1$ otrok (edina izjema je koren, ki jih sme imeti manj). Drevo ima torej globino $O(\log_t n)$.

Razmislimo zdaj o postopku za preložitev k palačink. Iz korena začasno pobrišimo desnih nekaj poddreves, kolikor je še mogoče, ne da bi skupno število palačink v njih preseglo k. Nato v naslednjem najbolj desnem poddrevesu začnemo brisati njegova poddrevesa, spet kolikor je mogoče, ne da bi skupno število palačink v pobrisanih poddrevesih (vključno s tistimi od prej) preseglo k. Tako nadaljujemo navzdol po drevesu, dokler skupno število palačink v pobrisanih poddrevesih ne doseže točno k. V korenih pobrisanih poddreves obrnimo f (iz **false** v **true** in obratno) in nato ta drevesa v obrnjenem vrstnem redu vrinimo na začetek drevesa.

Kakšna je cena te operacije? Ko smo iz nekega vozlišča pobrisali eno ali več poddreves, je to vozlišče zdaj mogoče podhranjeno in bomo mogoče morali premakniti vanj nekaj poddreves iz kakšnega brata ali pa ju celo združiti. To vzame $O(t)$ časa in ker

[2] Razlika v primerjavi z B-drevesom je sicer ta, da imajo pri B-drevesu elementi nekakšne ključe, po katerih so urejeni. Pri nas takih ključev ne bo, vrstni red elementov pa bo pač tak, kakršen je vrstni red palačink na kupu.

se lahko zgodi po enkrat na vsakem nivoju, bo to skupaj $O(t \log_t n)$. Podobno je pri dodajanju; zaradi dodajanja novih otrok je lahko vozlišče prepolno, zato ga je treba razcepiti (ali pa preseliti nekaj poddreves v brata). Tudi to bo skupaj $O(t \log_t n)$. Ker je t konstanta, torej vidimo, da nam obračanje k palačink (za poljuben k) vzame le $O(\log n)$ časa. Izpis kupa pa gre še vedno v $O(n)$ časa, saj moramo le po vrsti obiskati vsa vozlišča drevesa in pri tem izpisovati elemente v listih.

5. Jabolka

Koristno je vzdrževati tabelo z ocenami jabolk na tekočem traku. Če imamo n jarkov (pri naši nalogi je $n = 5$), imejmo tabelo $n + 1$ elementov, tako da prvi element predstavlja jabolko pred kamero, ostali pa jabolka pred posameznimi jarki. Če na nekem mestu jabolka sploh ni, naj bo tisti element tabele enak 0.

Ko premaknemo tekoči trak za en korak naprej, moramo ustrezno premakniti za eno mesto naprej tudi elemente tabele. To je lažje početi od konca tabele proti začetku, tako da si ob premikanju elementov naprej ne povozimo tistih, ki jih še nismo premaknili. Ob vrnitvi iz funkcije PremakniTrak dobimo oceno jabolka, ki je zdaj na novo prišlo pred kamero; to vpišemo v prvo celico naše tabele.

Tako tabela spet vsebuje prave podatke o stanju vseh jabolk na traku; zdaj se lahko v zanki zapeljemo po vseh jarkih in gledamo, ali je pred jarkom ravno takšno jabolko, ki sodi vanj. Če je, premaknimo roko do tistega jarka in jo sprožimo.

```
int main()
{
  enum { n = 5 };    /* Število jarkov. */
  int jabolka[n + 1];  /* Ocene jabolk pred kamero in jarki. */
  for (int k = 0; k <= n; k++) jabolka[k] = 0; /* Na začetku je trak prazen. */
  while (true)
  {
    /* Premaknimo jabolka naprej po tabeli. */
    for (int k = n; k >= 1; k--) jabolka[k] = jabolka[k - 1];
    /* Premaknimo trak in vpišimo v tabelo tisto jabolko, ki je zdaj pred kamero. */
    jabolka[0] = PremakniTrak();
    /* Poglejmo, katera jabolka je treba pahniti v jarke. */
    for (int k = 1; k <= n; k++)
      if (jabolka[k] == k) /* Je pred pravim jarkom? */
      {
        PremakniRoko(k); SproziRoko();
        jabolka[k] = 0; /* To mesto na traku je zdaj prazno. */
      }
  }
  return 0;
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Buteljke

Ob branju vhodnih podatkov štejmo za vsakega človeka, od koliko ljudi prejme buteljko za svoj rojstni dan in koliko ljudem podari buteljko za njihove rojstne dneve. Če se na koncu pri kakšnem človeku število prejetih in podarjenih buteljk ne ujemata, lahko takoj zaključimo, da sistem podajanja buteljk ne bo deloval — če nekdo podari v enem letu drugim več buteljk, kot jih sam prejme na svoj rojstni dan, to pomeni, da jih bo moral vsako leto dokupovati; in podobno, če nekdo na svoj rojstni dan prejme več buteljk, kot jih sam v enem letu podari drugim, to pomeni, da se bo pri njem iz leta v leto nabiralo več buteljk (ki jih bo moral nekdo drug dokupovati).

Če pa vsakdo prejme enako število buteljk, kolikor jih tudi podari, lahko sistem podajanja buteljk deluje v nedogled: ko ima neko prvič po uvedbi sistema rojstni dan, dobi natanko toliko buteljk, kolikor jih bo moral podariti drugim v času med tem rojstnim dnevom in naslednjim. Enako se potem spet zgodi na naslednji rojstni dan in tako naprej. Vprašanje je torej le, koliko buteljk bo moral ta človek podariti drugim še pred svojim prvim rojstnim dnevom — te bo moral namreč kupiti, ker drugače pred svojim prvim rojstnim dnevom še nima nobene buteljke.

Če sistem uvedemo na začetku leta, lahko skupno število kupljenih buteljk določimo zelo preprosto: za vsak par (a, b), kjer oseba a obdaruje osebo b, moramo pogledati, če ima b rojstni dan prej kot a (torej če $r_b < r_a$); če da, potem vemo, da bo moral a to buteljko prvo leto kupiti.

Naloga pravi, da prijatelji z načrtom začnejo „v optimalnem trenutku“, torej moramo razmisliti še o možnosti, da sistema ne uvedemo ravno na začetku leta, ampak nekoč kasneje. Naj bo u tisti človek, čigar rojstni dan nastopi najbolj zgodaj v letu (torej tisti z najmanjšo vrednostjo r_u). Recimo zdaj, da našega sistema ne uvedemo v začetku leta, ampak šele takoj takoj za u-jevim rojstnim dnevom. Kaj se zaradi tega spremeni pri kupovanju buteljk?

- Človek u je prej (ko smo sistem uvedli na začetku leta) imel rojstni dan kot prvi po uvedbi sistema, torej je takrat dobil toliko buteljk, kot jih preostanek leta razdeli drugim, zato mu ni bilo treba kupiti nobene. Po novem pa, ker se sistem uvede tik za njegovim rojstnim dnevom, to pomeni, da bo prve buteljke od drugih dobil šele čisto zadnji, na koncu prvega leta po uvedbi sistema. Vse buteljke, ki jih mora sam podariti drugim, bo moral torej prvo leto kupovati. Število kupljenih buteljk se torej poveča za toliko, kolikor buteljk podari človek u.
- Ko smo sistem uvedli na začetku leta, so morali vsi, ki podarijo u-ju buteljko, le-to prvo leto kupiti, ker je imel on rojstni dan kot prvi po uvedbi sistema in zato do takrat še nihče drug ni prejel nobene buteljke. Po novem pa, ker ima u rojstni dan šele čisto na koncu prvega leta po uvedbi sistema, to pomeni, da imajo vsi ostali svoje rojstne dneve prej in takrat dobi vsak toliko buteljk, kolikor jih bo moral sam podariti drugim do svojega naslednjega rojstnega dneva. Zato po novem nihče več ne kupuje buteljke za u-ja, ampak mu podari eno od tistih, ki jih je pred tem sam dobil za rojstni dan. Število kupljenih buteljk se torej zmanjša za toliko, kolikor buteljk prejme človek u.

Če obe omenjeni spremembi seštejemo in upoštevamo, da u podari enako število buteljk, kolikor jih tudi sam dobi (saj drugače sistem sploh ne bi mogel delovati in bi nad njim že na začetku obupali), lahko zaključimo, da se skupno število kupljenih buteljk sploh nič ne spremeni. Če datum uvedbe sistema počasi pomikamo še naprej po letu, nam enak razmislek pove, da se število kupljenih buteljk tudi kasneje ne bo spreminjalo. V resnici se torej prijateljem ni treba truditi z iskanjem optimalnega trenutka za uvedbo sistema, saj bo število kupljenih buteljk enako ne glede na datum uvedbe sistema.

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

int main()
{
```

```
  int nPrimerov; scanf("%d", &nPrimerov);
  while (nPrimerov-- > 0)
  {
    int n, d, k; scanf("%d %d %d", &d, &n, &k);
    // Preberimo podatke o rojstnih dnevih.
    struct Oseba { int r, dobi = 0, da = 0; };
    vector<Oseba> osebe {n};
    for (int i = 0; i < n; i++) scanf("%d", &osebe[i].r);
    // Preberimo podatke o obdarovanjih in štejmo, koliko buteljk bo treba kupiti.
    int stKupljenih = 0;
    for (int i = 0; i < k; i++) {
      int a, b; scanf("%d %d", &a, &b);
      auto &A = osebe[a - 1], &B = osebe[b - 1]; A.da++; B.dobi++;
      if (B.r < A.r) stKupljenih++; }
    // Preverimo, če vsakdo prejme in podari enako število buteljk.
    for (const auto &O : osebe)
      if (O.da != O.dobi) { stKupljenih = -1; break; }
    // Izpišimo rezultat.
    if (stKupljenih < 0) printf("Pijanci so med nami!\n");
    else printf("%d\n", stKupljenih);
  }
  return 0;
}
```

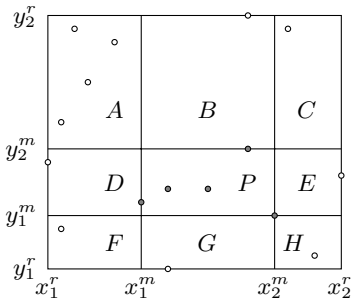
2. Pravokotniki

Za začetek poiščimo najmanjši tak pravokotnik, ki vsebuje vse modre točke (ko rečemo „vsebuje“, mislimo s tem, da ležijo na robu ali v notranjosti); moramo se le sprehoditi po njih in si zapomniti najmanjšo in največjo x-koordinato ter najmanjšo in največjo y-koordinato. Recimo temu pravokotniku $P = [x_1^m, x_2^m] \times [y_1^m, y_2^m]$.

Ker mora tudi tisti pravokotnik, po katerem sprašuje naša naloga, vsebovati vse modre točke, si ga lahko predstavljamo kot nekakšno razširitev pravokotnika P, ki jo dobimo tako, da P-jevo levo stranico premaknemo še malo bolj v levo, zgornjo stranico še malo bolj gor in tako naprej. Vprašanje je zdaj, katere stranice premakniti in kako daleč, da bomo dobili največji primerni pravokotnik.

Na primer, zgornjo stranico lahko dvigujemo le tako daleč, dokler se ne dotakne neke rdeče točke. Višino, na kateri se to zgodi, lahko dobimo tako, da med vsemi rdečimi točkami (x, y), za katere je $y \geq y_2^m$ in $x_1^m < x < x_2^m$, poiščemo najnižjo (tisto z najmanjšim y). Podoben razmislek opravimo tudi pri ostalih treh stranicah in tako dobimo v vsaki smeri skrajni možni položaj stranice našega razširjenega pravokotnika. Recimo, da gre na levi do x_1^r, na desni do x_2^r, spodaj do y_1^r in zgoraj do y_2^r. Vse rdeče točke, ki ležijo zunaj območja $Q = [x_1^r, x_2^r] \times [y_1^r, y_2^r]$, lahko popolnoma ignoriramo, saj pravokotnika zagotovo ne bomo mogli razširiti tako daleč, da bi nas tiste točke kaj ovirale.

Območje Q lahko zdaj v mislih razrežemo pri $x = x_{1,2}^m$ in $y = y_{1,2}^m$; dobimo devet kosov, kot kaže spodnja slika (za primer iz besedila naloge):



Za B že vemo, da ne vsebuje nobene rdeče točke, razen prav na zunanjem (zgornjem) robu. Podoben razmislek velja tudi za D, E in G. Rdeče točke, ki bi nas utegnile pri

raztegovanju pravokotnika P začeti ovirati že prej, preden dosežemo zunanji rob Q-ja, lahko torej ležijo le na vogalnih kosih A, C, F in H.

Spomnimo se, da iščemo največji pravokotnik, ki ne vsebuje nobene rdeče točke. To pa pomeni, da na vsaki od njegovih stranic leži vsaj ena rdeča točka; kajti če na neki stranici ne bi ležala nobena, bi lahko v tisto smer pravokotnik še malo razširili, ne da bi pri tem začel kršiti pogoje, ki jih postavlja naloga. Torej na primer za levo stranico našega iskanega pravokotnika pridejo v poštev le x-koordinate rdečih točk iz kosov A, D in F (tem bomo rekli *leve točke*), za desno stranico pa le x-koordinate rdečih točk iz kosov C, E in H (tem bomo rekli *desne točke*).

Vse možne položaje leve in desne stranice lahko torej preiščemo z dvema gnezdenima zankama, eno po x-koordinatah levih točk in eno po x-koordinatah desnih točk. Toda ko pravokotnik širimo (s premikanjem leve in desne stranice), pride vse več rdečih točk v tak položaj (nad ali pod našim pravokotnikom), da nas bodo omejevale pri premikanju zgornje in spodnje vrstice. Vprašanje je torej, kako pri izbranem položaju leve in desne stranice učinkovito izračunati, kako daleč smemo premakniti zgornjo in spodnjo stranico.

Recimo, da levo stranico fiksiramo in desno stranico počasi premikamo desno. Ko se desna stranica premakne desno mimo neke desne točke $T(x, y)$, to za zgornjo in spodnjo stranico pomeni naslednje: če je $y \geq y_2^m$ (torej če $T \in C$), v bodoče (ko bo desna stranica ležala desno od x) zgornja stranica ne bo smela biti nad y (ker bi sicer točka T prišla v notranjost pravokotnika); in podobno, če je $y \leq y_1^m$ (torej če $T \in H$), v bodoče spodnja stranica ne bo smela biti pod y. Če pa je $y_1^m < y < y_2^m$ (torej če $T \in E$), se desna stranica sploh ne sme premakniti še bolj v desno, saj bi pri tem točka T takoj prišla v notranjost pravokotnika.

S tem razmislekom lahko torej počasi premikamo desno stranico v desno in pri tem sproti spuščamo zgornjo in dvigujemo spodnjo stranico. Pred vsakim premikom moramo še izračunati ploščino pravokotnika in si izmed tako dobljenih ploščin zapomniti največjo.

Ko tako pregledamo vse možne položaje desne stranice pri trenutnem položaju leve stranice, lahko premaknemo levo stranico malo v levo (do naslednje primerne x-koordinate) in s podobnim razmislekom kot prej pri desni stranici tudi zdaj primerno popravimo višino zgornje in spodnje stranice; nato pa z notranjo zanko spet preizkusimo vse možne položaje desne stranice in tako naprej. Ker imamo dve gnezdeni zanki po rdečih točkah, je časovna zahtevnost naše rešitve $O(m + r^2)$.

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

typedef long long llint;
struct Tocka { llint x, y; };

int main()
{
  // Preberimo vhodne podatke.
  int m, r; scanf("%d %d", &m, &r);
  vector<Tocka> mt {m}, rt {r};
  for (int i = 0; i < m; i++) scanf("%lld %lld", &mt[i].x, &mt[i].y);
  for (int i = 0; i < r; i++) scanf("%lld %lld", &rt[i].x, &rt[i].y);

  // Poiščimo najmanjši pravokotnik P, ki pokrije vse modre točke.
  llint xm1 = mt[0].x, ym1 = mt[0].y, xm2 = xm1, ym2 = ym1;
  for (const auto &T : mt) {
    if (T.x < xm1) xm1 = T.x; else if (T.x > xm2) xm2 = T.x;
    if (T.y < ym1) ym1 = T.y; else if (T.y > ym2) ym2 = T.y; }

  // Poglejmo, kako daleč ga lahko raztegnemo gor in dol.
  llint yr1 = ym1 + 1, yr2 = ym2 - 1;
  for (const auto &T : rt)
    if (xm1 < T.x && T.x < xm2) {
      if (T.y <= ym1) if (yr1 > ym1 || T.y > yr1) yr1 = T.y;
      if (T.y >= ym2) if (yr2 < ym2 || T.y < yr2) yr2 = T.y; }

  // Pripravimo si urejen seznam rdečih točk levo in desno od P.
```

```
  sort(rt.begin(), rt.end(), [] (const auto &a, const auto &b) {
    return a.x < b.x || (a.x == b.x && a.y < b.y); });
  vector<Tocka> leve, desne;
  for (const auto &T : rt)
    if (T.x <= xm1) leve.push_back(T);
    else if (T.x >= xm2) desne.push_back(T);

  // Preizkusimo vse možne položaje leve stranice.
  llint maxPloscina = (xm2 - xm1) * (ym2 - ym1);
  llint yl1 = yr1, yl2 = yr2;
  for (int il = leve.size() - 1; il >= 0; il--)
  {
    const auto &L = leve[il];

    // Če postavimo levo stranico na L.x, nas leve točke po y-koordinati
    // omejijo na območje [yl1, yl2].
    llint y1 = yl1, y2 = yl2;

    // Preizkusimo vse možne položaje desne stranice.
    for (const auto &D : desne)
    {
      // Če premaknemo desno stranico na D.x, lahko po y-koordinati
      // pokrijemo območje [y1, y2].
      maxPloscina = max(maxPloscina, (D.x - L.x) * (y2 - y1));

      // Če premaknemo desno stranico desno od D.x, nas začne ovirati tudi točka D.
      if (D.y <= ym1) y1 = max(y1, D.y);
      else if (D.y >= ym2) y2 = min(y2, D.y);
      else break;
    }

    // Če premaknemo levo stranico levo od L.x, nas začne ovirati tudi točka L.
    if (L.y <= ym1) yl1 = max(yl1, L.y);
    else if (L.y >= ym2) yl2 = min(yl2, L.y);
    else break;
  }

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

3. Tekoči trak

Pot izdelka po trakovih lahko predstavimo kot lomljeno črto na ravnini. Če gre črta skozi točko (x, y) za $x \in \{1, \ldots, L\}$ in $y \in \{1, \ldots, n\}$, nam bo to pomenilo, da je izdelek prepotoval x-ti meter na y-tem tekočem traku. Ker se lahko premikamo le po vsakem prevoženem metru in še takrat le za en trak stran od trenutnega, to pomeni, da mora točki (x, y) slediti ena od točk $(x + 1, y)$, $(x + 1, y - 1)$ in $(x + 1, y + 1)$. Naša lomljena črta je torej sestavljena iz odsekov, ki so lahko vodoravni ali pa gredo diagonalno gor ali diagonalno dol.

Ker mora izdelek svojo pot začeti in končati na prvem traku, se mora naša lomljena črta začeti v točki $(1, 1)$ in končati v $(L, 1)$. Ker se od $(1, 1)$ lahko dviguje največ z naklonom 1, se ne bo nikoli povzpela nad premico $y = x$. Podobno, ker se lahko proti $(L, 1)$ spušča največ z naklonom -1, se ne sme nikoli povzpeti nad premico $y = -x + L + 1$. In ker imamo le n trakov, se naša lomljena črta ne sme nikoli povzpeti nad premico $y = n$.

Kako pa na potek naše lomljene črte vplivajo omejitve hitrosti? Recimo, da imamo omejitev (s, e, v).

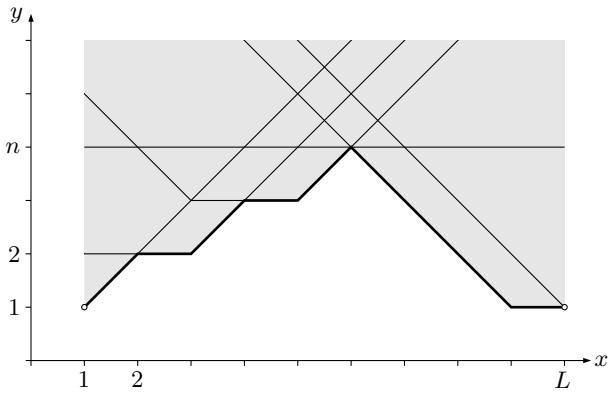
- Ta omejitev pomeni, da smemo po $(s + 1)$-vem, $(s + 2)$-gem, ..., e-tem metru peljati s hitrostjo največ v; naša lomljena črta se torej na intervalu $[s+1, e]$ ne sme povzpeti nad premico $y = v$.
- Ista omejitev pomeni tudi, da smemo iti po s-tem metru s hitrostjo največ $v+1$, po $(s-1)$-vem metru s hitrostjo največ $v+2$ in tako nazaj (če bi namreč šli hitreje kot toliko, potem ne bi mogli pravočasno zmanjšati hitrosti, preden bi prišli do začetka

omejitve). Naša lomljena črta torej se na intervalu $[1, s+1]$ ne sme povzpeti nad premico $y = -x + s + 1 + v$.

- Omenjena omejitev pomeni tudi, da gremo lahko po $(e+1)$-vem metru s hitrostjo največ $v+1$, po $(e+2)$-gem metru s hitrostjo največ $v+2$ in tako naprej (ker hitreje kot toliko ne moremo pospeševati po koncu naše omejitve). Naša lomljena črta se torej na intervalu $[e, L]$ ne sme povzpeti nad premico $y = x + v - e$.

Tako se nam je nabral cel kup daljic (z naklonom -1, 0 ali $+1$), nad katere naša lomljena črta ne sme iti. Ker hočemo, da bi se izdelek premikal čim hitreje, pa se mora naša lomljena črta gibati čim višje; najbolje bo torej, če pri vsakem x teče prav po zgornjem robu dovoljenega območja.

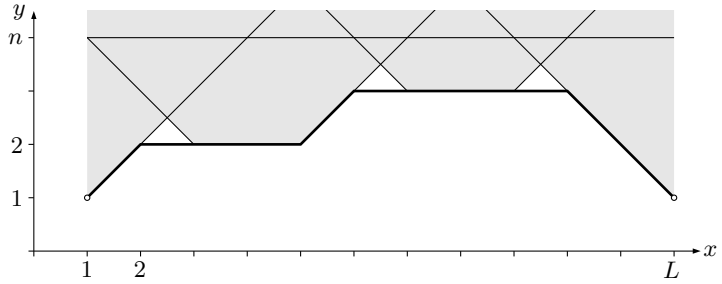
Naslednja slika kaže, kaj nastane po tem razmisleku za primer iz besedila naloge. Začetek in konec naše lomljene črte sta označena s krožcema, prepovedani deli ravnine (tisti, ki ležijo nad kakšno daljico) pa so osenčeni sivo. Če se ves čas gibljemo po zgornjem robu dovoljenega območja, nastane pot, ki je na sliki narisana z debelo črto.



Naloga zahteva, naj izpišemo tiste dele poti, kjer je izdelek potoval po enem traku vsaj dva metra skupaj. Na naši lomljeni črti se tak del poti pokaže kot vodoraven odsek. Tako na primer odsek od (x_1, y) do (x_2, y) pomeni, da je naš izdelek prepotoval na y-tem traku vse metre od vključno x_1-vega do vključno x_2-gega (skupaj je to $x_2 - x_1 + 1$ metrov, kar je ≥ 2, če je $x_2 > x_1$).

Poteku naše lomljene črte ni težko slediti. Začnemo v točki $(1, 1)$, nato pa na vsakem koraku pogledamo, katera izmed daljic, ki so prisotne na trenutnem x, ima pri tem x najnižjo y-koordinato (če je takih več, moramo izbrati tisto z najnižjim naklonom); po tisti se nato premaknemo naprej. Premaknemo se do prvega naslednjega presečišča te daljice s kakšno drugo, če pa takega presečišča ni, gremo pač vse do desnega krajišča daljice. Postopek se ustavi, ko dosežemo točko $(L, 1)$.

Pri računanju presečišča dveh daljic moramo paziti še na naslednjo podrobnost. Če imamo eno naraščajočo in eno padajočo daljico, recimo $y = x + b$ in $y = -x + b'$, je njuno presečišče pri $x = (b' - b)/2$, kar ni nujno celo število. Tudi y-koordinata pri tem x potem ne bi bila celo število. V takem primeru naša lomljena črta prav v to presečišče ne more priti; največ, kar lahko naredimo, je, da gremo po naraščajoči daljici do $\lfloor x \rfloor$, od tam naredimo korak vodoravno do $\lceil x \rceil$ in potem nadaljujemo po padajoči daljici. Naslednja slika kaže več takšnih primerov:



Ta slika ustreza vhodnemu primeru z $n = 4$ trakovi dolžine $L = 12$ in dvema omejitvama hitrosti: $(2, 5, 2)$ in $(6, 9, 3)$.

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

typedef long long llint;

struct Daljica // Daljica, ki jo določata pogoja y = ax + b in xOd ≤ x ≤ xDo.
{
  llint a, b, xOd, xDo;

  Daljica(llint x1, llint y1, llint x2, llint y2) // Inicializira daljico z danima dvema krajiščema.
  {
    if (x2 < x1) swap(x1, x2), swap(y1, y2);
    xOd = x1; xDo = x2;
    if (y1 == y2) a = 0, b = y1;
    else if (y2 - y1 == x2 - x1) a = 1, b = y1 - x1;
    else /* if (y2 - y1 == x1 - x2) */ a = -1, b = y1 + x1;
  }
};

struct Izhod // Pomožni razred za izpis poti izdelka.
{
  llint x = 1, y = 1;                // Trenutni položaj.
  llint xv1 = 0, xv2 = 1, yv = 1;  // Zadnji vodoravni odsek (še ne izpisan).

  // Izpiše zadnji vodoravni odsek (če je dolg vsaj 2 metra).
  void Izpisi() { if (xv2 - xv1 > 1) printf("%lld %lld %lld\n", xv1, xv2, yv); }

  void Dodaj(llint xx, llint yy)      // Doda premik od (x, y) do (xx, yy).
  {
    if (x == xx) return;
    if (y != yy) { x = xx; y = yy; return; } // Poševen odsek.

    // Sicer imamo vodoravni odsek.
    if (x == xv2) { // Podaljšajmo trenutni vodoravni odsek.
      xv2 = xx; x = xx; return; }
    else {
      Izpisi(); // Izpišimo prejšnji vodoravni odsek, če je treba.
      // Začnimo nov vodoravni odsek.
      xv1 = x - 1; xv2 = xx; yv = y; x = xx; }
  }
};

int main()
{
  // Preberimo vhodne podatke.
  int n, L, m;
  scanf("%d %d %d", &n, &L, &m);
  vector<Daljica> ds; // Seznam daljic, ki predstavljajo omejitve.
  ds.emplace_back(1, 1, L, L); // ker moramo prvi meter prevoziti po prvem traku
  ds.emplace_back(L, 1, 1, L); // ker moramo zadnji meter prevoziti po prvem traku
  ds.emplace_back(1, n, L, n); // ker imamo le n trakov

  for (int i = 0; i < m; i++)
  {
    llint s, e, v; scanf("%lld %lld %lld", &s, &e, &v);
    // Za vsako omejitev hitrosti dodajmo ustrezne tri daljice.
    s++; if (s < e) ds.emplace_back(s, v, e, v);
    if (llint d = s - 1; d > 0) ds.emplace_back(s, v, s - d, v + d);
    if (llint d = L - e; d > 0) ds.emplace_back(e, v, e + d, v + d);
  }

  // Sledimo poti po spodnjem robu omejitev.
  Izhod izhod; llint &xOd = izhod.x, &yOd = izhod.y;
  while (xOd < L)
  {
    // Naš trenutni položaj je (xOd, xDo), pri čemer je yOd najnižji y, pri katerem
```

```
    // je za x = xOd prisotna kakšna od naših daljic. Izmed teh daljic bomo v
    // naslednjem koraku sledili tisti, ki ima najnižji naklon.
    int dNaj = -1; llint aNaj;
    for (int i = 0; i < ds.size(); i++)
    {
      const auto &D = ds[i];
      if (xOd < D.xOd || xOd >= D.xDo) continue; // Ne pokriva trenutnega xOd.
      if (D.a * xOd + D.b != yOd) continue;      // Ne gre skozi (xOd, xDo).
      if (dNaj < 0 || D.a < aNaj) dNaj = i, aNaj = D.a;
    }
    // Do kod se lahko gotovo premaknemo? Šli bomo do prvega presečišča s kakšno
    // drugo daljico, če pa takega ni, pa do konca trenutne daljice DD.
    const auto &DD = ds[dNaj];
    llint xDo2 = DD.xDo * 2;
    for (int i = 0; i < ds.size(); i++)
    {
      // Daljice, ki so vzporedne trenutni, preskočimo.
      const auto &D = ds[i];
      if (D.a == DD.a) continue;
      // Poglejmo, kje se D seka z našo DD. Namesto x bomo izračunali
      // njegov dvakratnik, ki je gotovo celo število.
      llint xPres2 = ((D.b - DD.b) * 2) / (DD.a - D.a);
      // To je pravzaprav presečišče njunih nosilk;
      // preverimo še, če res leži tudi na obeh daljicah.
      if (xPres2 > 2 * xOd && xPres2 > 2 * D.xOd && xPres2 <= 2 * D.xDo)
        xDo2 = min(xDo2, xPres2);
    }
    // Izračunajmo novi y in se premaknimo tja.
    llint xDo = xDo2 / 2, yDo = DD.a * xDo + DD.b;
    izhod.Dodaj(xDo, yDo);
    // Če je preseščišče na ne-celih koordinatah, ne moremo čisto do njega,
    // zato naredimo namesto tega vodoraven korak dolžine 1.
    if ((xDo2 % 2) == 1) izhod.Dodaj(++xDo, yDo);
  }
  // Izpišimo še zadnji vodoravni korak.
  izhod.Izpisi(); return 0;
}
```

Ko naša lomljena črta zapusti neko daljico, se nanjo ne more več vrniti. Daljic pa je $O(m)$, zato ima tudi naša lomljena črta le $O(m)$ odsekov. Toliko je zato tudi iteracij zunanje zanke, v vsaki iteraciji pa imamo $O(m)$ dela, da pogledamo, katere izmed ostalih daljic se sekajo s trenutno. Tako vidimo, da je časovna zahtevnost tega postopka $O(m^2)$, kar je za naše testne primere dovolj hitro.

Do še hitrejše rešitve pa pridemo, če za vsak možni naklon (-1, $+1$ in 0) vzdržujemo seznam trenutno aktivnih daljic s tem naklonom (torej takih, ki pokrivajo našo trenutno x-koordinato); v vsakem od teh seznamov naj bodo daljice urejene po konstantnem členu b. Potem je pri računanju presečišč dovolj, če gledamo le presečišča trenutne daljice z najnižjo daljico pri vsakem naklonu (torej tisto z najmanjšim b). Ko se z x premaknemo mimo začetnega krajišča kakšne daljice, jo moramo dodati na ustrezni seznam, ko se premaknemo mimo končnega krajišča, pa jo moramo s tistega seznama pobrisati. (V ta namen si je koristno na začetku pripraviti urejen seznam začetnih in končnih krajišč vseh daljic; to nam vzame $O(m \log m)$ časa.) Da bomo lahko učinkovito dodajali daljice v sezname in jih brisali, je bolje vsakega od teh seznamov v resnici predstaviti s primerno uravnoteženim binarnim drevesom (npr. rdeče-črnim), tako da bo dodajanje, brisanje in iskanje najnižje daljice vzelo le po $O(\log m)$ časa. Časovna zahtevnost celotnega postopka je tako le $O(m \log m)$ namesto $O(m^2)$.

Za konec si oglejmo še veliko krajšo in preprostejšo rešitev, ki sledi poti izdelka meter za metrom in na vsakem koraku preveri, kateri je najhitrejši trak, po katerem lahko nadaljuje svojo pot. Če smo trenutno na traku y, lahko poskusimo iti na $y + 1$, nato

pa to število po potrebi zmanjšamo, če tako zahtevajo omejitve: upoštevati moramo, da je trakov le n; če na naslednjem metru velja kakšna omejitev hitrosti, je ne smemo prekoračiti; pred omejitvami moramo pravočasno začeti zavirati, enako pa tudi proti koncu linije, da bomo zadnji meter prevozili na traku 1. Da vse to upoštevamo, moramo iti v gnezdeni zanki po vseh omejitvah, zato je časovna zahtevnost te rešitve $O(L \cdot m)$. V splošnem je to prepočasi, deluje pa dovolj hitro pri majhnih L. Besedilo naloge pravi, da pri 60 % testnih primerov velja $L \leq 100$, torej bi ta rešitev dobila 60 % točk.

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

int main()
{
  // Preberimo vhodne podatke.
  int n, L, m; scanf("%d %d %d", &n, &L, &m);
  struct Omejitev { int s, e, v; };
  vector<Omejitev> om {m};
  for (auto &O : om) scanf("%d %d %d", &O.s, &O.e, &O.v);

  // Odsimulirajmo pot izdelka.
  for (int xOd = 0, x = 1, y = 1; x <= L; x++)
  {
    // Trenutno smo na traku y, po katerem smo prevozili interval od
    // xOd do x (kjer smo zdaj). Po katerem traku naj nadaljujemo?
    int yy = y + 1;         // Lahko bi poskusili pospešiti.
    yy = min(yy, n);        // Toda ne smemo prekoračiti števila trakov n.
    yy = min(yy, L - x);  // Imeti moramo dovolj prostora za zaviranje na koncu.

    // Preglejmo še omejitve hitrosti.
    for (const auto &O : om)
      if (x < O.s)                      // Imeti moramo dovolj prostora za zaviranje,
        yy = min(yy, O.s - x + O.v); // preden dosežemo omejitev.
      else if (x < O.e)
        yy = min(yy, O.v);              // Smo na območju omejitve.

    // Ali se bomo premaknili na nov trak?
    if (yy != y || x == L) {
      // Prej še izpišimo, kaj smo prevozili na starem traku.
      if (x - xOd > 1) printf("%d %d %d\n", xOd, x, y);
      y = yy; xOd = x; }
  }
  return 0;
}
```

4. Knjige

Označimo naše vhodno zaporedje velikosti knjig z $a_1, a_2, \ldots, a_n$. Naloga sprašuje po najdaljšem takem podzaporedju, ki najprej narašča in nato pada. Razmislimo najprej o malo lažjem problemu: recimo, da nas zanimajo podzaporedja, ki ves čas le naraščajo.

Izberimo si nek konkretni člen našega zaporedja, recimo a_i, in se vprašajmo, kako dolgo je najdaljše naraščajoče podzaporedje, ki se konča s tem členom. Tej dolžini recimo d_i. Ena možnost za d_i je vedno $d_i = 1$, saj je člen a_i sam zase tudi naraščajoče podzaporedje (dolžine 1). Glede daljših podzaporedij pa lahko razmišljamo takole: pred a_i mora biti v takem podzaporedju nek zgodnejši člen a_j, za katerega mora veljati $a_j < a_i$ (da bo podzaporedje res naraščajoče) in seveda $j < i$ (da bo to res podzaporedje prvotnega zaporedja, saj naloga pravi, da vrstnega reda knjig ne smemo spreminjati). Najdaljše naraščajoče podzaporedje s koncem pri a_i torej dobimo tako, da vzamemo najdaljše naraščajoče podzaporedje s koncem pri a_j (to pa je dolgo d_j) in mu dodamo člen a_i; dolžina tako podaljšanega podzaporedja bo torej $d_i = d_j + 1$. Da bomo dobili največji možni d_i, moramo med vsemi možnimi j-ji (tistimi, ki ustrezajo pogojema $j < i$ in $a_j < a_i$) izbrati tistega z največjim d_j. Tako si lahko predstavljamo naslednji postopek:

for $i := 1$ **to** n:
  $d_i := 1$;
  for $j := 1$ **to** $i - 1$:
    if $a_j < a_i$ **then** $d_i := \max\{d_i, d_j + 1\}$;

Slabost te rešitve je, da ima časovno zahtevnost $O(n^2)$, kar je za največje testne primere pri naši nalogi že prepočasi.

Do boljše rešitve nas pripelje naslednji razmislek. Ko dobimo nov člen a_i, se lahko vprašamo: ali je mogoče s tem novim členom podaljšati kakšno naraščajoče podzaporedje dolžine k? (Če je to mogoče, potem vemo, da obstaja naraščajoče podzaporedje dolžine $k + 1$ s koncem pri členu a_i.) To bomo najlažje naredili, če izmed vseh naraščajočih podzaporedij dolžine k vzamemo tisto, pri katerem je zadnji člen najmanjši. Če je celo pri njem zadnji člen $\geq a_i$, to pomeni, da niti tega podzaporedja niti nobenega drugega naraščajočega podzaporedja dolžine k ne bomo mogli podaljšati s členom a_i. Koristno bi bilo torej hraniti vrednost najmanjšega takega člena (v doslej pregledanem delu zaporedja a), pri katerem se konča kakšno naraščajoče podzaporedje dolžine k. Recimo tej vrednosti z_k. Pri $k = 0$ si lahko mislimo $z_k = -\infty$.

Ko dobimo nov člen a_i, moramo torej najti največji tak k, pri katerem je $z_k < a_i$, potem pa vemo, da lahko neko naraščajoče zaporedje dolžine k (z zadnjim členom z_k) podaljšamo s členom a_i, tako da bo $d_i = k + 1$. Ni se težko prepričati, da je zaporedje z_k strogo naraščajoče ($z_0 < z_1 < z_2 < \ldots$), zato lahko največji primerni k poiščemo kar z bisekcijo. Pri tako dobljenem k-ju imamo $z_k < a_i \leq z_{k+1}$. Ker je a_i manjši od (dosedanjega) z_{k+1} in ker smo pravkar ugotovili, da se pri členu a_i konča neko naraščajoče podzaporedje dolžine $k + 1$, lahko torej po novem popravimo z_{k+1} na a_i, tako da bo z_{k+1} tudi po novem res vseboval najmanjši doslej znani člen, pri katerem se konča kakšno naraščajoče podzaporedje dolžine $k + 1$. Tako imamo naslednji postopek:

$z_0 := -\infty$; $K := 0$;
for $i := 1$ **to** n:
  z bisekcijo poišči tak k (od 0 do K), da je $z_k < a_i \leq z_{k+1}$
    (če je treba, si mislimo $z_{K+1} = \infty$);
  $d_i := k + 1$; $z_{k+1} := a_i$;
  if $k = K$ **then** $K := k + 1$;

V spremenljivki K hranimo dolžino najdaljšega doslej znanega naraščajočega podzaporedja, tako da vemo, do kod ima smisel iskati k-je. Ker nam bisekcija vzame vsakič le $O(\log n)$ časa, je časovna zahtevnost tega postopka le še $O(n \log n)$.[3]

Vrnimo se zdaj k prvotni nalogi. Za vsak člen a_i znamo torej izračunati d_i, dolžino najdaljšega naraščajočega podzaporedja, ki se konča s členom a_i. Na enak način lahko izračunamo tudi dolžino najdaljšega padajočega podzaporedja, ki se začne s členom a_i — recimo tej dolžini d'_i. (Uporabimo enak postopek kot prej, le da zaporedje a pregledujemo od konca proti začetku.)

Podzaporedje, po kakršnem sprašuje naloga, mora najprej nekaj časa naraščati in nato nekaj časa padati; vmes je torej nek največji element, recimo a_i, pred tem pa imamo naraščajoče podzaporedje (ki se konča pri a_i), za njim pa padajoče podzaporedje (ki se začne pri a_i). Za prvi del torej vemo, da je lahko dolg največ d_i, za drugi del pa, da je dolg največ d'_i. Podzaporedje, po kakršnem sprašuje naloga, z vrhom pri a_i je torej lahko dolgo največ $d_i + d'_i - 1$ (-1 potrebujemo zato, ker bi sicer člen a_i šteli dvojno). Vse, kar moramo torej še narediti, je, da gremo v zanki po vseh i in pogledamo, katera od tako dobljenih dolžin $d_i + d'_i - 1$ je največja.

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

[3] S problemom najdaljšega naraščajočega podzaporedja smo se na naših tekmovanjih že srečali; gl. rešitev naloge 1992.3.3 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj, 1988–2004* (str. 148–52) in tam navedeno literaturo. Ker so pri naši nalogi velikosti knjig cela števila od 1 do n, bi se dalo rešitev še izboljšati, če bi namesto bisekcije po tabeli z uporabili van Emde Boasovo drevo; časovna zahtevnost bi bila potem le še $O(n \log \log n)$.

```
void NajdaljseNarascajoceZaporedje(const vector<int> &a, vector<int> &d)
{
  vector<int> z; z.push_back(0);
  int n = a.size(), k = 0; d.resize(n);
  for (int i = 0; i < n; i++)
  {
    int ai = a[i];
    // Poiščimo najdaljše tako naraščajoče podzaporedje,
    // ki se konča z vrednostjo, manjšo od a[i].
    int L = 0, R = k + 1;
    while (R - L > 1) {
      // Na tem mestu velja a[L] < a[i] ≤ a[R]. (Za R = k + 1 si mislimo a[R] = ∞.)
      int M = (L + R) / 2;
      if (z[M] > ai) R = M; else L = M; }
    // Najdaljše tako naraščajoče podzaporedje, čigar zadnji element
    // je manjši od a[i], je dolgo L členov; torej ga lahko s členom a[i]
    // podaljšamo v naraščajoče podzaporedje dolžine L + 1 (to pa je enako R).
    if (R > k) z.push_back(ai), k++; else z[R] = ai;
    d[i] = R;
  }
}

int main()
{
  // Preberimo vhodne podatke.
  int n; scanf("%d", &n);
  vector<int> a(n), nnz, npz;
  for (int i = 0; i < n; i++) scanf("%d", &a[i]);
  // Naj bo nnz[i] dolžina tistega najdaljšega naraščajočega podzaporedja,
  // ki se konča s členom a[i].
  NajdaljseNarascajoceZaporedje(a, nnz);
  // Obrnimo a in poženimo isti postopek še enkrat. Tako bo (gledano z
  // vidika prvotnega zaporedja a) nnz[n - 1 - i] dolžina tistega najdaljšega
  // padajočega podzaporedja, ki se začne s členom a[i].
  reverse(a.begin(), a.end());
  NajdaljseNarascajoceZaporedje(a, npz);
  // Poiščimo dolžino najdaljšega zaporedja, po katerem sprašuje naloga.
  int naj = 0;
  for (int i = 0; i < n; i++)
    // Seštejmo dolžino naraščajočega podzaporedja s koncem pri a[i] in
    // padajočega podzaporedja z začetkom pri a[i].
    naj = max(naj, nnz[i] + npz[n - 1 - i]);
  // Izpišimo rezultat.
  printf("%d\n", naj - 1); return 0;
}
```

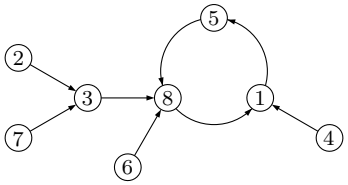
5. Posredne volitve

Vhodne podatke si lahko predstavljamo kot graf, v katerem imamo za vsakega državljana u po eno točko (ki ji tudi recimo u) in še usmerjeno povezavo od u do točke $g(u)$, ki predstavlja njegovega zastopnika. Tako imamo torej n točk in prav toliko povezav; vsaka točka ima natanko eno izhodno povezavo.

Recimo, da začnemo v neki točki u in se premikamo naprej po izhodnih povezavah. Ker ima vsaka točka natanko eno izhodno povezavo, nimamo pri tem nobene izbire, kako nadaljevati pot; možna je ena sama pot z začetkom pri u. To je $u \to g(u) \to g(g(u)) \to g(g(g(u))) \to \ldots$ in tako naprej. To zaporedje se nadaljuje v neskončnost, torej se morajo točke v njem prej ali slej začeti ponavljati, saj ima naš graf le n različnih točk. Čim se neka točka v zaporedju pojavi drugič, to pomeni, da iz tiste točke po nekaj korakih (lahko tudi v enem samem koraku) pridemo nazaj v isto točko, torej je tu na grafu cikel in se bo naša pot odtlej le še v nedogled vrtela naprej po tem ciklu.

Če bi namesto pri u našo pot začeli pri kakšni drugi točki, na primer v, nam zdaj

enak razmislek pove, da bi prej ali slej tudi tista pot prišla do cikla — lahko do istega kot pot iz u, lahko pa do kakšnega drugega. Tako torej vidimo, da je naš graf sestavljen iz ene ali več ločenih komponent, pri čemer vsako komponento tvori en usmerjen cikel in mogoče še eno ali več dreves, ki so pripeta na točke cikla in v katerih so vse povezave usmerjene k ciklu. Primer take komponente kaže naslednja slika (to je graf iz primera v besedilu naloge):



Kaj ta struktura grafa pomeni za postopek podajanja kroglic, ki ga opisuje besedilo naloge? Pot $u \to g(u) \to g(g(u)) \to \ldots$ pravzaprav opisuje, kako se premika med volilci tista kroglica, ki jo je na začetku (pred prvo fazo) imel volilec u. Ker smo videli, da ta pot sčasoma pride v cikel, to pomeni, da bo tudi u-jeva kroglica sčasoma pristala na tem ciklu, kjer si jo bodo potem v nedogled podajali volilci, ki tvorijo ta cikel. Ker se to zgodi pri vsakem u, to pomeni, da bodo tisti volilci, ki ne ležijo na ciklih, sčasoma vsi ostali brez kroglic in bodo izpadli iz volitev.

Tisti volilci pa, ki ležijo na kakšnem ciklu, ne morejo nikoli izpasti: na začetku je imel vsak od jih po eno kroglico in te kroglice se v vsakem koraku ciklično zamaknejo za eno mesto naprej po ciklu, tako da ima tudi v naslednjem koraku še vedno vsak od njih po eno od teh kroglic. Tako nikoli nihče od njih ne ostane brez kroglic (se jim pa lahko seveda število kroglic še poveča, ker bodo sčasoma na cikel prišle kroglice iz dreves, ki so bila pripeta na ta cikel).

Naj bo d_u razdalja med točko u in njej najbližjo točko na kakšnem ciklu; če u že sama leži na ciklu, je $d_u = 0$. To torej pomeni, da bo u-jeva kroglica po natanko d_u fazah prišla na cikel (in se odtlej gibala po ciklu). Maksimum tega po vseh u je torej število faz, ki je potrebno za to, da vse kroglice pridejo na cikle; do takrat volilci še izpadajo, po tistem pa ne več. Število K, po katerem sprašuje naloga, je torej ravno $K = \max_u d_u$.[4]

Ostane nam še izračun funkcije f_K, po kateri tudi sprašuje naloga. Namesto da se za vsak u vprašamo, koliko kroglic ima po K fazah (to je vrednost $f_K(u)$), se je lažje za vsak u vprašati, kateri volilec ima po K fazah tisto kroglico, ki jo je imel na začetku (pred prvo fazo) volilec u. Vemo, da u-jeva kroglica po d_u fazah doseže cikel, odtlej pa se le še premika naprej po tem ciklu, torej naredi še $K - d_u$ korakov naprej po ciklu. Če je cikel dolg c točk (in povezav), je to isto, kot če bi naredila le $(K - d_u) \bmod c$ korakov naprej po ciklu. Tako torej vemo, pri kom je po K fazah u-jeva kroglica. Če ta razmislek ponovimo za vsak u, bomo točno vedeli, koliko kroglic ima kdo.

Da bomo ta postopek lahko izvedli učinkovito, si je koristno za vsak u poleg razdalje do cikla d_u zapomniti tudi številko točke, ki jo iz u po teh d_u korakih dosežemo. Poleg tega je koristno tudi za vsak cikel imeti seznam točk v njem, za vsako točko na ciklu pa podatek o tem, na katerem ciklu leži in katera po vrsti je ta točka v seznamu točk tega cikla. S temi podatki bomo lahko zelo učinkovito prišli najprej od u do najbližje točke cikla, nato pa še skočili za $(K - d_u) \bmod c$ korakov naprej po ciklu. Pazimo tudi na to, da ko za vsako točko u sledimo izhodnim povezavam, da ugotovimo, do katerega cikla se pride iz nje, nam ni treba nujno iti čisto do cikla; ustavimo se lahko, čim naletimo na neko tako točko, za katero smo že nekoč prej ugotovili, do katerega cikla se pride iz nje (in kako daleč je ta cikel). To nam zagotavlja, da se bomo z vsako povezavo grafa ukvarjali le $O(1)$-krat in časovna zahtevnost naše rešitve je le $O(n)$.

[4] O tem se lahko bolj formalno prepričamo takole. Vzemimo nek tak u, za katerega je $d_u = K$. Označimo pot od njega do cikla z $u = u_0 \to u_1 \to \ldots \to u_K$. Zadnja od teh točk, u_K, torej leži na ciklu, njena predhodnica u_{K-1} pa še ne. Za $t = 0, \ldots, K-1$ vidimo, da ima po končanih t fazah volilec u_{K-1} med drugim tisto kroglico, ki jo je imel na začetku (pred prvo fazo) volilec u_{K-1-t}. Po teh fazah torej volilec u_{K-1} še ne izpade. Kaj pa po K-ti fazi? Če bi imel takrat u_{K-1} še kakšno kroglico, to pomeni, da obstaja neka točka v, od katere je pot do u_{K-1} dolga K korakov, toda iz take v bi bila potemtakem pot do najbližjega cikla dolga $K + 1$ korakov, to pa je protislovje, saj smo za K vzeli maksimum dolžine poti do cikla po vseh možnih točkah. Tako torej vidimo, da je res šele K-ta faza (in ne že kakšna zgodnejša) zadnja, v kateri izpade kak volilec.

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

struct Tocka
{
  int g;          // zastopnik
  int f = 0;      // število kroglic po K fazah
  int uc;         // najbližja točka na ciklu
  int d = -1;     // razdalja do uc
  int stCikla = -1, stNaCiklu = -1;  // uc == cikli[stCikla][stNaCiklu]
};

int main()
{
  // Preberimo število točk.
  int n; scanf("%d", &n);

  // Preberimo podatke o zastopnikih.
  vector<Tocka> t {n};
  for (const auto &T : t) { scanf("%d", &T.g); T.g--; }

  // Za vsako točko sledimo poti do cikla.
  vector<vector<int>> cikli;
  vector<int> pot; int K = 0;
  for (int u = 0; u < n; u++)
  {
    if (t[u].d >= 0) continue; // že poznamo

    // Sledimo povezavam u → g(u), dokler ne dosežemo cikla ali pa
    // vsaj neke točke, za katero že poznamo oddaljenost od cikla.
    // Točke na tej poti začasno dodajamo v vektor „pot“ in jim
    // razdaljo d začasno spremenimo iz −1 na −2.
    pot.clear();
    int v = u;
    while (t[v].d == -1) {
      pot.push_back(v); t[v].d = -2;
      v = t[v].g; }

    if (t[v].d == -2)
    {
      // v je prva točka na tej poti, ki smo jo videli dvakrat,
      // torej smo našli nov cikel.
      int i = 0; while (pot[i] != v) i++;
      int stCikla = (int) cikli.size();
      cikli.emplace_back(pot.begin() + i, pot.end());
      pot.resize(i);
      auto &cikel = cikli.back();
      for (i = 0; i < cikel.size(); i++) {
        v = cikel[i]; auto &V = t[v];
        V.uc = v; V.stCikla = stCikla; V.stNaCiklu = i; V.d = 0; }
    }

    // Pot od u do v ni del cikla, pač pa je za v znano, do katerega cikla
    // se pride iz nje; pojdimo nazaj po tej poti in si za vsako
    // točko zapomnimo razdaljo do najbližjega cikla (za v je že znana).
    for (int i = pot.size() - 1; i >= 0; --i)
    {
      v = pot[i]; auto &V = t[v]; const auto &W = t[V.g];
      V.uc = W.uc; V.stCikla = W.stCikla; V.stNaCiklu = W.stNaCiklu;
      V.d = W.d + 1;
    }
    if (t[u].d > K) K = t[u].d;
  }
  // Izračunajmo razporeditev kroglic po K fazah.
  for (const auto &T : t) {
    const auto &cikel = cikli[T.stCikla];
```

```
    t[cikel[(T.stNaCiklu + K − T.d) % cikel.size()]].f++; }
  // Izpišimo rezultate.
  printf("%d\n", K);
  for (int u = 0; u < n; u++) printf("%d%c", t[u].f, u == n − 1 ? '\n' : ' ');
  return 0;
}
```

Viri nalog: Collatz++, sestavljanka, buteljke — Nino Bašić; jabolka — Matija Grabnar; cevi, knjige — Tomaž Hočevar; pravokotnik — Vid Kocijan; palačinke — Filip Koprivec; brzinomer — Mark Martinec; križci in krožci — Polona Novak; popravljanje testov, tekoči trak — Jure Slak; alfa bravo — Jasna Urbančič; pisalni stroj, posredne volitve — Janez Brank.

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