17. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2022

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(f"{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(f"{i}. vrstica: \"{s}\"")
print(f"{i} vrstic, {d} znakov.")
```

```
# 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(f"Skupaj {i} znakov.")
```

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.");
   }
}
```

17. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2022

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. Če oddajaš kaj na papirju, napiši na vsak oddani list svoje ime. Pri delu si lahko pomagaš s prevajalniki in razvojnimi orodji, ki so na voljo na tvojem računalniku, vendar bomo tvoje odgovore v vsakem primeru pregledali in ocenili ročno (ne glede na to, ali si jih oddal prek računalnika ali na papirju), zato manjše napake v sintaksi ali pri klicih funkcij standardne knjižnice niso tako pomembne, kot bi bile na tekmovanjih z avtomatskim ocenjevanjem.

Tekmovanje bo potekalo na strežniku `https://rtk.fri.uni-lj.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla vas bodo čakala na mizi v učilnici. Pri oddaji preko računalnika odpreš dotično nalogo v spletni učilnici in rešitev natipkaš oz. prilepiš v polje za programsko kodo. 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“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v kakšnem drugem urejevalniku na računalniku (Visual Studio Code, Geany, Lazarus) in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

Ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge, uporabi gumb „Shrani spremembe“ in nato klikni na „Nazaj na seznam nalog“, da se vrneš v glavni meni. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na svojem lokalnem računalniku.

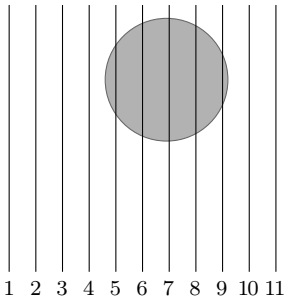
Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno) ali pa vprašaš člane komisije, ki bodo prisotni v učilnicah. Prek zasebnih sporočil bomo pošiljali tudi morebitna pojasnila in popravke, če bi se izkazalo, da so v besedilu nalog kakšne nejasnosti ali napake. Zato med reševanjem redno preverjaj, če so se pojavila kakšna nova zasebna sporočila. Če imaš težave z računalnikom ali s povezavo s spletnim strežnikom za oddajo nalog in komunikacijo s tekmovalno komisijo, se nemudoma obrni na nadzornika v učilnici, ki bo zagotovil drug računalnik. **Če zaradi morebitnih težav pri oddajanju rešitev na strežnik želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici, še preden odideš iz nje.**

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 `https://rtk.ijs.si/`. Predvidoma nekaj dni po tekmovanju bodo tam objavljeni tudi rezultati.

1. Snežinke

Radi bi imeli napravo za merjenje velikosti snežink. V ta namen imamo rešetko vzporednih toplih merilnih žic, razdalja med njimi je en milimeter. Žice so oštevilčene od 1 do 100. Predpostavimo, da so snežinke okrogle, cela snežinka je vedno znotraj območja merilnih žic in vedno se dotakne vsaj ene žice, nato pa spolzi skozi (snežinke se ne nabirajo na žicah). Med dvema snežinkama je vedno dovolj časovnega razmaka, da se prejšnja stopi in spolzi mimo merilnih žic.



Slika kaže prvih enajst od 100 žic. Snežinko, ki jo predstavlja sivi krog, zaznavajo žice 5, 6, 7, 8 in 9.

Na voljo imaš funkcijo Senzor(n), ki vrne vrednost **true**, če n-ta žica zaznava snežinko, sicer vrne **false**. Predpostavimo, da se snežinka dotakne vseh žic hkrati, ko pa se čez čas stopi, je v istem trenutku ne zaznava nobena žica več. Delovanje programa je dovolj hitro, da lahko v času obstoja ali odsotnosti snežinke na merilnih žicah večkrat odčitamo stanje merilnih žic. Snežinka lahko pade na žice kadarkoli, tudi med dvema zaporednima klicema funkcije Senzor.

Napiši program, ki stalno pregleduje stanje merilnih žic in ob vsaki novi snežinki izpiše (natanko enkrat), koliko žic se je dotaknila (in tako izvemo njeno približno velikost). V primeru snežinke z gornje slike mora program izpisati 5.

2. Semafor

Na nekatere semaforje so v zadnjih letih namestili dodatne prikazovalnike, ki prikazujejo čas v sekundah do vklopa zelene luči. Ker želimo čas čakanja izkoristiti za kaj koristnega, smo v avto vgradili inteligentno kamero, ki prepozna številko na prikazovalniku.

Predpostavi, da je za dostop do kamere na voljo funkcija BeriStevec(), ki počaka, da se stanje prikazovalnika spremeni, in poskuša prepoznati številko, ki je po novem prikazana na prikazovalniku. Žal pa se je pri uporabi pokazalo, da kamera pri prepoznavanju ni vedno najbolj natančna (sonce, megla, kót snemanja), zato funkcija BeriStevec ne vrne nujno ene same številke, ampak eno ali več možnih številk, ki jih je kamera prepoznala. Vse te številke so z območja od 0 do 99; prava številka je zagotovo med njimi. Ko prikazovalnik kaže številko 0, vrne tudi BeriStevec le številko 0 in nobene druge. Funkcija BeriStevec je takšne oblike:

```
vector<int> BeriStevec();          // v C++
public static int[] BeriStevec();  // v javi ali C#
def BeriStevec() -> list[int]: ... # v pythonu
```

```
int stevilke[100];   /* v C/C++: funkcija BeriStevec shrani prepoznane številke v globalno */
int BeriStevec();    /* tabelo „stevilke“, kot vrednost funkcije pa vrne število teh številk. */
```

```
var stevilke: array [1..100] of integer;   { v pascalu; deluje enako }
function BeriStevec: integer;              { kot tista v C/C++ }
```

Napiši program, ki s čim manj zaporednimi klici funkcije BeriStevec ugotovi, katera številka je trenutno na prikazovalniku, in jo izpiše.

Primer: spodnja tabela kaže možen potek dogajanja pri več zaporednih klicih. V prvem stolpcu je prava številka s prikazovalnika, v drugem pa je seznam številk, ki bi ga utegnila vrniti funkcija BeriStevec.

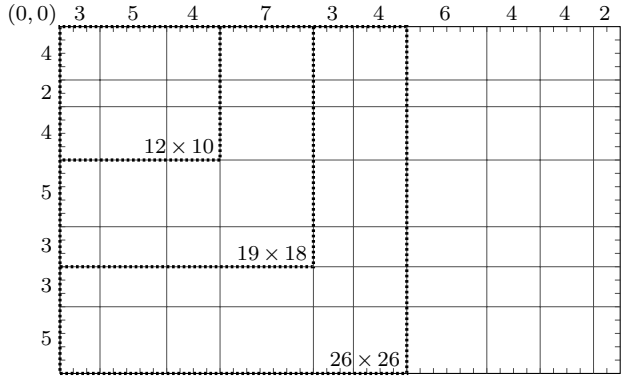
Prikazana številka	Ob klicu BeriStevec vrne
53	$[59, 93, 53, 99]$
52	$[52, 92, 56, 96]$
51	$[51, 57, 91]$
50	$[90, 50, 58, 98]$
49	$[43, 79, 45, 48, 49]$

Po klicih v prikazanem primeru bi se že dalo z gotovostjo zaključiti, da je številka na prikazovalniku res 49 in ne kakšna druga. Tvoj program mora torej takrat izpisati 49.

3. Iskanje kvadrata

Imamo tabelo oz. razpredelnico (*spreadsheet*) z različnimi širinami stolpcev in višinami vrstic. Če vzamemo presek zgornjih nekaj vrstic in levih nekaj stolpcev, lahko dobimo pravokotnike različnih oblik in velikosti (odvisno od tega, koliko vrstic in koliko stolpcev smo uporabili), vsi pa se začnejo v zgornjem levem kotu. Med vsemi takimi pravokotniki želimo izbrati tistega, ki je po obliki čim bližje kvadratu (ali pa je celo res kvadrat). Z drugimi besedami: radi bi, da bi bilo razmerje med dolžino daljše in krajše stranice pravokotnika čim bližje 1.

Opiši postopek (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki kot vhodne podatke dobi zaporedje širin stolpcev in zaporedje višin vrstic ter izračuna širino in višino pravokotnika, o katerem govori prejšnji odstavek. Če je možnih več enako dobrih rešitev, je vseeno, katero izpiše. Širine stolpcev in višine vrstic so cela števila, večja od 0. Poleg tega tudi dobro **utemelji**, zakaj tvoj postopek vrača pravilne rešitve.



Primer: na sliki zgoraj imamo stolpce s širinami $[3, 5, 4, 7, 3, 4, 6, 4, 4, 2]$ in vrstice z višinami $[4, 2, 4, 5, 3, 3, 5]$. Izmed vseh možnih pravokotnikov z začetkom v zgornjem levem kotu so označeni trije (z debelimi črtkanimi črtami), od katerih je najprimernejši tisti z velikostjo 26×26.

4. Neprevidni poeti

Znani poet Gimon Sregorčič se je odločil, da bo v umetnosti pesnjenja izuril nekaj vajencev. Ti seveda niso njegove generacije, temveč so bistveno mlajši in imajo na Gimonovo žalost bistveno bolj moderne poglede na svet in, kar je sploh hudo, poezijo. Najbolj opazna stvar je, da se sploh ne držijo ritma. Celo noč je obupan prebiral njihove pesnitve in beležil, kdo je upošteval ritem in kdo ga ni. Zdaj je že tako utrujen, da tudi sam ne zaznava več poudarjenih in nepoudarjenih zlogov in potrebuje pomoč pri ugotavljanju, katere pesmi se držijo ritma.

V besedilu pesmi so nekateri zlogi naglašeni, drugi pa ne. Zaporedje naglašenih in nenaglašenih zlogov tvori ritem. V nalogi so naglašeni samoglasniki označeni z velikimi črkami, vse ostale črke pa so male. Sklop črk je zlog samo v primeru, da ima natanko en samoglasnik (*a*, *e*, *i*, *o* ali *u*) (v besedi „stržen“ je torej po našem štetju en zlog, zlogotvornega *r* ne upoštevamo). **Napiši program**, ki za dano pesem izpiše, ali je v vseh verzih pesmi isti ritem in če da, kakšen je. Tvoj program lahko prebere pesem s standardnega vhoda ali pa iz datoteke `vhod.txt` (kar ti je lažje). Predpostavi, da posamezne vrstice niso daljše od 100 znakov.

Primer vhoda:

```
od nEkdaj lepE so ljubljAnke slovEle
a lEpse od Urske bilO ni nobEne
nobEne ocEm bilo bOlj zazelEne
ne spOmnem se bEsedilA naprej sOri
```

Pripadajoči izhod:

```
DA
U-UU-UU-UU-U
```

Komentar: druga vrstica je ritem, `U` predstavlja nepoudarjen zlog, `-` pa poudarjen zlog.

Še en primer vhoda:

```
cuj vlAka zvIzg z vetrOvi gnAn
med mrAk oblAkov in poljAn
narascajOc, pojemajOc
takO moj klIc gre v nOc polnOc
```

Pripadajoči izhod:

```
NE
```

5. Stonoge

Malokdo ve, da je Alfred Hitchcock pred svojo uspešno grozljivko *Ptiči* posnel tudi *Stonoge*, ki pa iz različnih razlogov, med drugim zaradi izjemno nizkega filmskega proračuna, niso postale uspešnica. Ker stonoge niso pretirano ubogljiva bitja, so v filmu uporabljali izključno umetne stonoge, ki pa niso bile izdelane preveč prepričljivo. Za primer podajmo eno prepričljivo in eno očitno umetno stonogo:

```
.........\||||||\/\///||/............
.........##############>............
........./||||||/\/\/\||\............
....................................
...................\|||/||/.........
...................<######..........
..................../|||\||\.........
```

Trup stonoge torej tvori eden ali več zaporednih znakov „`#`“ (vsi v eni vrstici), glavo pa predstavlja bodisi znak „`<`“ tik levo ob trupu ali pa znak „`>`“ tik desno ob trupu. (V primeru zgoraj ima gornja stonoga glavo na desnem koncu, spodnja pa na levem.) Stonoge so prepričljive, če so vsi njihovi pari nog simetrični in če sta prvi in zadnji par nog usmerjena k trupu. V primeru zgoraj prva stonoga ni prepričljiva, ker peti par nog (gledano od spredaj) ni simetričen (leva noga kaže naprej, desna pa nazaj), druga stonoga pa je prepričljiva. V spodnjem primeru pa nobena stonoga ni prepričljiva, ker se prvi in/ali zadnji par nog pri nobeni ne drži trupa:

```
.............................////............
..||\//.............|\\||/...##>.....//\\|..
...###>.............<####...\\\\......###>..
..||/\\..../|||/....|//||\..........\\//|..
...........<###..............................
...........\|||\..............................
........................................
```

Napiši program, ki iz položaja stonog v nekem trenutku filma ugotovi, koliko izmed njih je prepričljivih in koliko očitno umetnih. Stonoge se nahajajo na polju iz n vrstic in m stolpcev (n in m sta največ 100). Polje je sestavljeno zgolj iz znakov „`\`“, „`|`“, „`/`“, „`<`“, „`>`“, „`#`“ in „`.`“. Nobeni dve stonogi se ne prekrivata ali dotikata in nobena stonoga ni v kadru le delno. Predpostaviš lahko, da je z glavami stonog vse v redu in ti ni treba preverjati, ali ima res vsaka stonoga glavo na natanko enem koncu. Podrobnosti tega, v kakšni obliki tvoj program dobi ali prebere vhodne podatke, si izberi sam(a) in jih v svoji rešitvi tudi opiši.

17. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2022

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. Če oddajaš kaj na papirju, napiši na vsak oddani list svoje ime. Pri delu si lahko pomagaš s prevajalniki in razvojnimi orodji, ki so na voljo na tvojem računalniku, vendar bomo tvoje odgovore v vsakem primeru pregledali in ocenili ročno (ne glede na to, ali si jih oddal prek računalnika ali na papirju), zato manjše napake v sintaksi ali pri klicih funkcij standardne knjižnice niso tako pomembne, kot bi bile na tekmovanjih z avtomatskim ocenjevanjem.

Tekmovanje bo potekalo na strežniku `https://rtk.fri.uni-lj.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla vas bodo čakala na mizi v učilnici. Pri oddaji preko računalnika odpreš dotično nalogo v spletni učilnici in rešitev natipkaš oz. prilepiš v polje za programsko kodo. 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“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v kakšnem drugem urejevalniku na računalniku (Visual Studio Code, Geany, Lazarus) in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

Ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge, uporabi gumb „Shrani spremembe“ in nato klikni na „Nazaj na seznam nalog“, da se vrneš v glavni meni. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na svojem lokalnem računalniku.

Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno) ali pa vprašaš člane komisije, ki bodo prisotni v učilnicah. Prek zasebnih sporočil bomo pošiljali tudi morebitna pojasnila in popravke, če bi se izkazalo, da so v besedilu nalog kakšne nejasnosti ali napake. Zato med reševanjem redno preverjaj, če so se pojavila kakšna nova zasebna sporočila. Če imaš težave z računalnikom ali s povezavo s spletnim strežnikom za oddajo nalog in komunikacijo s tekmovalno komisijo, se nemudoma obrni na nadzornika v učilnici, ki bo zagotovil drug računalnik. **Če zaradi morebitnih težav pri oddajanju rešitev na strežnik želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici, še preden odideš iz nje.**

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 `https://rtk.ijs.si/`. Predvidoma nekaj dni po tekmovanju bodo tam objavljeni tudi rezultati.

1. Varnostno kopiranje

Podan imamo seznam poti do več map oz. imenikov (direktorijev) na računalniku, ki jih želimo kopirati na zunanji disk za varnostno kopijo. Toda na seznamu so tudi podvojene mape ter mape, ki so že podmape drugih map. Če imamo na seznamu mape `/home/user/`, `/home/user` in `/home/user/slike`, je to v resnici enako seznamu samo s `/home/user/`, saj se prvi dve poti nanašata na isto mapo, zadnja pa je podmapa in je ni treba kopirati posebej, saj bo skopirana, ko bomo kopirali njeno starševsko mapo.

Napiši program, ki prebere seznam poti in izpiše le tiste izmed njih, ki jih je treba skopirati, da se izognemo nepotrebnemu kopiranju. Pri tem ni pomemben vrstni red izpisanih poti, važno je le, da so podmnožica podanih poti in da jih je čim manj. Če obstaja več enako dobrih rešitev, je vseeno, katero izpišeš.

Predpostaviš lahko, da so poti na voljo v datoteki `poti.txt` in vsaka pot je v svoji vrstici. Zagotovljeno je, da se vse poti začnejo s poševnico „`/`“, imena map pa bodo vsebovala le male črke angleške abecede, številke ter podčrtaj „`_`“. Vse poti bodo veljavne (ni ti treba npr. skrbeti zaradi poti oblike `abc//def`). Dolžina vsake poti bo manjša od 4096 znakov.

Poti je lahko veliko, zato naj bo tvoj program čim bolj učinkovit.

Primer vhoda:

```
/home/admin/config
/home/user/slike
/home/user/slike_stare
/home/user/dokumenti/sola/
/home/user/dokumenti/sola/slo/spisi/
/home/admin/config/
/home/user/minecraft/savegames/svet1
/home/user/minecraft/savegames/svet2
/home/user/slike/2019/smucanje/
/var/www/web/
/home/admin/config/
```

Možen izhod (vrstni red ni pomemben):

```
/home/user/slike
/home/user/slike_stare
/home/user/dokumenti/sola/
/home/admin/config/
/home/user/minecraft/savegames/svet1
/home/user/minecraft/savegames/svet2
/var/www/web/
```

2. Luči

Dano je zaporedje n luči; znano je njihovo začetno stanje (katere so prižgane in katere ugasnjene) ter želeno končno stanje. Osnovna operacija, ki jo lahko nad lučmi izvajamo, je, da si izberemo števili i in j (z območja $1 \leq i \leq j \leq n$) in spremenimo stanje vseh luči od vključno i-te do vključno j-te (torej ugasnemo tiste med njimi, ki so bile prej prižgane, in prižgemo tiste, ki so bile prej ugasnjene).

Napiši podprogram (oz. funkcijo) Luci(n, zacetno, koncno), ki izpiše zaporedje čim manj takšnih operacij, s katerimi lahko luči iz začetnega stanja spravimo v želeno končno stanje. Za vsako operacijo naj izpiše števili i in j, torej številko prve in zadnje spremenjene luči (luči so oštevilčene od 1 do n). Če obstaja več enako dobrih rešitev z najmanjšim možnim številom operacij, je vseeno, katero izpiše. Kot parametra zacetno in koncno naj tvoj podprogram sprejme niza n znakov, ki opisujeta začetno in končno stanje luči (znak `'P'` predstavlja prižgano luč, znak `'U'` pa ugasnjeno).

Dobro tudi **utemelji**, zakaj tvoja rešitev res najde najmanjše možno število operacij.

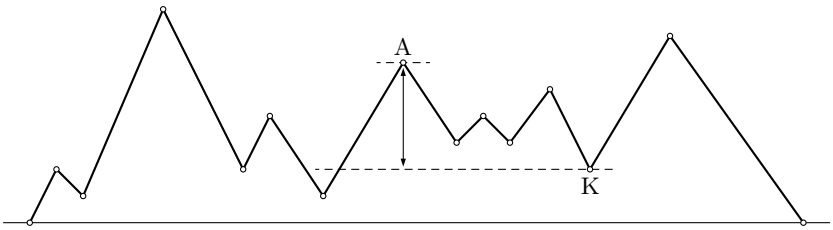
Primer: če imamo $n = 7$ luči in je začetno stanje `UUPPPUP`, končno pa `UPPUPPU`, potrebujemo vsaj tri operacije. Eno od možnih zaporedij treh operacij je: $i = 2$, $j = 5$ (dobimo `UPUUUUP`); $i = 4$, $j = 7$ (dobimo `UPUPPPU`); $i = 3$, $j = 4$ (dobimo `UPPUPPU`).

3. Planinarjenje

Za planince je pri izbiri ciljev ponavadi bolj zanimiv vrh, ki se izraziteje dviga višje od svoje okolice, manj pomembna pa je njegova absolutna nadmorska višina.

V ta namen je vpeljan pojem *topografske prominence* vrha (ali njegova *relativna višina*). Ta je določena z višinsko razdaljo (razliko višin) med vrhom in najnižjo tako plastnico (izohipso) terena, ki obkroža ta vrh in hkrati ne obkroža kakšnega višjega vrha. Z drugimi besedami: če bi se morska gladina dvignila do tiste plastnice, bi bil ta vrh najvišja točka otoka.

Problem si poenostavimo v dve dimenziji. Na spodnji sliki je prominenca vrha A razlika med nadmorskima višinama točk A in K (če bi se voda dvignila do višine točke K oz. tik nad njo, bi bila točka A najvišji vrh svojega otoka):



Da se ne trudimo z iskanjem vrhov in dolin, so podatki že pripravljeni tako, da si v seznamu (ali vhodni datoteki) sledijo izmenoma nadmorske višine dolin in vrhov. (Na gornji sliki so to točke, označene s krožci ∘, tako da bi za ta primer dobili seznam 15 višin.) Podatkov je liho število, seznam pa se začne in konča z dolino na nadmorski višini 0 (vse ostale višine so večje od 0).

Napiši program ali podprogram (funkcijo), ki bo za vsak vrh iz seznama ugotovil in izpisal njegovo prominenco. (Na primeru iz gornje slike bi moral program torej izpisati rezultate za 7 vrhov.) Tvoj (pod)program lahko seznam dobi kot parameter ali globalno spremenljivko (vektor, tabelo ali kaj podobnega), lahko pa ga prebere s standardnega vhoda ali iz vhodne datoteke (kar ti je lažje).

4. Sedežni red

Zloglasni 5. c je dobil novo učiteljico. Ta se je odločila, da bo v razredu naredila red, in sicer tako, da bo otroke posedla na točno določen način. Po novih pravilih noben otrok ne sme sedeti za otrokom, ki je strogo višji kot on sam, prav tako pa drug poleg drugega v isti vrsti ne smeta sedeti dva dečka ali dve deklici, ker bi bilo v tem primeru zagotovo preveč klepetanja.

Primera dobrih sedežnih redov za 6 otrok, kjer črka predstavlja spol, številka pa višino:

```
143M   150Z
139Z   128M
129M   127Z
  (TABLA)

139Z   154M
135Z   148M
135Z   129M
  (TABLA)
```

Dva primera slabih postavitev:

```
130Z   154M
135Z   148M
130Z   129M
  (TABLA)
```

(V gornji postavitvi je učenka levo v zadnji vrsti nižja od učenke pred njo.)

```
150Z   154M
145Z   148Z
141M   129M
  (TABLA)
```

(V tej postavitvi v prvi in drugi vrsti drug poleg drugega sedita otroka istega spola.)

Opiši postopek (ali napiši (pod)program oz. funkcijo, če ti je lažje), ki prejme podatke o višinah in spolu vseh u otrok v razredu ter jih razporedi v n vrst in m stolpcev tako, kot si je to zaželela učiteljica (ali pa ugotovi, da takšen razpored sploh ne obstaja). Za število stolpcev m predpostavi, da je sodo. Če je možnih več različnih pravilnih postavitev, je vseeno, katero izmed njih najde tvoj postopek. S podrobnostmi branja vhodnih podatkov in izpisa rezultatov se ti ni treba ukvarjati.

5. Žabe

Nad močvirjem leta r rojev muh. Vsak roj je sestavljen iz zelo velikega števila muh. Gibanje posameznega roja opišemo z zaporedjem koordinat in časov, ko se roj na svoji poti za trenutek ustavi. Tako bi postanke i-tega roja muh opisali s seznamom trojic $(x_{i,j}, y_{i,j}, t_{i,j})$, ki predstavljajo j-ti postanek i-tega roja na koordinati $(x_{i,j}, y_{i,j})$ ob času $t_{i,j}$. Predpostavimo lahko, da so postanki na poti posameznega roja podani po naraščajočih časih, torej $t_{i,j} < t_{i,j+1}$.

Poleg muh se v močvirju nahaja tudi z žab, ki lovijo muhe. Žaba lahko ulovi muho iz roja samo v trenutku, ko se roj ustavi, če se ravno takrat nahaja na istem mestu kot roj. Poznamo tudi lokacije žab: i-ta žaba se trenutno (ob času 0) nahaja na koordinati (a_i, b_i). Vse žabe se lahko premikajo po močvirju s hitrostjo 1 enote na sekundo (lahko pa tudi stojijo pri miru). Žaba se lahko med poljubnima dvema točkama premika v ravni črti, torej za merjenje razdalje uporabi evklidsko razdaljo (Pitagorov izrek).

Žabe načrtujejo lov na muhe. Ulovile bodo nekaj muh, nato pa se zbrale v koordinatnem izhodišču $(0, 0)$, kjer si bodo plen razdelile. Vsaka žaba bo ujela največ eno muho. Ker so muhe dobro rejene, je žabam dovolj, če vse skupaj ujamejo k muh (velja $k \leq z$), ki si jih nato razdelijo.

Opiši in utemelji **postopek**, ki bo določil najkrajši čas, v katerem se lahko vse žabe zberejo v izhodišču in pri tem skupaj ulovijo k muh. Koordinate in časi so realna števila.

17. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2022

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.

Tvoji programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali python, mi pa jih bomo preverili s prevajalniki FreePascal, GNUjevima `gcc` in `g++` 9.4.0 (ta verzija podpira C++17, novejše različice standarda C++ pa le delno), prevajalnikom za javo iz JDK 17, s prevajalnikom Mono 6.8 za C# in z interpreterjem za python 3.8.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2022-3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Pred začetkom tekmovanja lahko poskusiš oddati katero od nalog iz arhiva `https://putka-rtk.acm.si/tasks/s/test-sistema/list/`. Uporabniško ime in geslo za Putko sta enaki kot za računalnike. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava „Diskusija“ na dnu besedila posamezne naloge).

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih. 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 predolgo ali pa porabil preveč pomnilnika (točne omejitve so navedene na ocenjevalnem sistemu pri besedilu vsake naloge), 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 (za podrobne nastavitve prevajalnikov na ocenjevalnem strežniku glej `https://putka-rtk.acm.si/info/`. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku.

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 vse točke, če je izpisal pravilen odgovor, sicer pa 0 točk (izjema je 1. naloga, kjer je možno tudi delno točkovanje). Pri drugi nalogi je testnih primerov 20 in vsak je vreden po 5 točk, pri ostalih nalogah pa je testnih primerov po 10 in vsak je vreden po 10 točk.

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

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

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

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));
  }
}
```

17. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2022

NALOGE ZA TRETJO SKUPINO

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

1. Mastermind

Pri igri Mastermind računalnik izbere zaporedje n barvnih žetonov (barve se lahko ponavljajo) iz nabora m barv, ki jih označujemo s števili od 1 do m, in ga ne razkrije igralcu. Igralec nato poskuša uganiti izbrano zaporedje, računalnik pa mu ob vsakem ugibanju odgovori s parom števil, od katerih prvo pove, koliko žetonov je v danem ugibanju prave barve in na pravem mestu, drugo število pa pove, koliko žetonov je prave barve, vendar ne na pravem mestu.

Če je žetonov neke barve v ugibanju več kot v računalnikovem zaporedju, se pri vračanju informacij upošteva samo toliko žetonov, kot jih je v računalnikovem zaporedju, pri čemer imajo prednost žetoni prave barve na pravem mestu. Na primer: če računalnik izbere 1 1 2 2 2 in če igralec ugiba 1 3 1 1 1, bo računalnik vrnil $(1, 1)$. (Prvi žeton v igralčevem zaporedju je prave barve na pravem mestu, eden od zadnjih treh pa je prave barve na napačnem mestu.)

Recimo, da je v nekem ugibanju igralec predlagal zaporedje u, računalnik pa mu je odgovoril s parom števil o; rekli bomo, da je zaporedje z *skladno* z ugibanjem (u, o), če bi računalnik v primeru, da je njegovo izbrano zaporedje (ki ga mora igralec uganiti) ravno z in da je igralec pri ugibanju predlagal zaporedje u, odgovoril z odgovorom o.

Naš igralec je že podal k ugibanj in dobil odgovore nanje. V svojem naslednjem ugibanju želi predlagati takšno zaporedje p, po katerem bo tudi v najslabšem primeru (najslabšem po vseh možnih računalnikovih odgovorih) število še skladnih zaporedij (torej takih, ki so skladna z vsemi preteklimi ugibanji in tudi s tem novim, pravkar izvedenim ugibanjem) najmanjše možno. S tem kriterijem pa p ni nujno enolično določen; mogoče je, da obstaja več enako dobrih optimalnih predlogov p. **Napiši program**, ki iz zgodovine ugibanj in računalnikovih odgovorov nanje določi število vseh možnih takšnih zaporedij p, poleg tega pa še število vseh zaporedij, ki so skladna z dosedanjimi ugibanji in odgovori.

Vhodni podatki: v prvi vrstici so cela števila n (dolžina zaporedij), m (število barv) in k (število preteklih ugibanj). Nato sledi k vrstic s po $n + 2$ števili; pri tem prvih n števil predstavlja igralčevo ugibanje, zadnji dve pa računalnikov odgovor.

Omejitve: $1 \leq n \leq 5$, $1 \leq m \leq 8$, $0 \leq k \leq 5$; število še skladnih zaporedij glede na dano zgodovino ugibanj je vsaj 1 in manj kot 100.

Izhodni podatki: izpiši dve celi števili, vsako v svoji vrstici. Prvo naj bo število zaporedij, ki so skladna z dosedanjimi ugibanji in odgovori nanje; drugo pa naj bo število takšnih zaporedij p, o katerih govori besedilo naloge.

Točkovanje: za vsako od obeh izhodnih števil dobiš polovico točk, če je pravilno (še vseeno pa moraš izpisati dve števili, sicer ne dobiš nobene točke).

Primer vhoda:	Pripadajoči izhod:

`3 5 2`
`1 2 3 1 1`
`1 2 4 1 1`
	`4`
	`30`

Razlaga: računalnikova zaporedja, ki so skladna z dano zgodovino, so štiri: 1 1 2, 1 5 2, 2 2 1 in 5 2 1. Če igralec ugiba na primer 1 5 3, je računalnikov odgovor pri vseh še skladnih zaporedjih drugačen; v najslabšem primeru (in pravzaprav v vseh primerih) bo po tem ugibanju torej obstajalo samo še eno tako zaporedje, ki bo skladno z vsemi odgovori. To pa ne velja le za ugibanje 1 5 3, pač pa še za devetindvajset drugih ugibanj, zato je pravilni odgovor pri tem testnem primeru 30.

2. Dolgovi

Srednja šola Butale se je odločila, da bo po prekinitvi ukrepov zaradi popularnega virusa organizirala ekskurzijo. Na njej ima vseh n dijakov polpenzion, kar pomeni, da si bodo morali kar nekaj obrokov priskrbeti sami.

Butalski dijaki pa so prijazna bitja in nočejo po nepotrebnem obremenjevati raznih natakarjev, dostavljalcev hrane in drugih prodajalcev. Zato so dogovorjeni, da jedo v skupini (njeno članstvo se skozi čas lahko spreminja), pri čemer vsak kdaj plača nakup in bodo na koncu ekskurzije poravnali račune med sabo. Pri tem ne upoštevajo morebitnih razlik v ješčosti dijakov in si bodo zneske razdelili enakomerno po vseh, ki so trenutno prisotni v skupini. Ker pa matematika naših Butalcev šepa, te prosijo, da jim **napišeš program**, ki iz zapisa njihovega zapravljanja izračuna, koliko je kdo „v plusu“ ali „v minusu“. Ta zapis je zaporedje *dogodkov* naslednjih treh tipov:

1. `+` *ime* ali `-` *ime* — pomeni, da se je oseba z imenom *ime* pridružila skupini (če je prvi znak `+`) ali jo zapustila (če je prvi znak `-`). Kdor skupino zapusti, se ji lahko kasneje tudi spet pridruži.

2. `+` *prefiks*`*` ali `-` *prefiks*`*` — pomeni, da so se skupini pridružile oz. jo zapustile vse tiste osebe, ki so bile do tega trenutka že omenjene v dogodkih prvega tipa in ki se jim ime začne na niz *prefiks*. Dogodkom tega tipa pravimo „prefiksni dogodki“. Vsaka oseba je torej prvič dodana s svojim celotnim imenom, kasneje pa je lahko dodana ali odstranjena tudi samo s prefiksom. Prefiks je lahko tudi prazen — v tem primeru se vrstica nanaša na vse dotlej poznane osebe.

3. `<-` *ime* z — pomeni, da je oseba z imenom *ime* darovala znesek z v skupno dobro. Pri tem se njen dolg do skupnosti zmanjša za ta znesek, nato pa se dolg vseh trenutnih članov skupine poveča za z/k, če je k trenutno število ljudi v skupini (predpostaviš lahko, da se dogodki tega tipa nikoli ne zgodijo takrat, ko je skupina prazna). Oseba, ki nekaj plača v skupno dobro, v tistem trenutku ni nujno tudi sama član skupine (je pa bila pred plačilom že vsaj enkrat dodana v skupino).

Predpostaviš lahko, da ne bo nobeno dodajanje (niti posamezno niti s prefiksom) pokrivalo kakšne take osebe, ki je trenutno že v skupini, in prav tako ne bo nobeno brisanje (niti posamezno niti s prefiksom) pokrivalo kakšne take osebe, ki je trenutno ni v skupini. Pri prefiksnih dogodkih velja to zagotovilo samo za tiste osebe, ki so bile pred tem vsaj enkrat že dodane v skupino. (Primer: zaporedje dogodkov `+ jaka`, `+ jure`, `- jaka`, `- j*` se v vhodnih podatkih ne more pojaviti; prav tako ne zaporedje `+ jaka`, `+ jure`, `- j*`, `+ jaka`, `+ j*`. Lahko pa se pojavi zaporedje `+ jaka`, `- j*`, `+ j*`, `+ jure`.)

Vhodni podatki: v prvi vrstici je celo število q, ki pove, koliko dogodkov sestavlja ta testni primer. Sledi q vrstic, od katerih vsaka opisuje po en dogodek v taki obliki, kot je opisano zgoraj.

Izhodni podatki: za vsako osebo, ki je bila kdaj dodana v skupino, izpiši po eno vrstico; v njej naj bo najprej ime osebe, nato presledek, nato pa znesek, ki ga ta oseba dolguje skupnosti (to število je lahko tudi negativno, če je v resnici skupnost dolžna tej osebi). Vrstni red oseb v izhodnih podatkih je lahko poljuben.

Rešitev se bo štela kot pravilna, če bodo dolgovi od uradne rešitve odstopali za manj kot 10^{-5} po relativni ali absolutni vrednosti.

Omejitve:

- Vedno bo veljalo $2 \leq q \leq 10^6$. Če z n označimo število različnih imen, omenjenih v ne-prefiksnih dogodkih, bo veljalo tudi $1 \leq n \leq 10^5$.
- Imena so sestavljena iz vsaj 1 in največ 10 znakov, vsi pa so male črke angleške abecede; pri prefiksnih dogodkih je prefiks dolg vsaj 0 in največ 9 malih črk, za njim pa pride še znak `*` (zvezdica). Imena so enolična (nobena dva človeka nimata enakega imena).
- Zneski pri dogodkih `<-` so cela števila z območja $0 \leq z < 10^5$.

(*Nadaljevanje na naslednji strani.*)

Pri nekaterih testnih primerih veljajo dodatne omejitve:

- Pri prvih 20 % testnih primerov bo veljalo $n \leq 1000$ in $q \leq 5000$.
- Pri naslednjih 10 % testnih primerov ne bo prefiksnih dogodkov in veljalo bo $n \leq 1000$ ter $q_3 \leq 5000$ (kjer q_3 pomeni število dogodkov 3. tipa).
- Pri naslednjih 10 % testnih primerov ne bo prefiksnih dogodkov.
- Pri preostalih 60 % testnih primerov ni dodatnih omejitev.

Primer vhoda:

```
14
+ ljubinka
<- ljubinka 37
+ vilma
<- vilma 80
+ ticjana
<- ljubinka 48
+ onur
<- vilma 54
+ bernardica
- ticjana
+ ticjana
<- ticjana 22
<- vilma 35
<- onur 62
```

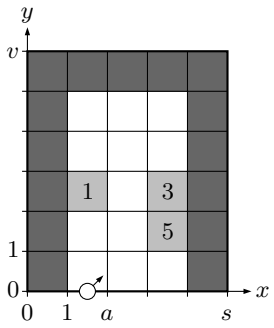
Eden od možnih pripadajočih izhodov:

```
vilma -75.700000
onur -24.700000
ticjana 31.300000
ljubinka 45.300000
bernardica 23.800000
```

3. Breakout

Morda poznate računalniško igro Breakout. Cilj igre je z odbijanjem žogice razbiti vse opeke na igralnem polju. Igralno polje je z leve, desne in zgornje strani obdano s stenami, znotraj polja pa se nahaja več opek. Žogica se odbija po polju in razbija opeke ob odboju od njih, pri tem pa moramo s premikanjem odbojne palice na spodnjem robu skrbeti, da žogica slučajno ne pade iz igralnega polja.

V tej nalogi ne bomo upoštevali odbojne palice, temveč nas zanima samo, kako bi se žogica odbijala po igralnem polju brez posredovanja igralca. Polje predstavimo s pravokotno mrežo širine s in višine v. Levo spodnje oglišče mreže se nahaja na koordinati $(0, 0)$, desno zgornje pa na (s, v). Nekatere celice polja vsebujejo opeke, ki se razbijejo po določenem številu odbojev žogice od njih. Najbolj levi in desni stolpec ter zgornja vrstica vsebujejo stene, ki jih bomo predstavili s trdnimi opekami, ki se razbijejo šele po 100 odbojih. Če se opeka zaradi odboja razbije, se to zgodi šele po odboju — žogica torej svojo pot nadaljuje enako, kot bi jo, če se opeka ob tem odboju ne bi razbila. Žogica začne svojo pot v spodnji vrstici na sredini a-tega stolpca, torej na koordinatah $(a - \frac{1}{2}, 0)$, in se od tam sprva (do prvega odboja) premika pod kotom 45° v smeri desno navzgor, kot kaže spodnja ilustracija:



Primer igralnega polja širine $s = 5$ in višine $v = 6$. Temno siva polja so trdne opeke, ki tvorijo stene in se razbijejo po 100 odbojih; svetlo siva polja pa so običajne opeke in števila na njih povedo, po koliko odbojih se razbijejo. Prikazan je tudi začetni položaj in smer gibanja žogice za $a = 2$.

Napiši program, ki izračuna, koliko opek razbije žogica, preden izstopi iz igralnega polja, in na kateri koordinati se to zgodi. Izstopi lahko v spodnji vrstici ali pa tudi kje drugje, če je zaradi številnih odbojev prebila steno.

Vhodni podatki: v prvi vrstici so podana cela števila s, v, n in a, ki po vrsti predstavljajo širino igralnega polja, njegovo višino, število opek znotraj polja in začetni stolpec žogice. Naslednjih n vrstic opisuje opeke znotraj polja (v to niso vključene opeke, ki predstavljajo stene). Vsaka opeka je opisana v svoji vrstici s števili x_i, y_i in k_i, pri čemer sta (x_i, y_i) koordinati zgornjega desnega kota celice, ki jo ta opeka zaseda, k_i pa je število odbojev, po katerih se opeka razbije (velja $2 \leq x_i \leq s - 1$, $1 \leq y_i \leq v - 1$ in $1 \leq k_i \leq 100$). Opeke se bodo nahajale v različnih celicah igralnega polja in nobena od njih nima koordinat $(a, 1)$, pri katerih bi se prekrivala z začetnim položajem žogice.

Omejitve: veljalo bo $3 \leq s \leq 10^5$, $3 \leq v \leq 10^5$ in $0 \leq n \leq 10^5$. V prvih 40 % testnih primerov bo veljalo $s, v, n \leq 1000$. V naslednjih 40 % testnih primerov bo veljalo $s, v, n \leq 10^5$. V zadnjih 20 % testnih primerov ni dodatnih omejitev.

Izhodni podatki: v prvi vrstici izpiši število razbitih opek (kar naj vključuje tudi razbite stenske opeke), v drugi vrstici pa s presledkom ločeni koordinati x in y, na katerih žogica izstopi iz polja. Če je koordinata celo število, jo tako tudi izpiši, če pa je ne-cela, jo izpiši na eno decimalko.

Primer vhoda:	Pripadajoči izhod:

`5 6 3 2`
`2 3 1`
`4 2 5`
`4 3 3`
	`1`
	`3.5 0`

(*Opomba:* to je primer, ki ga kaže slika zgoraj.)

4. Pijansko urejanje

V skladišču je n podstavkov, oštevilčenih od 0 do $n-1$. Na vsakem podstavku stoji po en zaboj; tudi ti so oštevilčeni od 0 do $n-1$, vendar so premešani — številke zabojev se ne ujemajo s številkami podstavkov, na katerih posamezni zaboj stoji. Radi bi jih pravilno uredili, torej tako, da bo zaboj 0 na podstavku 0, zaboj 1 na podstavku 1 in tako naprej. (Različni zaboji imajo različne številke; različni podstavki prav tako.)

Žal se z viličarjem po skladišču preganja pijani skladiščnik in nas ne pusti blizu, da bi si ogledali razpored zabojev ali jih sami premikali. Edino, kar lahko počnemo, je, da mu od zunaj vpijemo številke podstavkov; ko sliši od nas številko k, naj bi skladiščnik zamenjal zaboja na podstavkih 0 in k. Žal pri tem včasih zgreši in namesto podstavka k uporabi enega od sosednjih dveh ($k-1$ in $k+1$), vendar ne vemo, kateri podstavek je zares uporabil. Vemo pa, da si podstavek med temi tremi (torej $k-1$, k in $k+1$) izbere naključno, pri čemer imajo vsi trije podstavki enako verjetnost, da bodo izbrani. (Možnost $k-1$ seveda odpade, če je $k=0$; in podobno možnost $k+1$ odpade, če je $k=n-1$. V teh primerih skladiščnik naključno izbere enega od ostalih dveh možnih podstavkov.) V vsakem primeru pa nam skladiščnik po zamenjavi pove številko zaboja, ki je pri tem prišel na podstavek 0 (ne glede na to, s katerega podstavka je ta zaboj prišel).

Napiši program, ki z zaporedjem takšnih operacij uredi zaboje (torej poskrbi, da se bo številka vsakega podstavka ujemala s številko zaboja, ki stoji na njem). To je interaktivna naloga; tvoj program se bo z ocenjevalnim strežnikom „pogovarjal“ tako, da bo bral s standardnega vhoda in pisal na standardni izhod. Ta pogovor naj poteka po naslednjih korakih:

1. Na začetku preberi s standardnega vhoda eno vrstico, v kateri bo celo število n (in nič drugega).

2. Nato lahko izvedeš 0 ali več zamenjav. Zamenjavo izvedeš tako, da na standardni izhod izpišeš eno vrstico, v kateri naj bo le celo število k (zanj mora veljati $0 \le k < n$), torej številka podstavka, za katerega želiš izvesti zamenjavo (še enkrat poudarimo, da bo računalnik zamenjavo mogoče izvedel s podstavkom $k-1$ ali $k+1$ namesto k). Nato s standardnega vhoda preberi eno vrstico; v njej bo le eno celo število, namreč številka zaboja, ki je pri tej zamenjavi prišel na podstavek številka 0.

3. Na koncu izpiši vrstico, v kateri naj bo le celo število -1, in prenehaj z izvajanjem.

Opozorilo: po vsaki izpisani vrstici splakni standardni izhod (*flush*), da bodo podatki res sproti prišli do ocenjevalnega sistema.

Tvoj program dobi pri posameznem testnem primeru vse točke, če izvede kvečjemu 10 000 zamenjav in če so na koncu zaboji pravilno urejeni; sicer pa ne dobi pri njem nobene točke (npr. če izvede preveč zamenjav, če zaboji na koncu niso pravilno urejeni ali če izpiše kaj, kar ni v skladu z zgoraj opisanim protokolom).

Omejitve: veljalo bo $2 \le n \le 100$. Pri 50 % testnih primerov bo $n \le 40$.

Primer:

Tvoj program izpiše	Sistem izpiše	Komentar
	`2`	v skladišču sta dva zaboja
`1`	`1`	zaboj 1 je prišel na podstavek 0
`0`	`1`	zaboj 1 je še vedno na podstavku 0
`1`	`0`	zaboj 0 je prišel na podstavek 0
`-1`		končali smo z urejanjem

V gornjem primeru smo po tretji zamenjavi od sistema izvedeli, da je zaboj 0 prišel na podstavek 0; torej mora biti takrat zaboj 1 na podstavku 1 in lahko sklepamo, da sta oba zaboja urejena tako, kot naloga zahteva.

5. L-sistem

Teoretični biolog Polde preučuje obnašanje preprostih mnogoceličnih mikroorganizmov. Organizem je zaporedje celic; obstaja a različnih tipov celic in Polde je vsakemu tipu celice pripisal neko malo črko angleške abecede (črko, ki predstavlja i-ti tip celice, označimo s t_i), zato lahko organizem opiše z nizom takih črk. Organizem se skozi čas razvija, vendar na zelo predvidljiv način: v enem dnevu iz vsake celice tipa t_i nastane (vedno enako) zaporedje 0 ali več celic, ki ga torej spet lahko predstavimo z nizom znakov; temu nizu recimo $f(t_i)$.

Iz organizma, ki ga tvori niz n celic $s = c_1c_2 \ldots c_n$, torej v enem dnevu nastane niz $f(c_1) \cdot f(c_2) \cdot \ldots \cdot f(c_n)$; temu novemu nizu recimo $f(s)$. Pri tem pike „$\cdot$“ pomenijo, da moramo nize $f(c_1), \ldots, f(c_n)$ po vrsti stakniti med seboj.

Po dveh dneh torej iz s nastane niz $f(f(s))$, kar zapišimo krajše kot $f^2(s)$; po treh dneh nastane $f(f(f(s)))$, kar zapišimo krajše kot $f^3(s)$; in tako naprej. Nizu, ki nastane po k dneh, recimo $f^k(s)$. Velja torej $f^0(s) = s$ in $f^k(s) = f(f^{k-1}(s))$.

Poldeta še posebej zanimajo organizmi, pri katerih se v njihovem nizu čim večkrat pojavlja podniz p ali kakšen od njegovih anagramov (to so nizi, ki jih lahko dobimo iz p, če premešamo vrstni red črk v njem). Pomagaj mu in **napiši program**, ki za dani začetni niz s in število dni k izračuna, koliko je v nizu $f^k(s)$ takih strnjenih podnizov, ki so anagrami niza p (tudi p je seveda sam svoj anagram). Ker utegne biti to število zelo veliko, bo dovolj, če v resnici izračunaš njegov ostanek po deljenju z nekim manjšim številom M.

Vhodni podatki: v prvi vrstici je niz $t_1t_2 \ldots t_a$, ki ima torej toliko znakov, kolikor različnih tipov celic je pri tem testnem primeru, in i-ti znak tega niza pove, s katero črko so predstavljene celice i-tega tipa. Sledi a vrstic, od katerih i-ta vsebuje niz $f(t_i)$ — to je niz, ki se v enem dnevu razvije iz celice i-tega tipa. V naslednji vrstici je niz s. V naslednji vrstici sta celi števili k in M, ločeni s presledkom. Nazadnje pride še vrstica z nizom p.

Omejitve: v spodnjih omejitvah zapis $|x|$ pomeni dolžino niza x.

- $1 \le a \le 26$; $0 \le k \le 300$; $2 \le M \le 10^9 + 20$;
- znaki t_i so vsi različni med sabo in vsi so male črke angleške abecede;
- $1 \le |s| \le 1000$; $1 \le |p| \le 300$; $0 \le |f(t_i)| \le 100$ za vse $i = 1, \ldots, a$;
- v vseh nizih $f(t_i)$, s in p nastopajo le črke iz množice $\{t_1, t_2, \ldots, t_a\}$, torej take, ki predstavljajo tipe celic, naštete v prvi vrstici vhodnih podatkov.

V 50 % testnih primerov bo veljalo tudi $|f^k(s)| \le 10^7$. V naslednjih 10 % primerov bo veljalo $|p| = 1$, v naslednjih 10 % primerov pa $|p| = 2$.

Izhodni podatki: izpiši rezultat, po katerem sprašuje besedilo naloge, torej število takih strnjenih podnizov niza $f^k(s)$, ki so anagrami p-ja; oz. natančneje povedano, izpiši ostanek po deljenju tega števila z M.

Primer vhoda:

```
vftb
b

ffvffv
fbv
tf
5 6
vbfb
```

Pripadajoči izhod:

```
3
```

Komentar: v tem testnem primeru nastopajo štirje tipi celic, označeni s črkami `v`, `f`, `t` in `b`. V enem dnevu iz celice tipa `v` nastane celica tipa `b`; iz celice tipa `f` nastane prazen niz (celica izgine brez sledu); iz celice tipa `t` nastane zaporedje šestih celic `ffvffv`; iz celice tipa `b` pa nastane zaporedje treh celic `fbv`. Niz $s =$ `tf` se v $k = 5$ dneh razvija takole:

$$\texttt{tf} \to \texttt{ffvffv} \to \texttt{bb} \to \texttt{fbvfbv} \to \texttt{fbvbfbvb} \to \texttt{fbvbfbvfbvbfbv}$$

Zadnji od teh nizov je $f^k(s)$ in ima kar 9 takih strnjenih podnizov, ki so anagrami niza $p =$ `vbfb` (edina njegova podniza dolžine $|p| = 4$, ki *nista* anagrama p-ja, sta `fbvf` in `vfbv`). Testni primer zahteva, da izpišemo ostanek po deljenju tega števila z $M = 6$; pravilni odgovor je torej 3 (ostanek po deljenju 9 s 6).

17. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2022

REŠITVE NALOG ZA PRVO SKUPINO

1. Snežinke

Naš program bo tekel v neskončni zanki. Na vsakem koraku preglejmo vse žice (z vgnezdeno zanko), da vidimo, če kakšna od njih zaznava snežinko. Če je tako, se pojavi vprašanje, ali moramo to snežinko šele izpisati ali pa smo jo morda že opazili in zaznali nekoč v preteklosti; spodnji program ima v ta namen spremenljivko snezinka, ki pove, ali smo snežinko zaznavali že tudi v prejšnji iteraciji glavne zanke. (Pri tem se zanašamo na zagotovilo iz besedila naloge, da se nova snežinka ne pojavi, dokler prejšnja ne izgine, in da vmes mine dovolj časa, da lahko naš program pregleda vse žice.)

Če se izkaže, da trenutne snežinke doslej še nismo videli in moramo torej zdaj izpisati njeno velikost, je treba ugotoviti, pri kateri žici se začne in konča. Ni nujno, da je žica, pri kateri smo jo doslej našli, že tudi najbolj leva žica te snežinke, saj naloga pravi, da se snežinka dotakne več žic hkrati in da se lahko pojavi kadarkoli med dvema klicema funkcije Nadzor. Iti moramo torej v zanki po žicah levo in desno od trenutne, dokler ne pridemo do take, ki snežinke ne zaznava več; tako bomo ugotovili najbolj levo in najbolj desno žico te snežinike, iz njiju pa lahko izračunamo njeno velikost in jo izpišemo.

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

int main()
{
  enum { N = 100 };       // Število žic.
  bool snezinka = false;  // Ali smo v zadnji iteraciji glavne zanke videli snežinko?
  while (true)
  {
    // Poiščimo prvo žico, ki zaznava snežinko.
    int L = 1; while (L <= N && ! Senzor(L)) ++L;
    // Morda snežinke ne zaznava nobena žica.
    if (L > N) { snezinka = false; continue; }
    // Morda smo snežinko videli že v prejšnji iteraciji glavne zanke.
    if (snezinka) continue;
    // Sicer poglejmo, kako daleč na desno se razteza.
    int D = L; while (D < N && Senzor(D + 1)) ++D;
    // Poglejmo še, kako daleč na levo se razteza.
    while (L > 1 && Senzor(L - 1)) --L;
    // Snežinka pokriva žice od L do D; izpišimo jo.
    cout << (D - L + 1) << endl;
    // Zapomnimo si, da smo jo že videli.
    snezinka = true;
  }
}
```

Še enaka rešitev v pythonu:

```
N = 100             # Število žic.
snezinka = False;   # Ali smo v zadnji iteraciji glavne zanke videli snežinko?
while True:
  # Poiščimo prvo žico, ki zaznava snežinko.
  L = 1
```

```
  while L <= N and not Senzor(L): L += 1

  # Morda snežinke ne zaznava nobena žica.
  if L > N: snezinka = False; continue

  # Morda smo snežinko videli že v prejšnji iteraciji glavne zanke.
  if snezinka: continue

  # Sicer poglejmo, kako daleč na desno se razteza.
  D = L
  while D < N and Senzor(D + 1): D += 1

  # Poglejmo še, kako daleč na levo se razteza.
  while L > 1 and Senzor(L - 1): L -= 1

  # Snežinka pokriva žice od L do D; izpišimo jo.
  print(D - L + 1)

  # Zapomnimo si, da smo jo že videli.
  snezinka = True
```

2. Semafor

Ko prvič pokličemo BeriStevec, nimamo nobene podlage za to, da bi se odločili, katera od številk, ki nam jih je funkcija vrnila, je prava. Ob naslednjem klicu BeriStevec vemo, da je številka na prikazovalniku zdaj za 1 manjša kot ob prvem klicu; če je na primer zdaj na prikazovalniku številka k, je morala biti ob prvem klicu tam številka $k + 1$; in ker vemo, da nam BeriStevec vrne med drugim tudi pravo številko, to pomeni, da nam je ob prvem klicu gotovo vrnil $k + 1$, ob drugem pa k. Tako so torej zdaj kandidati za pravo številko le tista števila k, ki jih je BeriStevec vrnila ob drugem klicu in za katera je ob prvem klicu vrnila $k + 1$.

Podobno razmišljamo tudi v nadaljevanju. Po tretjem klicu so kandidati za pravo številko le tisti k, ki jih je BeriStevec vrnila ob tretjem klicu in za katere je bila vrednost $k + 1$ eden od kandidatov po drugem klicu. Tako nadaljujemo in vse bolj klestimo množico kandidatov, dokler ne ostane v njej eno samo število; tisto mora biti potem prava trenutna številka na prikazovalniku in jo lahko izpišemo ter končamo z izvajanjem.

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

extern vector<int> BeriStevec();

int main()
{
  vector<int> kandidati = BeriStevec(), noviKandidati;
  while (kandidati.size() > 1)
  {
    noviKandidati.clear();
    for (int k : BeriStevec())
      // Število k, ki ga kamera trenutno zaznava, je smiselno upoštevati kot kandidata
      // le, če smo prejšnjo sekundo imeli med kandidati število k + 1.
      if (find(kandidati.begin(), kandidati.end(), k + 1) != kandidati.end())
        noviKandidati.push_back(k);
    swap(kandidati, noviKandidati);
  }
  cout << kandidati.front() << endl; return 0;
}
```

Še v pythonu:

```
kandidati = set(BeriStevec())
while len(kandidati) > 1:
  # Izmed števil k, ki jih vrne BeriStevec v naslednjem klicu,
  # obdržimo le tiste, za katere je bil k + 1 kandidat po prejšnjem klicu.
```

```
    kandidati = set(k for k in BeriStevec() if k + 1 in kandidati)
# Izpišimo edinega preostalega kandidata.
print(list(kandidati)[0])
```

3. Iskanje kvadrata

Število vrstic označimo z n_v, število stolpcev pa z n_s. Naj bo s_i vsota širin prvih i stolpcev, v_j pa vsota višin prvih j vrstic. Tega dvojega ni težko izračunati iz zaporedij širin stolpcev in višin vrstic, ki ju dobimo kot vhodna podatka. Pravokotnik, ki ga iščemo, je potem v vsakem primeru velikosti $s_i \times v_j$ za neki dve števili i in j; vprašanje pa je, kako izbrati i in j, da bo ta pravokotnik čim bolj kvadraten.

Recimo, da se za hip omejimo na pravokotnike višine v_j, torej take, ki se raztezajo čez prvih j vrstic. Vzemimo najmanjši tak i, pri katerem je $s_i \geq v_j$. Potem so pravokotniki $s_k \times v_j$ za $k \geq i$ vsaj tako široki kot visoki in razmerje dolžine med daljšo in krajšo stranico je pri njih enako s_k/v_j; najbližje 1 je takrat, ko je s_k najmanjša, to pa je pri $k = i$. Med vsemi temi pravokotniki je torej kandidat za najboljšega le $s_i \times v_j$.

Podobno so pravokotniki $s_k \times v_j$ za $k < i$ vsaj tako visoki kot široki, zato je razmerje dolžine med daljšo in krajšo stranico pri njih enako v_j/s_k; najbližje 1 je takrat, ko je s_k največja, to pa je pri $k = i - 1$. Med vsemi temi pravokotniki je torej kandidat za najboljšega le $s_{i-1} \times v_j$.

Tako torej vidimo, da sta med vsemi pravokotniki oblike $s_k \times v_j$ (pri nekem konkretnem j) za nas zanimiva le dva, namreč tista za $k = i$ in $k = i - 1$. Ker vnaprej ne vemo, pri katerem j bomo dobili najboljši rezultat, bomo preizkusili vse možne j od 1 do števila vrstic. Recimo torej, da v zanki počasi povečujemo j za 1; kako se pri tem spreminja i? Spomnimo se, da smo i definirali kot najmanjše število stolpcev, pri katerem je $s_i \geq v_j$. Ko se j poveča za 1, se tudi v_j poveča (za višino naslednje vrstice), zato je pogoj $s_i \geq v_j$ še strožji kot prej; vsak tak s_i, ki je bil že prej premajhen, je zdaj še bolj premajhen; morda bo isti s_i kot doslej še vedno dovolj velik, drugače pa ga bo treba še povečati. Torej, ko se j poveča za 1, se lahko i poveča ali pa ostane enak, ne more pa se zmanjšati.

Zapišimo ta postopek s psevdokodo:

$i := 1$;
for $j := 1$ **to** n_v:
  while $i < n_s$ **and** $s_i < v_j$ **do** $i := i + 1$;
  if $i > 1$ **then** pravokotnik $s_{i-1} \times v_j$ je kandidat za rešitev;
  pravokotnik $s_i \times v_j$ je kandidat za rešitev;

Med vsemi kandidati za rešitev si zapomnimo tistega, ki je najbolj kvadraten. Časovna zahtevnost tega postopka je $O(n_v + n_s)$, saj naredimo n_v iteracij zunanje zanke, iteracij notranje zanke pa je vsega skupaj največ n_s.

Gornji postopek se premakne z j na $j+1$ pri takem i, za katerega je $s_i \geq v_j$; z drugimi besedami, če je pravokotnik širši kot višji, ga spodaj podaljšamo za eno vrstico (kar je smiselno, kajti če bi ga namesto tega na desni razširili za en stolpec, bi postal še bolj preširok kot prej). In po drugi strani, gornji postopek se premakne z i na $i + 1$ takrat, ko je $s_i < v_j$; z drugimi besedami, če je pravokotnik višji kot širši, ga na desni razširimo za en stolpec (kar je tudi smiselno, kajti če bi ga namesto tega spodaj podaljšali za eno vrstico, bi postal še bolj previsok kot prej).

Naš postopek lahko torej opišemo še preprosteje kot doslej, če rečemo takole: začnemo pri $i = j = 1$; potem pa, če je trenutni pravokotnik preširoč, mu dodamo eno vrstico (povečamo j za 1); sicer pa je preozek in mu dodamo en stolpec (povečamo i za 1). Med vsemi tako pregledanimi pravokotniki si zapomnimo tistega, ki je najbolj kvadraten. Oglejmo si implementacijo te rešitve v C++:

```
#include <vector>
#include <utility>
#include <algorithm>

pair<int, int> NajboljsiPravokotnik(const vector<int> &sirine, const vector<int> &visine)
{
```

```
  int ns = sirine.size(), nv = visine.size();
  int i = 1, j = 1, si = sirine[0], vj = visine[0];
  int najS = si, najV = vj;
  while (i <= ns || j <= nv)
  {
    // si = vsota prvih i širin; vj = vsota prvih j višin.
    // Če je pravokotnik višji kot širši, ga razširimo za en stolpec.
    if (si <= vj && i < ns) si += sirine[i++];

    // Če pa je širši kot višji, ga spodaj podaljšajmo za eno vrstico.
    else if (vj < si && j < nv) vj += visine[j++];

    // Če ga v izbrani smeri ne moremo povečati, končajmo,
    else break; // saj se lahko drugače rešitev le še poslabša.

    // Če je to najboljša rešitev doslej, si jo zapomnimo. Preveriti moramo
    // torej, ali je max(si, vj) / min(si, vj) < max(najS, najV) / min(najS, najV).
    // Da ne bo treba delati z ne-celimi števili, pomnožimo to
    // neenačbo z imenovalcema obeh ulomkov.
    if (max(si, vj) * min(najS, najV) < max(najS, najV) * min(si, vj))
      najS = si, najV = vj;
  }
  return {najS, najV}; // Vrnimo najboljšo rešitev.
}
```

In v pythonu:

```
def NajboljsiPravokotnik(sirine, visine):
  ns = len(sirine); nv = len(visine)
  i = 1; j = 1; si = sirine[0]; vj = visine[0]
  najS = si; najV = vj

  while i <= ns or j <= vj:
    # si = vsota prvih i širin; vj = vsota prvih j višin.
    # Če je pravokotnik višji kot širši, ga razširimo za en stolpec.
    if si <= vj and i < ns: si += sirine[i]; i += 1

    # Če pa je širši kot višji, ga spodaj podaljšajmo za eno vrstico.
    elif vj < si and j < nv: vj += visine[j]; j += 1

    # Če ga v izbrani smeri ne moremo povečati, končajmo,
    else: break # saj se lahko drugače rešitev le še poslabša.

    # Če je to najboljša rešitev doslej, si jo zapomnimo. Preveriti moramo
    # torej, ali je max(si, vj) / min(si, vj) < max(najS, najV) / min(najS, najV).
    # Da ne bo treba delati z ne-celimi števili, pomnožimo to
    # neenačbo z imenovalcema obeh ulomkov.
    if max(si, vj) * min(najS, najV) < max(najS, najV) * min(si, vj):
      najS = si; najV = vj

  return najS, najV # Vrnimo najboljšo rešitev.
```

4. Neprevidni poeti

Za posamezno vrstico lahko določimo njen ritem tako, da jo beremo znak po znak; če je trenutni znak samoglasnik in velika črka, dodamo v niz, ki predstavlja ritem, znak `'-'`; če je trenutni znak samoglasnik in mala črka, dodamo v ritem znak `'U'`; vse ostale znake besedila pa lahko ignoriramo.

Pri prvi vrstici moramo ritem le izračunati in si ga zapomniti (v spodnji rešitvi ga shranimo v spremenljivko ritem1). Pri vsaki naslednji vrstici pa njen ritem primerjamo s tistim iz prve vrstice; če pride do neujemanja, lahko takoj končamo, saj vemo, da bo rezultat na koncu `NE`. Če pa pridemo do konca vhodnih podatkov, ne da bi opazili kakšno neujemanje, izpišemo `DA` in ritem prebranih vrstic.

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

```
int main()
{
  string ritem, ritem1;      // Ritem trenutne in prve vrstice.
  bool vseEnake = true;      // Ali imajo vse doslej prebrane vrstice enak ritem?
  while (true)
  {
    int c = getchar();
    // Ko preberemo samoglasnik, dodamo U ali - v ritem trenutne vrstice.
    if (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') ritem += '-';
    else if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') ritem += 'U';
    // Ali smo na koncu vrstice?
    else if (c == '\n' || c == EOF)
    {
      // Ritem prve vrstice si zapomnimo.
      if (ritem1.empty()) ritem1 = move(ritem);
      // Pri ostalih primerjamo ritem trenutne s prvo; prazne preskočimo.
      else if (! ritem.empty() && ritem != ritem1) { vseEnake = false; break; }
      if (c == EOF) break;
      ritem.clear(); // Pripravimo se na naslednjo vrstico.
    }
  }
  // Izpišimo rezultat.
  printf("%s\n", vseEnake ? "DA" : "NE");
  if (vseEnake) printf("%s\n", ritem1.c_str());
  return 0;
}
```

Oglejmo si še primer rešitve v pythonu. Vhodne podatke bomo brali kar po vrsticah, saj naloga zagotavlja, da niso pretirano dolge:

```
import sys

ritem1 = ""        # Ritem prve vrstice.
vseEnake = True    # Ali imajo vse doslej prebrane vrstice enak ritem?

for vrstica in sys.stdin:
  ritem = []
  for c in vrstica:
    # Ko preberemo samoglasnik, dodamo U ali - v ritem trenutne vrstice.
    if c in "AEIOU": ritem.append('-')
    elif c in "aeiou": ritem.append('U')
  # Ritem predelamo iz seznama v niz.
  ritem = "".join(ritem)
  if not ritem: continue # Preskočimo prazne vrstice.
  # Ritem prve vrstice si zapomnimo.
  if not ritem1: ritem1 = ritem; continue
  # Pri ostalih primerjamo ritem trenutne vrstice s prvo.
  if ritem != ritem1: vseEnake = False; break

# Izpišimo rezultat.
print("DA" if vseEnake else "NE")
if vseEnake: print(ritem1)
```

Za ljubitelje pretiravanja z regularnimi izrazi: ritem bi lahko računali tudi tako, da bi iz vrstice pobrisali vse minuse, nato spremenili velike samoglasnike v minuse, nato male samoglasnike v U-je in končno pobrisali vse ostale črke.

```
import re
ritem = re.sub("[^U\\-]", "", re.sub("[aeiou]", "U", re.sub("[AEIOU]", "-",
                                                    vrstica.replace("-", ""))))
```

5. Stonoge

Polje lahko predstavimo kot seznam (ali tabelo ali vektor) nizov, pri čemer vsak niz

predstavlja eno vrstico polja. V zanki pregledujmo znake, dokler ne najdemo znaka #, ki kaže, da se tam začenja trup stonoge. Zapomnimo si njegov položaj in se v zanki premaknimo naprej mimo vseh znakov #, ki mu sledijo, dokler ne najdemo konca stonoge. Zdaj vemo, kje se trup začne in konča, in gremo lahko v še eni zanki po vseh parih nog (ne pozabimo tudi na tista tik pred in tik za trupom); za vsak par nog preverimo, če sta simetrični (torej je lahko ena \ in druga / ali pa sta obe |), pri prvem in zadnjem paru pa še, če sta usmerjeni k trupu. Če je z nogami vse v redu, povečamo števec prepričljivih stonog, sicer pa števec umetnih.

V prvi in zadnji vrstici ter v prvem in zadnjem stolpcu nam trupov (znakov #) načeloma ni treba iskati, saj naloga zagotavlja, da nobena stonoga ni v kadru le delno. To pa potem tudi pomeni, da ko ob trupu pregledujemo noge, nam ni treba skrbeti, da bi pri tem poskušali dostopati do znakov z neveljavnimi indeksi (onkraj roba mreže).

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

void Preglej(const vector<string>& a, int &prepricljive, int &umetne)
{
  int h = a.size(), w = a[0].length(); prepricljive = 0; umetne = 0;
  for (int y = 1; y < h - 1; ++y) for (int x = 1; x < w - 1; ++x) if (a[y][x] == '#')
  {
    // Poglejmo, do kod gre trup te stonoge.
    int xx = x + 1; while (xx < w - 1 && a[y][xx] == '#') ++xx;
    // Trup stonoge obsega znake od a[y][x] do a[y][xx - 1].
    // Preverimo, ali so noge simetrične in usmerjene k trupu.
    bool ok = true;
    for (int u = x - 1; u <= xx; ++u) {
      char zgoraj = a[y - 1][u], spodaj = a[y + 1][u];
      if (! (zgoraj == '\\' && spodaj == '/' && u < xx ||
             zgoraj == '/' && spodaj == '\\' && u >= x ||
             zgoraj == '|' && spodaj == '|' && u >= x && u < xx))
        { ok = false; break; } }
    // Povečajmo ustreznega izmed števcev.
    if (ok) ++prepricljive; else ++umetne;
    x = xx; // Nadaljujmo desno od te stonoge.
  }
}
```

Še enaka rešitev v pythonu:

```
def Preglej(a):
  h = len(a); w = len(a[0]); prepricljive = 0; umetne = 0
  for y in range(1, h - 1):
    x = 1
    while x < w - 1:
      if a[y][x] != '#': x += 1; continue
      # Pri x se začenja trup stonoge. Kje se konča?
      xx = x
      while xx < w - 1 and a[y][xx] == '#': xx += 1
      # Trup stonoge obsega znake od a[y][x] do a[y][xx - 1].
      # Preverimo, ali so noge simetrične in usmerjene k trupu.
      ok = True
      for u in range(x - 1, xx + 1):
        zgoraj = a[y - 1][u]; spodaj = a[y + 1][u]
        if not (zgoraj == '\\' and spodaj == '/' and u < xx or
          zgoraj == '/' and spodaj == '\\' and u >= x or
          zgoraj == '|' and spodaj == '|' and x <= u < xx):
          ok = False; break
      # Povečajmo ustreznega izmed števcev.
      if ok: prepricljive += 1
```

```
        else: umetne += 1
        x = xx + 1 # Nadaljujmo desno od te stonoge.
    return (prepricljive, umetne)
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Varnostno kopiranje

Naloga med drugim pravi, da če se dve poti nanašata na isti direktorij, moramo eno od njiju zavreči. Taki dve poti nista nujno čisto enaki; lahko se razlikujeta po tem, da ima ena na koncu poševnico `/`, druga pa ne (na primer: `/ab/cd` in `/ab/cd/` predstavljata isti direktorij). Za lažje preverjanje tega pogoja je torej koristno, če tistim potem, ki se v vhodnih podatkih ne končajo na poševnico, le-to dodamo.

S tako dopolnjenimi potmi pa je potem lažje preverjati tudi drugi pogoj, namreč ali ena pot predstavlja poddirektorij druge. To je namreč res natanko tedaj, ko je druga pot prefiks prve (ali, z drugimi besedami: ko se prva pot začne na drugo). Če na primer pogledamo poti `/ab/cd/` in `/ab/`, vidimo, da je druga prefiks prve (niz `"/ab/cd/"` se začne na niz `"/ab/"`), zato prva predstavlja poddirektorij druge in jo lahko zavržemo. To, da smo potem dodali znak `/` na koncu, kjer ga še ni bilo, je koristno zato, ker bi drugače na primer pri `/ab/cd` in `/ab/c` videli, da je drugi niz prefiks prvega, čeprav prvi niz ne predstavlja poddirektorija drugega.

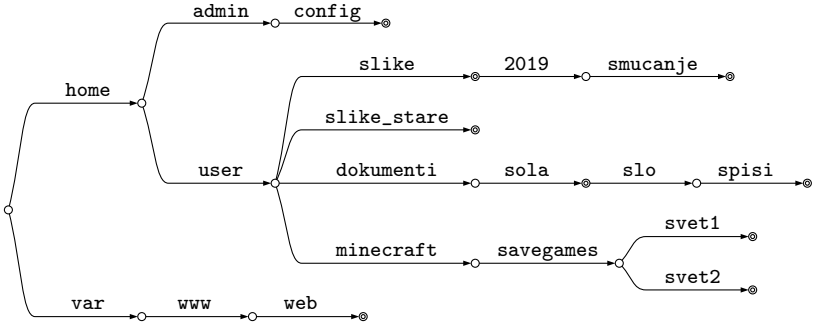
Oba pogoja lahko zdaj zelo preprosto preverjamo tako, da poti uredimo leksikografsko. Če je v vhodnem seznamu več enakih poti, bodo tako prišle skupaj in bomo odvečne zlahka zavrgli; poti za poddirektorije pa bodo prišle takoj za njihove naddirektorije in jih tudi ne bo težko opaziti. Pri izpisu pa moramo paziti na to, da naloga zahteva, da morajo biti izpisane poti podmnožica vhodnih; to pomeni, da če je bila neka pot v vhodu prisotna brez poševnice na koncu, jo moramo tudi mi izpisati brez poševnice (razen če je bila pot do istega direktorija nekje drugje v vhodnem seznamu zapisana tudi *s* poševnico in se odločimo izpisati to različico namesto tiste brez poševnice). V ta namen je koristno ob vsaki poti hraniti še podatek o tem, ali je poševnico na koncu imela že v vhodnem seznamu ali smo ji jo dodali šele mi.

```
#include <vector>
#include <string>
#include <cstring>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <utility>

int main()
{
  ifstream ifs("poti.txt");
  vector<pair<string, bool>> poti;
  while (true)
  {
    // Preberimo naslednjo pot.
    string pot; if (! getline(ifs, pot)) break;

    // Če nima poševnice na koncu, jo dodajmo.
    bool dodaj = (pot.back() != '/');
    if (dodaj) pot += '/';

    // Dodajmo pot na seznam.
    poti.emplace_back(pot, dodaj);
  }

  sort(poti.begin(), poti.end());
  string zadnjaIzpisana;

  for (auto &[pot, dodaj] : poti)
  {
    // Če smo že izpisali kakšno pot in če je zadnja izpisana pot prefiks trenutne,
```

```
    // to pomeni, da se trenutna nanaša na isti direktorij ali pa na nek njegov poddirektorij.
    if (! zadnjaIzpisana.empty() && strncmp(zadnjaIzpisana.c_str(), pot.c_str(),
                                            zadnjaIzpisana.length()) == 0) continue;
    // Sicer moramo trenutno pot izpisati.
    // Pred izpisom pa s konca pobrišimo poševnico, če smo jo prej dodali.
    zadnjaIzpisana = pot;
    if (dodaj) pot.pop_back();
    cout << pot << endl;
  }
  return 0;
}
```

Za preverjanje, ali je ena pot prefiks druge, smo uporabili `strncmp` iz C-jeve standardne knjižnice; od C++20 naprej pa lahko uporabimo tudi `pot.starts_with(zadnjaIzpisana)`.

Zaradi urejanja nizov je časovna zahtevnost te rešitve $O(n \log n)$, če imamo na vhodu n nizov. Za namen naše naloge je to dovolj dobro, vseeno pa si oglejmo še rešitev s časovno zahtevnostjo $O(n)$. Poti lahko pri znakih `/` razrežemo na komponente in jih zložimo v drevo (*trie*), kot kaže naslednja slika:



Pri dodajanju novih poti v drevo pazimo na to, da če se več poti ujema v prvih nekaj komponentah, si delijo tudi ustrezna vozlišča v drevesu. V vsakem vozlišču si tudi označimo, ali se pri njem konča kakšna pot iz vhodnega seznama (na sliki so taka vozlišča označena z dvojnimi krožci) oz. na katerem mestu v vhodnem seznamu se nahaja. Na koncu se moramo le sprehoditi po drevesu in izpisati poti pri tistih vozliščih, za katera noben njihov prednik ni že sam v vhodnem seznamu.

Oglejmo si implementacijo takšne rešitve v pythonu:

```
# Preberimo vhodno datoteko.
with open("poti.txt", "rt") as f: poti = f.readlines()

# Pripravimo drevo, v katerem je zaenkrat le koren.
koren = 0; stars = [−1]; otrok = {}; odKod = [−1]

# Dodajmo v drevo vse vhodne poti.
for i, pot in enumerate(poti):
  u = koren
  for s in pot.rstrip().split('/'):
    # Premaknimo se iz trenutnega vozlišča u dol v otroka, do katerega
    # pelje povezava z oznako s.
    v = otrok.get((u, s), −1)
    if v < 0: # Če takega otroka še ni, ga dodajmo.
      v = len(stars); stars.append(u); odKod.append(−1)
      otrok[u, s] = v
    u = v
  odKod[u] = i # Zapomnimo si, katera vhodna pot se konča pri tem vozlišču.

# Izpišimo rezultate.
for u in range(len(stars)):
  if odKod[u] < 0: continue
  # Pri vozlišču u se konča ena od vhodnih poti.
  # Ali se konča tudi pri kakšnem njegovem predniku?
```

```
    v = stars[u]
    while v >= 0 and odKod[v] < 0: v = stars[v]

    # Če ne, lahko trenutno pot izpišemo.
    if v < 0: print(poti[odKod[u]].rstrip())
```

Vozlišča so oštevilčena z zaporednimi številkami od 0 naprej (0 je koren); `stars[u]` nam pove, kdo je starš vozlišča `u`; `otrok[u, s]` pa je tisti otrok vozlišča `u`, do katerega pelje povezava, označena z nizom `s`. Vrednost `odKod[u]` je -1, če se pri `u` ne konča nobena vhodna pot, sicer pa je `odKod[u]` indeks te poti (oz. zadnje od njih, če jih je več) v vhodnem seznamu — to pride prav pri izpisu.

Z vidika števila vhodnih poti n ima ta rešitev časovno zahtevnost $O(n)$; bolj pošteno pa bi bilo reči, da je njena časovna zahtevnost $O(d)$, če je d skupna dolžina vhodnih poti, kajti v najslabšem primeru bo imelo drevo $O(d)$ vozlišč. Boljše časovne zahtevnosti od te pa niti ne moremo pričakovati, saj mora vsaka rešitev porabiti $O(d)$ časa že samo zaradi branja vhodne datoteke.

2. Luči

V enem koraku lahko spremenimo stanje nekaj zaporednih luči. Opazimo lahko, da je vseeno, v kakšnem vrstnem redu izvajamo te operacije, kajti končno stanje posamezne luči je odvisno le od tega, koliko operacij je vplivalo nanjo: če jih je bilo sodo mnogo, bo končno stanje luči enako začetnemu, sicer pa nasprotno od začetnega.

Recimo zdaj, da na neko luč x vplivata dve različni operaciji, na primer ena, ki spremeni luči od a-te do b-te, in druga, ki spremeni luči od c-te do d-te. Ker obe tidve operaciji vplivata na luč x, gotovo velja $a \le x \le b$ in $c \le x \le d$. Vzemimo $A = \min\{a, c\}$, $B = \max\{a, c\}$, $C = \min\{b, d\}$ in $D = \max\{b, d\}$. Učinek naših dveh operacij je torej takšen: na luči od A do $B-1$ vpliva le ena operacija; na tiste od $C+1$ do D le druga; na tiste od B do C pa obe. Toda če na isto luč vplivata dve operaciji, bo druga postavila to luč nazaj v tisto stanje, v katerem je bila pred prvo operacijo, torej je učinek enak, kot če na tako luč ne bi vplivala nobena operacija. Lahko bi torej naši dve operaciji spremenili tako, da bi prva delovala na luči od A do $B-1$, druga pa na luči od $C+1$ do D, pa bo učinek enak kot doslej, le da zdaj na nobeno luč ne bosta vplivali obe operaciji. (Če je $A = B$, lahko prvo od teh dveh operacij celo sploh pobrišemo; in podobno za drugo, če je $C = D$.)

S tem razmislekom lahko postopoma odpravimo vse primere, ko sta na kakšno luč vplivali (vsaj) dve različni operaciji, ne da bi se pri tem število operacij kaj povečalo. Torej optimalne rešitve ne bomo spregledali, če se bomo že od začetka omejili na takšna zaporedja operacij, pri katerih vsako luč spremeni kvečjemu ena operacija.

Če primerjamo, kje se istoležni znaki začetnega in končnega stanja luči razlikujejo, bomo videli, katerim lučem je treba stanje spremeniti. Vsaka operacija lahko poskrbi za največ eno strnjeno skupino takih luči. Najmanjše število operacij dobimo torej tako, da imamo za vsako tako strnjeno skupino po natanko eno operacijo.

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

void Luci(int n, const char *zacetno, const char *koncno)
{
  for (int j = 0; j < n; ++j)
  {
    int i = j; while (j < n && zacetno[j] != koncno[j]) ++j;

    // Luči od vključno i do vključno j − 1 se morajo spremeniti,
    // luč j pa ne več (zato lahko v naslednji iteraciji zunanje
    // zanke nadaljujemo pri j + 1). Pri izpisu uporabimo števila
    // od 1 do n namesto od 0 do n − 1.
    if (i < j) cout << (i + 1) << " " << j << endl;
  }
}
```

3. Planinarjenje

Recimo, da nas, tako kot na sliki v besedilu naloge, zanima relativna višina vrha A.

Pojdimo od A v desno, dokler ne naletimo na prvi višji vrh, recimo $d(A)$; in naj bo D_A najnižja dolina na tako prehojeni poti. Podobno pojdimo od A še v levo do prvega višjega vrha, recimo $\ell(A)$, in naj bo L_A najnižja dolina na tako prehojeni poti. Vzemimo za K_A višjo izmed dolin L_A in D_A. Če voda naraste do višine K_A, bo vrh A ločen tako od levega višjega vrha kot od desnega, torej bo najvišji na svojem otoku; po drugi strani, če voda naraste do kakšne nižje višine, bo A še vedno na istem otoku kot vsaj eden od omenjenih višjih vrhov (namreč tisti, pri katerem je najnižja dolina med A in njim na višini K_A). Torej je K_A ravno tista višina, o kateri govori naloga pri definiciji topografske prominence; topografska prominenca vrha A je razlika med njegovo višino in višino doline K_A.

Če morda desno od A-ja sploh ni nobenega višjega vrha, nas torej desna stran nič ne omejuje glede tega, kako visoko mora voda narasti, preden bo A najvišji vrh svojega otoka; to lahko zajamemo v gornji razmislek tako, da za D_A vzamemo točko višine 0 na koncu vhodnih podatkov. Podobno tudi za L_A, če levo od A-ja ni nobenega višjega vrha, vzamemo točko višine 0 na začetku vhodnih podatkov.

Vprašanje je torej zdaj, kako za vsak vrh A poiskati naslednji višji vrh v vsaki smeri (levo in desno) ter najnižjo dolino na poti do njega. Tega seveda ne bi bilo težko početi z zanko, toda če bomo takšno zanko izvedli za vsak vrh A, bo imel naš postopek na koncu časovno zahtevnost $O(n^2)$, pri čemer n pomeni dolžino vhodnega zaporedja:

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

void Planinarjenje1(const vector<int> &v)
{
  int n = v.size();
  for (int i = 1; i < n; i += 2)
  {
    int A = v[i];

    // Pojdimo v levo do naslednjega višjega vrha in si zapomnimo najnižjo dolino na poti.
    int L = A; for (int j = i; j >= 0 && v[j] <= A; --j) L = min(L, v[j]);

    // Nato naredimo enako še v desno.
    int D = A; for (int j = i; j < n && v[j] <= A; ++j) D = min(D, v[j]);

    // Izračunajmo topografsko prominenco vrha A in jo izpišimo.
    cout << (A - max(L, D)) << endl;
  }
}
```

Z malo pazljivosti pa lahko z enim samim prehodom čez podatke, v $O(n)$ časa, določimo *vsem* vrhovom A najbližji vrh na desni ter najnižjo dolino na poti do njega. Tako bomo za vsak A dobili njegov D_A, nato pa bomo podobno pregledali vhodno zaporedje še v nasprotni smeri in za vsak A dobili še njegov L_A.

Recimo, da smo trenutno pri vrhu A; naj bo $\mathcal{D}(A)$ zaporedje vrhov $d(A), d(d(A)), \ldots$ — vsak je višji od prejšnjega in leži desno od njega. Prvi od njih je ravno $d(A)$, ki nas zanima. Recimo zdaj, da se od A premaknemo levo do naslednjega vrha, na primer B. Zdaj nas zanima $\mathcal{D}(B)$; ali ga lahko poceni dobimo iz $\mathcal{D}(A)$? Če je B nižji od A, bo $d(B) = A$, zato dobimo $\mathcal{D}(B)$ iz $\mathcal{D}(A)$ tako, da slednjemu na levem koncu dodamo še A. Če pa je B vsaj tako visok kot A, potem mora $d(B)$ ležati desno od A; in ker bo $d(B)$ višji od B, bo višji tudi od A; prvi primerni kandidat za $d(B)$ je torej kar $d(A)$. Če je tudi ta prenizek, bo naslednji primerni kandidat šele $d(d(A))$ in tako naprej. Zaključimo torej lahko, da dobimo $\mathcal{D}(B)$ iz $\mathcal{D}(A)$ v vsakem primeru tako, da slednjemu na levem koncu dodamo A in nato z levega konca pobrišemo vse vrhove, ki niso višji od B. Ker moramo ves čas brisati in dodajati na levem koncu zaporedja $\mathcal{D}$, je primerna podatkovna struktura za predstavitev tega zaporedja sklad, pri čemer vrh sklada predstavlja levi konec zaporedja. Tako bo prav na vrhu sklada vedno ravno tisti vrh, ki ga potrebujemo za $d(A)$. Ko se premikamo od desne proti levi po vhodnem zaporedju, dodamo vsak vrh enkrat na sklad in ga največ enkrat pobrišemo s sklada, zato je časovna zahtevnost vseh teh operacij skupaj le $O(n)$.

Res pa je, da nas pravzaprav ne zanima toliko $d(A)$ sam po sebi, pač pa najnižja dolina med njim in A — to je D_A, ki ga potrebujemo pri določanju topografske prominence A-ja. Na skladu je torej koristno ob vsakem vrhu hraniti še najnižjo dolino na poti od tega vrha do naslednjega višjega vrha (tistega, ki je na skladu eno mesto pod trenutnim vrhom). Ko pobiramo vrhove s sklada, lahko spotoma še računamo minimalno višino pripadajočih dolin, pa bomo na koncu dobili najnižjo dolino med A in $d(A)$.

Oglejmo si implementacijo te rešitve v C++. V prvem prehodu pregledamo vhodno zaporedje od desne proti levi, računamo D_A-je in jih shranjujemo v vektor D; v drugem prehodu pa pregledamo vhodno zaporedje od leve proti desni, računamo L_A-je in skupaj z D_A-ji (shranjenimi iz prvega prehoda) še topografske prominence, ki jih tudi sproti izpisujemo.

```
#include <stack>

vector<int> Planinarjenje2(const vector<int> &v)
{
  struct Par { int vrh, dno; };
  int n = v.size(); vector<int> D(n / 2);
  for (int prehod = 1; prehod <= 2; ++prehod) {
    // V prvem prehodu gremo od desne proti levi, v drugem pa od leve proti desni.
    stack<Par> S; S.push({-1, 0});
    for (int j = 1; j < n; j += 2) {
      int i = prehod == 2 ? j : n - 1 - j;
      int A = v[i], C = v[prehod == 2 ? i - 1 : i + 1];
      // Trenutni vrh je A, pred njim je dolina C. Pojdimo nazaj po zaporedju
      // do naslednjega višjega vrha, v C pa računajmo najnižjo dolino na tej poti.
      while (C > 0 && S.top().vrh <= A) {
        C = min(C, S.top().dno); S.pop(); }
      // Dodajmo na sklad trenutni vrh ter najnižjo dolino na poti do naslednjega
      // višjega vrha.
      S.push({A, C});
      // Pri prvem prehodu (od desne proti levi) si C le zapomnimo.
      if (prehod == 1) D[i / 2] = C;
      // Pri drugem prehodu (od leve proti desni) pa lahko že izračunamo topografsko
      else cout << A - max(D[i / 2], C) << endl; } }                    // prominenco.
}
```

4. Sedežni red

Naloga pravi, da je število stolpcev m sodo; recimo torej, da je $m = 2k$. V posamezno vrsto lahko torej postavimo največ k dečkov, saj bi drugače neizogibno morala sedeti dva skupaj v isti vrsti. Enako velja tudi za deklice. Če je torej enih in/ali drugih več kot $k \cdot n$, lahko takoj zaključimo, da razporeda, kakršnega zahteva naloga, ni mogoče sestaviti.

Recimo torej zdaj, da je dečkov kvečjemu $k \cdot n$, deklic pa tudi. Potem lahko sestavimo razpored, pri katerem sedijo v lihih stolpcih samo dečki, v sodih pa samo deklice. Dečke poljubno razdelimo v skupine po n, dokler jih pač ne zmanjka (nastane največ k skupin, pri čemer je v zadnji lahko tudi manj kot n otrok). Vsako skupino razporedimo v enega od lihih stolpcev (teh je k, torej vsaj toliko kot skupin), pri čemer seveda otroke v skupini uredimo po višini in jih postavimo tako, da so višji bolj zadaj. Nato enako naredimo še z deklicami.

```
#include <vector>
#include <algorithm>
using namespace std;
struct Otrok { int visina; char spol; };

bool Razporedi(int stVrstic, int stStolpcev, const vector<Otrok> &otroci)
{
  // Preštejmo dečke in deklice posebej.
  int stDeckov = 0, stDeklic = 0;
```

```
  for (auto &O : otroci) if (O.spol == 'M') ++stDeckov; else ++stDeklic;
  // Če je enih ali drugih preveč, primernega razporeda ni.
  if (max(stDeckov, stDeklic) > stVrstic * (stStolpcev / 2)) return false;
  // Razdelimo jih v stolpce.
  vector<vector<Otrok>> stolpci(stStolpcev); stDeckov = 0; stDeklic = 0;
  for (auto &O : otroci)
    if (O.spol == 'M') stolpci[2 * (stDeckov++ / stVrstic)].push_back(O);
    else stolpci[2 * (stDeklic++ / stVrstic) + 1].push_back(O);
  // Vsak stolpec uredimo padajoče po višini.
  for (auto &stolpec : stolpci) sort(stolpec.begin(), stolpec.end(),
    [] (const auto &x, const auto &y) { return x.visina > y.visina; });
  // Izpišimo rezultate (od zadnjih vrstic proti sprednjim).
  for (int y = 0; y < stVrstic; ++y)
    for (int x = 0; x < stStolpcev; ++x) {
      if (stolpci[x].size() <= y) printf("  ");
      else printf("%3d%c", stolpci[x][y].visina, stolpci[x][y].spol);
      putchar(x == stStolpcev - 1 ? '\n' : ' '); }
  return true;
}
```

Še en način, da pridemo do primernega razporeda, pa je naslednji: uredimo vse dečke po višini, nato pa v zadnjo vrsto posedemo najvišjih k dečkov (v lihe stolpce), v predzadnjo vrsto naslednjih k po višini in tako naprej. Nato naredimo enako še s deklicami, le da jih pošiljamo v sode stolpce. Manjša slabost pri tej rešitvi je, da moramo urejati do $k \cdot n$ otrok naenkrat (vse dečke ali vse deklice), pri prejšnji pa smo jih urejali le po n naenkrat (vsak stolpec posebej).

5. Žabe

Razdaljo med točkama (x, y) in (x', y') označimo z $d(x, y, x', y') = ((x - x')^2 + (y - y')^2)^{1/2}$. Ker se žabe premikajo s hitrostjo ene enote na sekundo, je $d(x, y, x', y')$ tudi čas, v katerem žaba pride od točke (x, y) do (x', y').

Za vsako žabo izračunajmo najkrajši čas, v katerem lahko ujame muho in pride v koordinatno izhodišče — recimo temu T_s za s-to žabo. Da ga izračunamo, pojdimo v zanki po vseh položajih vsakega roja; s-ta žaba lahko ujame muho na j-tem postanku i-tega roja le v primeru, če lahko od svojega začetnega položaja (a_s, b_s) pride do točke $(x_{i,j}, y_{i,j})$ v $t_{i,j}$ ali manj časa (sicer ji bo roj ušel naprej po svoji poti, še preden bo prišla do tja). Če je ta pogoj izpolnjen, gre potem lahko žaba ob času $t_{i,j}$ naprej proti koordinatnemu izhodišču in ga doseže po $d(x_{i,j}, y_{i,j}, 0, 0)$ časa.

Mogoče je tudi, da žaba ne more ujeti nobene muhe, ker je predaleč od vseh položajev vsakega roja; takrat si mislimo zanjo $T_s = \infty$.

Naloga pravi, da je dovolj, če muhe lovi k žab, ostale pa gredo lahko naravnost od svojega začetnega položaja proti koordinatnemu izhodišču. Za takšno pot porabi s-ta žaba $U_s := d(a_s, b_s, 0, 0)$ časa. Množico žab, ki lovijo muhe, označimo z A; čas, ko se vse žabe zberejo v koordinatnem izhodišču, je potem $f(A) = \max\{\max_{s \in A} T_s, \max_{s \notin A} U_s\}$.

Za vsako žabo seveda velja $U_s \leq T_s$, kajti pri U_s potuje ta žaba od začetnega položaja naravnost v koordinatno izhodišče, pri T_s pa naredi vmes še ovinek do kraja, kjer bo ujela muho (in morda tam celo še čaka, da bo roj sploh prišel tja); zato ima pri U_s žaba krajšo pot (trikotniška neenakost) kot pri T_s. To pa pomeni, da ni nobene koristi od tega, da bi muhe lovilo več žab, kot je nujno potrebno, kajti v rešitvi, kjer lovi muhe več kot k žab, lahko kakšno od njih pošljemo naravnost v koordinatno izhodišče, pa se rešitev ne bo nič poslabšala: če žabo s vržemo iz množice A, bo odslej v $f(S)$ prispevala manjšo vrednost U_s namesto večje vrednosti T_s, torej maksimum tega po vseh žabah ne bo nič večji kot prej.

Lahko se torej omejimo na rešitve, kjer muhe lovi natanko k žab. Recimo, da žabe oštevilčimo naraščajoče po T_s, tako da je $T_1 \leq T_2 \leq \ldots \leq T_z$. Potem je najbolje, če na lov pošljemo kar prvih k žab, torej vzamemo $A = \{1, 2, \ldots, k\}$. Prepričajmo se, da je to res. Recimo, da bi obstajala neka še boljša rešitev B. Ker tudi tam lovi muhe le k žab in ker sta množici A in B različni, mora obstajati neka žaba $i \leq k$, ki v A lovi muhe, v B

pa ne; in obstajati mora tudi neka žaba $j > k$, ki lovi muhe v B, ne pa v A. Recimo, da bi v B tema dvema žabama zamenjali vlogi, tako da bi i lovila muho, j pa ne; dobimo rešitev $C := B - \{j\} \cup \{i\}$. Zaradi te spremembe porabi žaba j zdaj manj ali enako časa kot prej (U_j namesto T_j), žaba i pa porabi zdaj manj časa (namreč T_i), kot ga je žaba j porabila prej (namreč T_j; spomnimo se, da je $i \leq k < j$, zato je $T_i \leq T_j$). Maksimum porabljenega časa po obeh žabah torej ni zdaj v C nič večji, kot je bil prej v B, ostalim žabam pa se čas sploh ni spremenil. Torej je C vsaj tako dobra rešitev kot B, poleg tega pa se ujema z našo rešitvijo A v eni žabi več kot B. Tako bi lahko nadaljevali in rešitev korak za korakom spremenili v A, ne da bi se kdaj poslabšala; torej je tudi A bila optimalna rešitev.

Oglejmo si še implementacijo te rešitve v C++:

```
#include <vector>
#include <utility>
#include <algorithm>
#include <limits>
#include <cmath>
using namespace std;

struct Tocka { double x, y; };
struct Postanek { Tocka kje; double kdaj; };

// Izračuna razdaljo med točkama in s tem tudi čas, ki ga žaba porabi za pot med njima.
double D(const Tocka &a, const Tocka &b) {
  double dx = a.x - b.x, dy = a.y - b.y; return sqrt(dx * dx + dy * dy); }

double Zabe(const vector<Tocka>& zabe, const vector<vector<Postanek>>& roji, int k)
{
  const Tocka cilj { 0, 0 };
  vector<pair<double, double>> casi; // pari (T_s, U_s)

  for (const Tocka &zaba : zabe)
  {
    // Izračunajmo najkrajši čas, v katerem lahko ta žaba ujame muho in pride na cilj.
    double T = numeric_limits<double>::infinity();
    for (auto &roj: roji) for (auto &postanek : roj)

      // Ali lahko žaba pravočasno doseže kraj postanka?
      if (D(zaba, postanek.kje) <= postanek.kdaj)

        // Če po postanku nadaljuje pot, kdaj doseže cilj?
        T = min(T, postanek.kdaj + D(postanek.kje, cilj));

    // Izračunajmo še najkrajši čas, v katerem lahko pride žaba na cilj brez lova na muho.
    casi.emplace_back(T, D(zaba, cilj));
  }

  // Uredimo žabe po T_s.
  sort(casi.begin(), casi.end());

  // Prvih k žab bo lovilo muhe in porabilo T_s časa, ostale gredo
  // naravnost na cilj in porabijo U_s časa.
  double rezultat = 0;
  for (int s = 0; s < zabe.size(); ++s)
    rezultat = max(rezultat, s < k ? casi[s].first : casi[s].second);

  return rezultat;
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Mastermind

Razmislimo najprej o tem, kako izračunati odgovor $o = (o_1, o_2)$, ki ga dobimo, če pri ugibanju predlagamo zaporedje $u = (u_1, \ldots, u_n)$, pravo zaporedje pa je $z = (z_1, \ldots, z_n)$. Žetonov prave barve na pravem mestu ni težko prešteti (to bo o_1), če gremo v zanki po indeksih od 1 do n in primerjamo istoležne žetone. Če pa pri nekem indeksu i pride do neujemanja (torej če velja $u_i \neq o_i$), je žeton u_i morda prave barve na napačnem mestu. Natančneje povedano: recimo, da v neujemanjih v zaporedju u nastopa $h_u[b]$ žetonov barve b, v zaporedju z pa $h_z[b]$ žetonov barve b. Potem lahko za žetone prave barve na pravem mestu razglasimo $\min\{h_u[b], h_z[b]\}$ žetonov — torej vse take žetone v u-ju, če jih ni več kot v z-ju, sicer pa le toliko, kolikor jih je v z-ju. Ob prvem prehodu čez zaporedji torej računajmo oba „histograma“ (torej tabeli h_u in h_z, ki za vsako barvo povesta število žetonov te barve na neujemajočih se mestih zaporedij u oz. z), nato pa seštejmo $\min\{h_u[b], h_z[b]\}$ po vseh barvah b in tako dobimo drugi del odgovora, število o_2.

Prvi del naloge zahteva, da preštejemo vsa skladna zaporedja; pravzaprav jih bomo tudi nekam shranili, ker bodo prišla prav pri drugem delu naloge. Pojdimo po vseh možnih zaporedjih z (teh je m^n, ker imamo za vsako od n mest po m možnosti, katere barve žeton je tam) in za vsako preverimo, ali bi dobili, če bi bilo to zaporedje pravo, pri dosedanjih ugibanjih ravno tiste odgovore, ki so zapisani v vhodnih podatkih. V ta namen uporabimo še vgnezdeno zanko po dosedanjih ugibanjih; čim pri kakšnem ugibanju vidimo, da bi bil odgovor pri z-ju drugačen od tistega v vhodnih podatkih, lahko nad trenutnim z obupamo in se lotimo naslednjega.[1] Na koncu nam tako nastane množica zaporedij, ki so skladna z vsemi dosedanjimi ugibanji; recimo ji S.

Malo več dela pa je z drugim delom naloge. Pojdimo po vseh m^n možnih zaporedjih p, ki bi jih igralec lahko predlagal v naslednjem ugibanju. Pri različnih kandidatih $z \in S$ bi ob takem ugibanju lahko nastali različni odgovori o; lahko si predstavljamo, da je ugibanje p razbilo množico S na podmnožice oblike $S_{p,o} := \{z \in S : \mathit{odgovor}(p, z) = o\}$. Z vidika igralca se torej, če na ugibanje p dobi odgovor o, množica kandidatov zmanjša s S na $S_{p,o}$. V za igralca najslabšem primeru je torej množica kandidatov po takem ugibanju ravno največja od vseh $S_{p,o}$, tako da mu ostane po ugibanju še $J(p) := \max_o |S_{p,o}|$ kandidatov. Vrednost $J(p)$ si lahko predstavljamo kot „oceno“ ugibanja p; naloga sprašuje, pri koliko različnih p nastopi najmanjša (in s tem najugodnejša) možna ocena, torej $\min_p J(p)$. Če torej za vsak p izračunamo oceno $J(p)$, ni težko določiti najmanjše ocene in tudi šteti, pri koliko p-jih je nastopila. Ocene posameznega p-ja pa tudi ni težko računati: lahko vzdržujemo tabelo z velikostmi množic $S_{p,o}$ za vse možne odgovore o; na začetku postavimo vse te velikosti na 0, nato pa gremo po vseh skladnih zaporedjih $z \in S$, pri vsakem izračunamo odgovor $o = \mathit{odgovor}(p, z)$ in povečamo v tabeli velikost množice $S_{p,o}$ za 1.

Oglejmo si implementacijo takšne rešitve v C++. Za naštevanje vseh možnih zaporedij lahko uporabimo kar zanko, ki gre po številih od 0 do $m^n - 1$ in vsako od njih sproti pretvori v zaporedje n števil (enako kot pri pretvorbi števila v m-iški številski sestav). Podobno tudi odgovor $o = (o_1, o_2)$ predstavimo s številom $o_1(m + 1) + o_2$. Za štetje velikosti množic $S_{p,o}$ imamo tabelo stKand; ker bo ocena trenutnega p-ja na koncu enaka maksimumu po vseh vrednostih te tabele, lahko nad tem p-jem takoj obupamo, če kakšna vrednost v tabeli preseže najmanjšo dosedanjo oceno; ta drobna optimizacija skrajša čas izvajanja za več kot polovico (je pa program tudi brez nje več kot dovolj hiter).

```
#include <iostream>
#include <vector>
#include <algorithm>
```

[1] Vse možne z lahko generiramo z rekurzijo in pri tem do neke mere že sproti preverjamo, ali je trenutno zaporedje sploh še mogoče podaljšati do nečesa, kar bo skladno z vsemi dosedanjimi ugibanji in odgovori nanje. Na primer, morda je bilo pri kakšnem ugibanju že z doslej zgeneriranim delom z-ja toliko ujemanj, da bo na koncu vrednost o_1 gotovo večja od tiste v vhodnih podatkih. Vendar pa pri naših testnih primerih ni nobene potrebe po tovrstnih optimizacijah, saj bo naš program levji delež časa tako ali tako porabil za drugi del naloge.

```
using namespace std;

enum { MaxN = 5, MaxM = 8 };
int n, m; // m barv, n žetonov v zaporedju

int Odgovor(int ugibanje, int prava)
{
  // hu[b], hp[b] = število takih žetonov barve b (v „ugibanje“ oz. „prava“),
  // ki niso prave barve na pravem mestu.
  int hu[MaxM] = {}, hp[MaxM] = {};
  int pp = 0, pn = 0; // št. pravih na pravem mestu, pravih na napačnem mestu
  for (int i = 0; i < n; ++i) {

    // Izluščimo naslednji žeton obeh zaporedij.
    int zu = ugibanje % m; ugibanje /= m;
    int zp = prava % m; prava /= m;

    // Preverimo, ali se ujemata; če ne, popravimo oba histograma.
    if (zu == zp) ++pp; else ++hu[zu], ++hp[zp]; }

  // Iz histogramov izračunajmo število pravih barv na napačnem mestu.
  for (int b = 0; b < m; ++b) pn += min(hu[b], hp[b]);

  // Oba odgovora, pp in pn, zapakirajmo v eno samo število.
  return pp * (n + 1) + pn;
}

int main()
{
  // Preberimo dosedanja ugibanja in odgovore.
  int k; cin >> n >> m >> k;
  vector<int> prejsnja(k), odgovori(k);
  for (int i = 0; i < k; ++i) {

    // Preberimo zaporedje in ga zapakirajmo v število u.
    int u = 0;
    for (int j = 0; j < n; ++j) { int b; cin >> b; u = u * m + (b - 1); }

    // Preberimo odgovor.
    int pp, pn; cin >> pp >> pn;
    prejsnja[i] = u; odgovori[i] = pp * (n + 1) + pn; }

  // Poglejmo, katera zaporedja so skladna z vsemi dosedanjimi odgovori.
  vector<int> skladna;
  int stVseh = 1; for (int i = 0; i < n; ++i) stVseh *= m;
  for (int z = 0; z < stVseh; ++z) {
    int i = 0; while (i < k && Odgovor(z, prejsnja[i]) == odgovori[i]) ++i;
    if (i == k) skladna.push_back(z); }

  // Ocenimo vse možnosti glede naslednjega ugibanja.
  int najOcena = stVseh + 1, stNaj = -1;
  for (int p = 0; p < stVseh; ++p)
  {
    // Če za naslednje ugibanje vzamemo p, nam množica „skladna“ dosedanjih
    // kandidatov z razpade na podmnožice glede na Odgovor(p, z).
    // Ocena p-ja je velikost največje izmed teh množic.
    int ocena = 0, stKand[(MaxN + 1) * (MaxN + 1)] = {};
    for (int z : skladna) {
      ocena = max(ocena, ++stKand[Odgovor(p, z)]);
      if (ocena > najOcena) break; } // Ta z gotovo ne bo dal najboljše ocene.

    // Zapomnimo si najnižjo oceno in to, pri koliko p-jih smo jo dobili.
    if (ocena < najOcena) najOcena = ocena, stNaj = 1;
    else if (ocena == najOcena) ++stNaj;
  }

  // Izpišimo rezultate.
  cout << skladna.size() << endl << stNaj << endl; return 0;
}
```

2. Dolgovi

Kot vidimo iz omejitev v besedilu naloge, lahko testne primere razdelimo na več skupin z vse večjimi in težjimi primeri. Približno tako lahko razmišljamo tudi o rešitvi; začnimo s preprosto rešitvijo za najmanjše primere in potem poglejmo, kaj bo treba v njej izboljšati.

Pri najmanjših testnih primerih je dovolj že imeti nekakšen seznam oz. tabelo, v kateri hranimo imena ljudi, njihove dolgove in podatek o tem, ali so v skupini ali ne. Poleg tega hranimo v neki spremenljivki tudi število ljudi v skupini. Pri vsakem dogodku se sprehodimo po tabeli in popravimo podatke o prisotnosti v skupini (če je trenutni dogodek prihod ali odhod) oz. o dolgovih (če je trenutni dogodek plačilo). Tako imamo rešitev s časovno zahtevnostjo $O(n \cdot q)$.

Potem imamo skupino testnih primerov, kjer je veliko prihodov in odhodov, vendar vedno le po enega človeka naenkrat; ljudi in plačil pa je malo. Zdaj si ne moremo privoščiti, da bi pri vsakem prihodu in odhodu pregledali celo tabelo. Namesto tabele bi lahko uporabili slovar oz. razpršeno tabelo, v kateri bi bili ključi imena, pripadajoče vrednosti pa dolgovi in podatki o pripadnosti skupini; tako bi lahko dodajali in brisali ljudi v $O(1)$ časa. Toda te rešitve ne bi mogli posplošiti na primere s prefiksnimi operacijami, saj so v razpršeni tabeli različna imena z istim prefiksom preveč razmetana naokrog. Namesto tega si raje pripravimo urejeno tabelo vseh n imen, ki se kdajkoli pojavijo v vhodnih podatkih; pri dodajanju ali brisanju posameznega človeka lahko z bisekcijo poiščemo, kje v tabeli se nahaja. Tako porabimo $O(\log n)$ časa za dodajanje ali brisanje posameznega človeka, pri plačilu pa moramo iti še vedno po celi tabeli in porabimo $O(n)$ časa.

Pri naslednji skupini testnih primerov so prihodi in odhodi še vedno le posamični (ne pa prefiksni), vendar je lahko zdaj tudi plačil veliko, zato si ne moremo več privoščiti, da bi šli pri vsakem plačilu po vseh ljudeh (niti po vseh ljudeh v skupini) in jim popravljali dolgove. Opazimo lahko, da je pri posameznem plačilu ta sprememba dolgá enaka za vse ljudi, ki so takrat v skupini; recimo, da je pri t-tem plačilu ta sprememba enaka $\triangle_t$. Če se je neki posameznik pridružil skupini tik za r-tim plačilom in jo zapustil tik za t-tim, se mu je v tem času dolg spremenil za $\triangle_{r+1} + \triangle_{r+2} + \ldots + \triangle_t$. S tem, ko smo pri naši dosedanji rešitvi pri vsakem plačilu šli po vseh članih skupine in jim popravljali dolgove, smo pri njih pravzaprav postopoma računali takšne vsote po več zaporednih $\triangle_t$. Cenejši način za izračun takšne vsote pa je, da si pripravimo delne (kumulativne) vsote zaporedja $\triangle_t$: naj bo torej s_t vsota prvih t členov zaporedja $\triangle_t$, torej $s_0 = 0$ in $s_t = s_{t-1} + \triangle_t$. Potem lahko vsoto $\triangle_{r+1} + \triangle_{r+2} + \ldots + \triangle_t$ računamo kot $s_t - s_r$; namesto da dolg človeka popravljamo pri vsakem plačilu, ki nastopi, medtem ko je ta človek v skupini, bi torej lahko ob njegovem prihodu odšteli od njegovega dolga s_r in mu nato ob odhodu prišteli s_t. Tako imamo zdaj z vsakim plačilom le $O(1)$ dela, saj moramo le izračunati $\triangle_t$ (deliti znesek plačila s številom ljudi v skupini) in s_t. Posamezno dodajanje ali brisanje še vedno traja $O(\log n)$ časa in skupaj bo časovna zahtevnost takšne rešitve $O(q \log n)$.

V zadnji skupini testnih primerov se pojavijo tudi prefiksna dodajanja in brisanja. Ker imamo imena urejena po abecedi, tvorijo tista, ki se začnejo na neki dani prefiks, strnjeno podzaporedje našega seznama imen; začetek in konec tega podzaporedja lahko poiščemo z bisekcijo. Preveč časa bi nam vzelo, če bi hoteli zdaj vsakemu človeku v tem podzaporedju posebej povečati ali zmanjšati dolg za trenutno vsoto s_t; potrebujemo torej način, da bomo lahko hkrati spremenili dolg več zaporednim ljudem (vsem za enako količino).

Pomagamo si lahko z neke vrste drevesom segmentov; nad tabelo za dolgove posameznih ljudi zgradimo še tabelo za dolgove po dveh, štirih, osmih itd. zaporednih ljudi. Tako imamo skladovnico tabel T_h za $0 \leq h \leq \lfloor \log_2 n \rfloor$. Tabela T_h ima $\lfloor n/2^h \rfloor$ elementov, pri čemer element $T_h[i]$ vsebuje znesek dolga, ki velja za vse ljudi od $i \cdot 2^h$ do $(i+1) \cdot 2^h - 1$. (Pri tem si mislimo, da so indeksi v tabelah od 0 naprej, pa tudi ljudje so oštevilčeni od 0 do $n-1$ v abecednem vrstnem redu.) Da dobimo pravi dolg osebe i, moramo zdaj sešteti po vseh tabelah tiste elemente, ki to osebo pokrivajo, torej $\sum_h T_h[\lfloor i/2^h \rfloor]$.

Element $T_{h+1}[i]$ pokriva natančno isto skupino ljudi kot elementa $T_h[2i]$ in $T_h[2i+1]$ skupaj. Recimo, da želimo povečati dolgove ljudi od L do vključno $D-1$ za s.

Naivna rešitev bi to naredila tako, da bi s prištela elementom $T_0[L], \ldots, T_0[D-1]$. Toda če sta L in D soda, lahko enak učinek dosežemo, če za s povečamo elemente $T_1[L/2], \ldots, T_1[D/2-1]$, torej polovico manj elementov v naslednji tabeli. (Če L ni bil sod, lahko povečamo $T_0[L]$ za s in nato povečamo L za 1, s čimer ta postane sod; in podobno, če D ni bil sod, lahko povečamo $T_0[D-1]$ za s in odtlej zmanjamo D za 1, s čimer postane sod.). Podobno razmišljamo tudi pri tabeli T_1 in tako naprej; pridemo do naslednjega postopka:

postopek PrefiksniDogodek(L, D, s):
  $h := 0$;
  while $L < D$:
    če je L lih: povečaj $T_h[L]$ za s; $L := L + 1$;
    če je D lih: $D := D - 1$; povečaj $T_h[D]$ za s;
    $L := L/2$; $D := D/2$; $h := h + 1$;

Časovna zahtevnost tega postopka je $O(\log n)$, saj pri vsakem h spremeni največ dva elementa tabele T_h; tako lahko prefiksno dodajanje ali brisanje izvedemo v samo $O(\log n)$ časa.

Prefiksni dogodki nam zapletejo rešitev še zaradi nečesa drugega: naloga pravi, da pri takem dogodku pridejo oz. odidejo le tisti ljudje, ki se jim ime začne na dani prefiks *in ki so nekoč prej že bili dodani v skupino*. To pomeni, da naš gornji postopek, ki je spremenil dolgove *vseh* ljudi od L do $D-1$, pravzaprav ni bil dober; moral bi spremeniti dolgove samo tistih, ki so bili kdaj prej že v skupini, toda če bi hoteli to preveriti za vsakega od ljudi na tem intervalu, bi postopek spet trajal $O(n)$ časa. Pomagamo si lahko takole: gornji postopek obdržimo brez sprememb, toda vzdržujmo tudi neko tabelo, ki nam za vsakega človeka pove, ali je bil že kdaj v skupini. Naloga zagotavlja, da prvi prihod človeka v skupino gotovo nastopi s posamičnim dodajanjem, ne s prefiksnim; pri posamičnem dodajanju torej preverimo, ali je ta človek že bil v skupini; če ni bil, izračunajmo njegov dosedanji dolg (ki se je nabral zaradi dosedanjih prefiksnih dohodkov in ki je v resnici popolnoma neupravičen, saj se na tega človeka dosedanji prefiksni dohodki niso nanašali) po prej omenjeni formuli $\sum_h T_h[\lfloor i/2^h \rfloor]$ (kjer je i številka človeka, ki zdaj prvič prihaja v skupino) in ga nato odštejmo od $T_0[i]$; tako bo njegov dolg prišel na 0, kar je zdaj tudi njegova prava vrednost.

Naslednji zaplet, povezan s prefiksnimi dogodki, se nanaša na plačila. Ko nekdo plača znesek z, se dolgovi članov skupine povečajo za z/k, kjer je k število ljudi v skupini. Vrednost k moramo torej znati vzdrževati tudi pri prefiksnih dodajanjih in brisanjih. Spomnimo se, da prefiksna operacija ne doda v skupino (ali pobriše iz nje) *vseh* ljudi od L do $D-1$, pač pa le tiste od njih, ki so bili kdaj prej že v skupini. Znati moramo torej hitro izračunati, koliko ljudi s takega intervala je bilo kdaj prej že v skupini. Tudi za to lahko uporabimo segmentno drevo; poleg tabel T_h imejmo še tabele Z_h, pri čemer $Z_h[i]$ pove, koliko ljudi od $i \cdot 2^h$ do $(i+1) \cdot 2^h - 1$ je bilo doslej že v skupini. Ko neki človek i prvič vstopi v skupino, povečamo pri vsakem h element $Z_h[\lfloor i/2^h \rfloor]$ za 1. Postopek PrefiksniDogodek pa je treba dopolniti tako, da kadarkoli poveča neko vrednost $T_h[i]$ za s, mora tudi povečati (če gre za prefiksno dodajanje) ali zmanjšati (če gre za brisanje) k (torej število ljudi v skupini) za $Z_h[i]$.

Še en zaplet pa nastopi na koncu, ko hočemo izpisati rezultate. Za ljudi, ki so takrat še v skupini, smo v segmentnem drevesu sicer njihov dolg zmanjšali za vsoto s_i, ki je veljala ob njihovem prihodu, nismo pa ga še povečali za vsoto s_i, ki velja zdaj na koncu. To moramo torej narediti zdaj, vendar moramo za to vedeti, kateri ljudje so v skupini. V ta namen lahko naše segmentno drevo še dopolnimo: poleg tabel T_h in Z_h imejmo še tabele P_h in O_h s časi prihodov oz. odhodov. Ko naš postopek PrefiksniDogodek poveča vrednost $T_h[i]$ za s, mora zdaj tudi vpisati trenutni „čas“ (zaporedno številko dogodka, pri katerem smo) v element $P_h[i]$ (če gre za prefiksni prihod) oz. $O_h[i]$ (če gre za odhod). Na koncu lahko za vsakega človeka i izračunamo čas zadnjega prihoda kot $\max_h P_h[\lfloor i/2^h \rfloor]$ in podobno čas zadnjega odhoda; če je zadnji prihod kasnejši kot zadnji odhod, je ta človek zdaj še v skupini, sicer pa ne.

Tako smo dobili rešitev, ki podpira tudi prefiksne dogodke, pri tem pa še vedno porabi le $O(\log n)$ časa za vsak dogodek, skupaj $O(q \log n)$:

```
#include <vector>
```

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

int main()
{
  int q; cin >> q;

  // Preberimo vse dogodke in si pripravimo slovar imen, omenjenih v njih.
  struct Dogodek { char op; string ime; int znesek; };
  vector<Dogodek> dogodki(q);
  unordered_map<string, int> slovarImen;
  for (auto &Q : dogodki) {
    string op; cin >> op >> Q.ime; Q.op = op[0];
    if (Q.op == '<') cin >> Q.znesek;
    if (Q.op == '+' && Q.ime.back() != '*') slovarImen.emplace(Q.ime, -1); }

  // Uredimo imena po abecedi in shranimo njihove indekse v slovar.
  int n = slovarImen.size();
  vector<string> imena; imena.reserve(n);
  for (auto &pr : slovarImen) imena.emplace_back(pr.first);
  sort(imena.begin(), imena.end());
  for (int i = 0; i < n; ++i) slovarImen[imena[i]] = i;

  // Pripravimo prazno drevo segmentov.
  struct Vozlisce { int znanih = 0, prihod = -1, odhod = -1; double dolg = 0; };
  vector<vector<Vozlisce>> drevo; drevo.emplace_back(n);
  while (drevo.back().size() > 1) drevo.emplace_back(drevo.back().size() / 2);
  int H = drevo.size() - 1;
  double vsotaDolgov = 0; int stVSkupini = 0; // s_t in k

  // Obdelajmo vse dogodke.
  for (int qi = 0; qi < q; ++qi)
  {
    auto &Q = dogodki[qi]; string &ime = Q.ime;

    if (Q.op == '<') { // Ali je ta dogodek plačilo?
      // Določimo indeks osebe, ki je plačnik pri tem dogodku.
      int L = lower_bound(imena.begin(), imena.end(), ime) - imena.begin();

      // Zmanjšajmo plačnikov dolg in povečajmo kumulativno vsoto dolgov.
      drevo[0][L].dolg -= Q.znesek; vsotaDolgov += double(Q.znesek) / stVSkupini;
      continue; }

    // Sicer je ta dogodek prihod ali odhod.
    bool prefiks = (ime.back() == '*'); if (prefiks) ime.pop_back();
    bool prihod = (Q.op == '+');
    double sprememba = prihod ? -vsotaDolgov : vsotaDolgov;

    // Naslednja funkcija vpiše v vozlišče spremembo dolga in čas zadnjega
    // prihoda ali odhoda ter popravi števec ljudi v skupini.
    auto Popravi = [&] (Vozlisce &v) { v.dolg += sprememba;
      stVSkupini += (prihod ? 1 : -1) * v.znanih; (prihod ? v.prihod : v.odhod) = qi; };

    // Poglejmo, kateri razpon imen pokriva ta dogodek.
    int L = lower_bound(imena.begin(), imena.end(), ime) - imena.begin();
    if (! prefiks) {

      // Če prvič dodajamo to osebo, dosedanji prefiksni dogodki zanjo niso veljali
      // in moramo njen dosedanji dolg postaviti na 0.
      if (prihod && ! drevo[0][L].znanih)
        for (int h = H; h >= 0; --h) if (int i = L >> h; i < drevo[h].size()) {
          auto &v = drevo[h][i]; ++v.znanih; sprememba -= v.dolg; }

      // Dogodek vpliva le na eno osebo; popravimo dolg v njenem listu.
      Popravi(drevo[0][L]); }
```

```
    else {
      int D = lower_bound(imena.begin(), imena.end(), ime + char('z' + 1)) - imena.begin();
      // Prefiksni dogodek vpliva na vse ljudi od vključno L do vključno D - 1.
      // Na vsakem nivoju drevesa moramo popraviti največ dve vozlišči.
      for (int h = 0; h <= H && L < D; ++h, L >>= 1, D >>= 1) {
        if (L & 1) Popravi(drevo[h][L++]);
        if (D & 1) Popravi(drevo[h][--D]); } }
  }
  // Porinimo vse podatke v liste drevesa.
  for (int h = H; h > 0; --h) for (int i = 0; i < 2 * (n >> h); ++i) {
    auto &v = drevo[h - 1][i], &p = drevo[h][i / 2];
    v.dolg += p.dolg; v.prihod = max(v.prihod, p.prihod); v.odhod = max(v.odhod, p.odhod); }
  // Izpišimo rezultate.
  for (int i = 0; i < n; ++i) { auto &v = drevo[0][i];
    cout << imena[i] << "  " << setprecision(9)
         << v.dolg + (v.prihod > v.odhod ? vsotaDolgov : 0) << endl; }
  return 0;
}
```

3. Breakout

Žogica začne svojo pot na koordinatah $(a - \frac{1}{2}, 0)$, vedno se premika diagonalno in ko se odbije od stranice kakšne opeke, se to vedno zgodi na razpolovišču stranice. Zato za koordinati žogice tudi pri vsakem odboju velja, da je ena od njiju celo število, druga pa je na pol poti med dvema sosednjima celima številoma: $(x, y + \frac{1}{2})$ ali $(x + \frac{1}{2}, y)$.

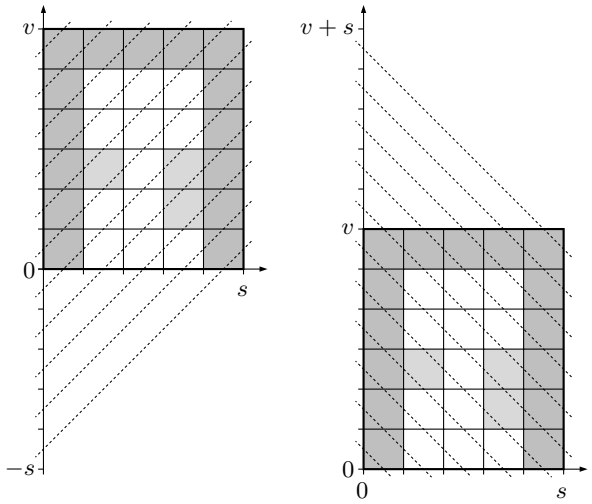
Smer gibanja žogice lahko opišemo s parom števil $(\triangle_x, \triangle_y)$, pri čemer je vsako od njiju enako bodisi 1 bodisi -1. Če je $\triangle_x = \triangle_y$, se žogica premika po *rastoči* diagonali, torej taki premici, na kateri imajo vse točke enako razliko $y - x$; lahko jo opišemo z enačbo oblike $y - x = C + \frac{1}{2}$, pri čemer je C celo število. V našem primeru, ker nas zanimajo le koordinate z območja $0 \le x \le s$ in $0 \le y \le v$, gre lahko C pri takih premicah od $-s$ do $v - 1$ (glej sliko 1).

Če pa je $\triangle_x = -\triangle_y$, se žogica premika po *padajoči* diagonali, torej premici, na kateri imajo vse točke enako vsoto $y + x$; lahko jo opišemo z enačbo oblike $y + x = C + \frac{1}{2}$ za neko celo število C, ki gre zdaj lahko od 0 do $s + v - 1$.

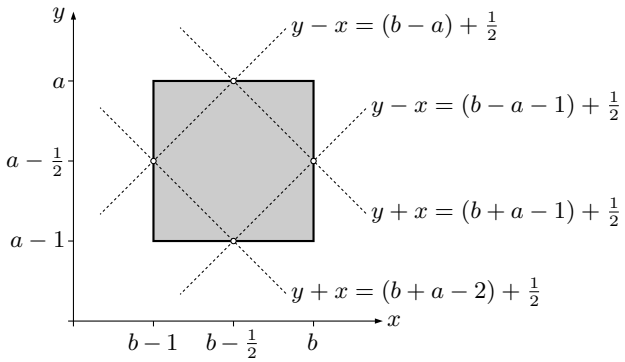
V poštev pride torej $s + v$ rastočih in prav toliko padajočih premic. Za vsako od njih bi bilo dobro imeti urejen seznam opek, skozi katere gre ta premica; urejene naj bodo v takem vrstnem redu, v kakršnem jih premica doseže od leve proti desni. To pomeni, da jih moramo urediti naraščajoče po x-koordinati; paziti pa moramo na možnost, da ista premica pri istem x zadene dve opeki, ki sta ravno ena nad drugo, torej (x, y) in $(x, y + 1)$. V takem primeru mora priti najprej na vrsto opeka (x, y), če je premica rastoča, oz. najprej opeka $(x, y + 1)$, če je premica padajoča. (Za urejanje opek po x-koordinati lahko uporabimo neke vrste urejanje s štetjem; ustvarimo s seznamov in nato vsako opeko (x, y) dodamo na x-ti seznam. Tako za urejanje porabimo le $O(N)$ časa, kjer je $N = 2(v - 1) + s + k$ število vseh opek, tudi tistih v stenah.)

Takšne sezname lahko uporabimo za iskanje naslednje opeke, od katere se bo žogica odbila. Če poznamo trenutni položaj in smer gibanja žogice, vemo tudi, po kateri premici se bo gibala. Na njej bi lahko z bisekcijo iskali x-koordinato žogice in videli, med katerima dvema opekama leži; odbila se bo torej od leve oz. desne od teh dveh opek, odvisno od tega, ali je $\triangle_x$ enak -1 ali $+1$. Ta rešitev ima dve slabosti: ena slabost je, da bo občasno treba pobrisati kakšno opeko (ker opeka po določenem številu odbojev razpade) in ta operacija bo draga, ker bomo morali imeti sezname predstavljene s tabelami ali vektorji (da bomo imeli do opek naključen dostop, ki ga potrebujemo za bisekcijo). Druga slabost je, da bomo imeli pri vsakem odboju po $O(\log n)$ dela zaradi bisekcije, odbojev pa je lahko pri največjih testnih primerih toliko, da bo to že prepočasi.

Namesto tega naj bo vsak seznam raje veriga členov, povezanih s kazalci (*linked list*); brisanje iz takega seznama vzame le $O(1)$ časa. Vsako opeko sekajo štiri premice (glej sliko 2), torej pripada štirim takim seznamom in v neki tabeli si bomo zapomnili člene, ki predstavljajo to opeko v tistih štirih seznamih. Ko žogica zadene opeko, to pomeni, da smo v seznamu, ki predstavlja premico, po kateri se je žogica zdaj premikala,



Slika 1. Primer premic, ki nas zanimajo pri mreži širine $s = 5$ in višine $v = 6$. Leva slika kaže rastoče diagonale $y - x = C + \frac{1}{2}$ za $-s \le C < v$, desna slika pa padajoče diagonale $y + x = C + \frac{1}{2}$ za $0 \le C < v + s$.



Slika 2. Primer opeke z zgornjim levim kotom (a, b) in štirih diagonalnih premic, ki jo sekajo. Krožci kažejo razpolovišča stranic, kjer se lahko žogica odbije od te opeke.

prišli do člena, ki predstavlja tisto opeko. Iz prejšnjega položaja žogice, položaja opeke ter smeri gibanja žogice lahko tudi določimo stranico, od katere se je žogica odbila; potem tudi poznamo koordinate točke odboja (to je ravno razpolovišče tiste stranice) in novo smer gibanja (pri odboju od vodoravne stranice se $\triangle_x$ pomnoži z -1, $\triangle_y$ pa ostane nespremenjen; pri odboju od navpične stranice je ravno obratno). Zdaj poznamo novi položaj in smer gibanja žogice, torej lahko tudi določimo, po kateri premici se bo premikala po odboju. Spomnimo se, da imamo za vsako opeko shranjene člene v seznamih, ki jim pripada; za trenutno opeko torej poznamo njen člen na premici, po kateri se bo gibala žogica po odboju; naslednja opeka, od katere se bo odbila, je torej bodisi prejšnji bodisi naslednji člen tega seznama (odvisno od tega, ali je novi $\triangle_x$ negativen ali pozitiven, saj vemo, da so seznami urejeni po x). Tako se torej lahko v $O(1)$ časa premaknemo na tisti naslednji ali prejšnji člen in s tem izvemo, od katere opeke se bo žogica odbila v naslednjem odboju. Vsak odboj nam torej vzame le $O(1)$ časa, ne pa $O(\log n)$ kot pri bisekciji.

Če pri iskanju opeke za naslednji odboj vidimo, da naslednjega oz. prejšnjega člena v seznamu (ki predstavlja premico, po kateri se bo žogica gibala) sploh ni, to pomeni, da bo žogica odletela iz mreže in lahko simulacijo njenega gibanja končamo. Da izračunamo, v kateri točki bo ušla iz mreže, moramo pogledati, katero stranico mreže bo prej zadela, navpično (levo, če je $\triangle_x < 0$, oz. desno, če je $\triangle_x > 0$) ali vodoravno (spodnjo, če je $\triangle_y < 0$, oz. zgornjo, če je $\triangle_y > 0$).

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

struct Opeka;
```

```
typedef list<Opeka *> Premica;

struct Opeka
{
  int x, y, k;
  int naslNaX; // naslednja opeka s to x-koordinato
  Premica::iterator it[2][2]; // prvi indeks: 0 = padajoča, 1 = rastoča;
                              // drugi indeks: 0 = spodnja, 1 = zgornja

  void Dodaj(Premica &premica, int rastoca, int zgornja) {
    // Pazimo na to, kako morajo biti na tej premici urejene opeke z istim x.
    // Morda moramo trenutno opeko vriniti na predzadnje mesto, ne dodati na konec.
    auto kam = premica.end();
    if (! premica.empty()) { auto b = premica.back();
      if ((x == b->x) && ((rastoca == 1) == (y < b->y))) --kam; }
    it[rastoca][zgornja] = premica.insert(kam, this); }

  // Vrne parameter C ene od premic skozi to opeko.
  int C(int rastoca, int zgornja) const { return y + (rastoca ? -x : x - 1) - (zgornja ? 0 : 1); }
};

int main()
{
  int v, s, n, a; scanf("%d %d %d %d", &s, &v, &n, &a);

  // Preberimo opeke.
  int N = n + 2 * (v - 1) + s;
  vector<Opeka> opeke; opeke.reserve(N); opeke.resize(n);
  for (Opeka &O : opeke) scanf("%d %d %d", &O.x, &O.y, &O.k);

  // Dodajmo opeke, ki tvorijo stene.
  enum { K = 100 };
  for (int y = 1; y < v; ++y) { opeke.push_back({1, y, K}); opeke.push_back({s, y, K}); }
  for (int x = 1; x <= s; ++x) opeke.push_back({x, v, K});

  // Uredimo opeke po x-koordinati.
  vector<int> prvaNaX(s + 1, -1);
  for (int i = 0; i < N; ++i) { auto &O = opeke[i];
    O.naslNaX = prvaNaX[O.x]; prvaNaX[O.x] = i; }

  // rastoce[C] predstavlja premico y - x = C + 1/2; padajoce[C] pa y + x = C + 1/2.
  vector<Premica> rastoce_(v + s), padajoce_(v + s);
  auto rastoce = rastoce_.begin() + s, padajoce = padajoce_.begin();

  // Dodajmo jih na premice; na vsaki premici bodo opeke urejene naraščajoče po x.
  // Če imata dve opeki na isti premici isti x, premica zadene prej tisto z nižjim y, če je rastoča,
  // oz. tisto z višjim y, če je padajoča. Na to pazimo pri dodajanju, ker imamo opeke urejene
  // le po x, ne pa tudi po y.
  for (int x = 0; x <= s; ++x) for (int i = prvaNaX[x]; i >= 0; ) { auto &O = opeke[i];
    for (int r = 0; r < 2; ++r) for (int z = 0; z < 2; ++z)
      O.Dodaj((r ? rastoce : padajoce)[O.C(r, z)], r, z);
    i = O.naslNaX; }

  // Začnimo s simulacijo.
  int x2 = 2 * a - 1, y2 = 0, dx = 1, dy = 1;

  // Žogica je na (x2/2, y2/2). To pomeni, da leži na naraščajoči premici
  // y - x = (y2 - x2) / 2 = (y2 - x2 - 1) / 2 + 1/2. Poiščimo prvo opeko, ki jo zadene.
  int C = (y2 - x2 - 1) / 2;
  Premica *premica = &rastoce[C];
  auto opeka = premica->begin();
  while (opeka != premica->end() && (*opeka)->x <= (x2 + 1) / 2) ++opeka;

  int stRazbitih = 0;
  while (true)
  {
    Opeka &O = **opeka;

    // Od katere stranice te opeke se žogica odbije?
    enum { S, J, V, Z } str =
      (dx == 1) ? (dy == 1 ? (O.C(1, 1) == C ? Z : J) : (O.C(0, 1) == C ? S : Z)) :
```

```
                (dy == 1 ? (O.C(0, 1) == C ? V : J) : (O.C(1, 1) == C ? S : V));
    // Izračunajmo smer gibanja po odboju.
    if (str == Z || str == V) dx = -dx; else dy = -dy;

    // Izračunajmo koordinate točke, kjer se žogica odbije od opeke.
    x2 = 2 * O.x - (str == Z ? 2 : str == V ? 0 : 1);
    y2 = 2 * O.y - (str == S ? 0 : str == J ? 2 : 1);

    // Določimo premico, po kateri se bo gibala po odboju.
    int rastoca = (dx == dy) ? 1 : 0;
    C = (y2 + (rastoca ? -x2 : x2) - 1) / 2;
    premica = &(rastoca ? rastoce : padajoce)[C];

    // Določimo naslednjo opeko na njeni poti.
    opeka = O.it[rastoca][(O.C(rastoca, 1) == C) ? 1 : 0];
    bool konec = false; if (dx > 0) konec = (++opeka == premica->end());
    else { if (opeka == premica->begin()) konec = true; else --opeka; }

    // Pobrišimo trenutno opeko, če pri tem odboju razpade.
    if (--O.k == 0) { ++stRazbitih;
      for (int r = 0; r < 2; ++r) for (int z = 0; z < 2; ++z)
        (r ? rastoce : padajoce)[O.C(r, z)].erase(O.it[r][z]); }

    if (konec) break;
  }
  // Od trenutnega položaja naprej se žogica ne odbija več;
  // izračunajmo, pri kateri koordinati zapusti igralno polje.
  int premik = min((dx == 1 ? 2 * s - x2 : x2), (dy == 1 ? 2 * v - y2 : y2));
  x2 += dx * premik; y2 += dy * premik;

  printf("%d\n%d%s %d%s\n", stRazbitih, x2 / 2, (x2 % 2 ? ".5" : ""),
                                        y2 / 2, (y2 % 2 ? ".5" : ""));
  return 0;
}
```

Še nekaj podrobnosti o gornji implementaciji: za predstavitev premic smo uporabili razred list iz standardne knjižnice; na člene teh seznamov kažejo iteratorji. Za predstavitev trenutnega položaja žogice namesto njenih koordinat hranimo dvakratnika teh koordinat (spremenljivki x2 in y2), da se izognemo delu z ne-celimi števili (razen čisto na koncu, pri izpisu). Pri dodajanju opek v sezname pazimo na to, da sicer gledamo opeke po naraščajočih x, niso pa nujno pravilno urejene po y, zato moramo včasih vriniti opeko na predzadnje mesto seznama, včasih pa jo dodati prav na konec seznama (metoda Opeka::Dodaj).

4. Pijansko urejanje

Poskusimo za začetek ugotoviti, na katerem podstavku stoji kateri zaboj. Po prvi zamenjavi (ne glede na to, s katerim podstavkom) izvemo, kateri zaboj zdaj stoji na podstavku 0. Od tu naprej lahko nadaljujemo v zanki; recimo, da za podstavke od 0 do $k-1$ že vemo, kateri zaboji so na njih, in da bi radi to izvedeli še za podstavek k. Če naročimo zamenjavo s podstavkom $k-1$, bomo v resnici dobili na podstavek 0 enega od zabojev s podstavkov $k-2$, $k-1$ ali k (pri $k=1$ seveda možnost $k-2$ odpade). (1) Če je bil to eden od zabojev s podstavkov $k-2$ ali $k-1$, ga bomo zlahka prepoznali, saj smo pred to zamenjavo za podstavke do vključno $k-1$ že vedeli, kateri zaboji so na njih. V tem primeru nismo z zamenjavo izvedeli ničesar novega in moramo poskusiti še enkrat. (2) Če pa nam zamenjava pripelje na podstavek 0 neki zaboj, ki ga še ne poznamo, lahko zaključimo, da je ta zaboj moral priti s podstavka k in da je torej zdaj na podstavek k prišel tisti zaboj, ki je bil prej na podstavku 0. V tem primeru zdaj poznamo vsebino prvih k podstavkov (in ne več le prvih $k-1$) in lahko v nadaljevanju povečamo k za 1.

Sčasoma pridemo s tem postopkom do konca zaporedja in potem za vsak podstavek vemo, kateri zaboj je na njem — ali obratno: za vsak zaboj vemo, na katerem podstavku se nahaja; pravzaprav je koristno imeti obe vrsti podatkov (spodnji program ima v ta namen globalni spremenljivki v in kje). Teh podatkov tudi ni težko vzdrževati pri vsaki izvedeni zamenjavi.

Za urejanje tabele lahko uporabimo znani postopek urejanja z izbiranjem (*selection sort*): najprej premaknemo zaboj $n-1$ na podstavek $n-1$, nato zaboj $n-2$ na podstavek $n-2$ in tako navzdol proti vse manjšim številkam zabojev in podstavkov. Recimo, da smo zaboje od $k+1$ do $n-1$ že postavili na pravo mesto; zdaj poskusimo to narediti z zabojem k. Tudi zanj, tako kot za vse zaboje, točno vemo, kje je; recimo, da je na podstavku p. Če naročimo zamenjavo s podstavkom $p-1$, bo na podstavek 0 prišel eden od zabojev s podstavkov $p-2$, $p-1$ in p, torej obstaja možnost, da bo to ravno zaboj k (če ni, moramo pač z enako zamenjavo poskusiti še večkrat). Ko imamo zaboj k na podstavku 0, pa naročimo zamenjavo s podstavkom $k-1$; tako pride zaboj k na enega od podstavkov $k-2$, $k-1$ in k. Če ni prišel ravno na podstavek k, ga moramo spraviti nazaj na podstavek 0 in nato poskusiti znova; prej ali slej bo prišel na podstavek k.

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

vector<int> v, kje;  // v[i] = številka zaboja na podstavku i;
                     // kje[i] = številka podstavka, na katerem je zaboj i.

void Zamenjaj(int k)
{
  printf("%d\n", k); fflush(stdout);
  int stari0 = v[0], novi0; scanf("%d", &novi0);

  // Če smo poznali stari položaj zaboja, ki je zdaj prišel na podstavek 0,
  // potem je to tudi novi položaj zaboja, ki je bil prej na podstavku 0.
  if (kje[novi0] >= 0) v[kje[novi0]] = stari0;
  if (stari0 >= 0) kje[stari0] = kje[novi0];

  // V vsakem primeru pa zdaj vemo, kje je zaboj, ki je prišel na podstavek 0.
  v[0] = novi0; kje[novi0] = 0;
}

int main()
{
  int n; scanf("%d", &n);
  v.resize(n, -1); kje.resize(n, -1);

  // Ugotovimo, kateri zaboj je na katerem podstavku.
  for (int k = 1; k < n; )
  {
    int x = v[0]; Zamenjaj(k - 1);                                    // (1)

    // Zaboj x je bil pred to zamenjavo na podstavku 0, zato je zdaj na enem
    // od podstavkov k - 2, k - 1 ali k. Toda za podstavka k - 2 in k - 1 smo
    // vedeli, kaj je bilo tam pred zamenjavo; če bi x prišel tja, bi zato
    // tudi zanj zdaj vedeli, da je tam. Če torej zdaj za x ne vemo, kje je,
    // to pomeni, da je moral priti na podstavek k.
    if (x >= 0 && kje[x] < 0) { kje[x] = k; v[k++] = x; }
  }

  // Uredimo zaboje z izbiranjem.
  for (int k = n - 1; k >= 1; --k) while (v[k] != k) {

    // Spravimo zaboj k na podstavek 0. Pri tem pazimo, da ne bomo
    // spremenili zaboja desno od njega, kajti ta je morda že na pravem mestu.
    while (v[0] != k) Zamenjaj(max(0, kje[k] - 1));                   // (2)

    // Poskusimo spraviti zaboj k na podstavek k.
    Zamenjaj(k - 1); }                                                // (3)

  printf("-1\n"); return 0;
}
```

Razmislimo o tem, koliko zamenjav izvede ta rešitev. V vrstici (1) čakamo pri vsakem k na to, da pride na podstavek 0 zaboj s podstavka k in ne s podstavka $k-1$ ali $k-2$. To se torej v vsakem poskusu zgodi z verjetnostjo $1/3$, zato potrebujemo v povprečju 3 poskuse. V vrstici (3), ko je zaboj k enkrat na podstavku 0, imamo $1/3$ možnosti, da bo

v naslednji zamenjavi prišel na podstavek k (in ne na $k-1$ ali $k-2$), torej potrebujemo v povprečju tri take poskuse; toda pred vsakim od njih moramo zaboj k najprej spraviti na podstavek 0, za kar potrebuje vrstica (2) povprečno tri zamenjave, da res pobere zaboj s podstavka $p =$ `kje[k]` in ne $p-1$ ali $p-2$. Skupaj imamo torej v vrsticah (2) in (3) povprečno po $3 \cdot (3+1)$ zamenjav pri vsakem zaboju k, da ga končno spravimo na podstavek k. Ker mora tako pri (1) kot pri (2) in (3) iti k od 1 do $n-1$, imamo vsega skupaj približno $15(n-1)$ zamenjav.[2] Ker gre pri tej nalogi n do 100, nam torej ni treba skrbeti, da bi prekoračili mejo 10 000, kolikor znaša največje dovoljeno število zamenjav.

5. L-sistem

Recimo, da je niz s dolg n črk: $s = c_1 c_2 \dots c_n$. Potem je niz $f^k(s)$, ki nas pri tej nalogi zanima, sestavljen iz n kosov oblike $f^k(c_i)$, torej $f^k(s) = f^k(c_1) f^k(c_2) \dots f^k(c_n)$. Vsaka pojavitev p-ja (ali kakšnega njegovega anagrama — tega v nadaljevanju ne bomo kar naprej ponavljali) v $f^k(s)$ potem bodisi v celoti leži znotraj enega od kosov $f^k(c_i)$ bodisi prečka vsaj eno mejo med dvema zaporednima kosoma, na primer $f^k(c_i)$ in $f^k(c_{i+1})$.

Tiste, ki v celoti ležijo znotraj enega kosa, lahko torej preštejemo tako, da za vsako črko c naše abecede preštejemo pojavitve p-ja v $f^k(c)$, nato pa to seštejemo po vseh črkah niza s. Pojavitve p-ja v $f^k(c)$ pa lahko preštejemo tako, da upoštevamo, da je $f^k(c) = f^{k-1}(f(c))$; rešiti moramo torej enak problem kot pri $f^k(s)$, le s $k-1$ namesto k in z nizom $f(c)$ namesto s. Tako se nakazuje rešitev z rekurzijo (ali, še bolje, z dinamičnim programiranjem).

Ostane še vprašanje, kako prešteti tiste pojavitve p-ja v $f^k(s)$, ki ne ležijo v celoti znotraj enega kosa $f^k(c_i)$. Če na primer pojavitev prečka mejo med $f^k(c_i)$ in $f^k(c_{i+1})$, to pomeni, da mora prvih nekaj znakov niza p ležati na koncu kosa $f^k(c_i)$, preostanek niza p pa na začetku kosa $f^k(c_{i+1})$. Na posamezni strani meje med $f^k(c_i)$ in $f^k(c_{i+1})$ je lahko največ $|p|-1$ znakov p-ja, kajti vsaj en znak mora še ostati na drugi strani meje (sicer bo pojavitev ležala v celoti znotraj enega kosa). Za štetje takšnih pojavitev p-ja, ki prečkajo meje med kosi, je torej pomembnih le prvih in zadnjih $|p|-1$ znakov posameznega kosa. To je koristno zato, ker so kosi pri tej nalogi lahko zelo dolgi (eksponentno v odvisnosti od k), niz p pa ni dolg; celega kosa $f^k(c_i)$ ne moremo izračunati, njegovih prvih in zadnjih $|p|-1$ znakov pa lahko.

Pišimo $q = |p| - 1$ in definirajmo funkcijo $\gamma(x)$ takole:

$$\gamma(x) = \begin{cases} x[:q] \,\#\, x[-q:], & \text{če je } |x| > 2q \\ x & \text{sicer.} \end{cases}$$

Od niza x torej funkcija $\gamma(x)$ obdrži le prvih in zadnjih q znakov, vse vmes pa zamenja s posebnim ločilnim znakom # (ki se drugače v naših nizih ne pojavlja). Če je x dolg kvečjemu $2q$ znakov, pa funkcija γ obdrži celega.

Tako je torej $\gamma(x)$ tisto pri nizu x, kar je pomembno za štetje pojavitev p-ja na mejah med kosi. Nas bo to seveda najbolj zanimalo za nize $x = f^k(c_i)$; definirajmo torej $g_k(s) = \gamma(f^k(s))$. Niza $g_k(s)$ ne moremo računati tako, da bi najprej izračunali $f^k(s)$ in ga potem oklestili, saj je lahko $f^k(s)$ za kaj takega predolg. Toda ker je $f^k(s) = f^k(c_1) \dots f^k(c_n)$, je prvih q znakov niza $f^k(s)$ ravno prvih q znakov niza $f^k(c_1)$; če pa je le-ta krajši od q, moramo vzeti še prvih nekaj znakov niza $f^k(c_2)$ in tako naprej. Podoben razmislek velja tudi za zadnjih q znakov. Vidimo torej, da je

$$g_k(s) = \gamma(g_k(c_1) g_k(c_2) \dots g_k(c_n)).$$

Tako lahko nize oblike $g_k(s)$ (ki so prijetno kratki: vsi so krajši od $2|p|$ znakov) računamo, ne da bi morali imeti kdaj opravka z (eksponentno dolgimi) nizi oblike $f^k(s)$.

Ko imamo niz $g_k(s)$ pred sabo, lahko v njem preštejemo pojavitve p-ja;[3] in s tem smo prešteli tudi tiste pojavitve p-ja v $f^k(s)$, ki prečkajo kakšno od mejá med dvema

[2] V resnici še malo manj, ker je bil naš razmislek na več mestih nekoliko pesimističen. Pri naših poskusih z 10^9 naključnimi testnimi primeri velikosti $n = 100$ je bilo povprečno število zamenjav približno 1411,6; pri nobenem od teh primerov ni bilo treba več kot 2185 zamenjav in nikoli ni šlo z manj kot 879 zamenjavami.

[3] Kako prešteti pojavitve p-ja — in, ne pozabimo, njegovih anagramov — v nekem daljšem nizu x? Preprosta možnost je, da se po nizu x premikamo z oknom dolžine $|p|$ znakov in v neki tabeli vzdržujemo

sosednjima kosoma. Če je kak kos slučajno krajši od $2|p|-1$ znakov, je zanj $g_k(c_i) = f^k(c_i)$ in smo v $g_k(s)$ dobili tudi tiste pojavitve p-ja v $f^k(s)$, ki v celoti ležijo znotraj kosa $f^k(c_i)$. Paziti moramo torej, da takih pojavitev ne bomo šteli dvojno — (1) ko seštevamo po vseh kosih tiste pojavitve, ko v celoti ležijo znotraj enega kosa, in (2) ko štejemo pojavitve p-ja v $g_k(s)$. Kose, pri katerih $g_k(c_i)$ ne vsebuje znaka # na sredi, lahko torej pri (1) preskočimo, saj bomo pojavitve p-ja v njih prešteli pri (2).

Število pojavitev p-ja v $f^k(s)$ označimo s $h_k(s)$. Zdaj imamo vse, kar potrebujemo, da zapišemo psevdokodo našega postopka:

podprogram KORAK(s, k, g_k, h_k):
vhod: niz $s = c_1 \dots c_n$, število k, vrednosti $g_k(t_i)$ in $h_k(t_i)$ za vse črke abecede;
izhod: vrednosti $g_k(s)$ in $h_k(s)$;

$x :=$ prazen niz; $H := 0$;
for $i := 1$ **to** n:
  if $g^k(c_i)$ vsebuje # **then** $H := H + h_k(c_i)$;
  dodaj $g_k(c_i)$ na konec niza x;
$H := H +$ število pojavitev p-ja v nizu x;
vrni $g_k(s) = \gamma(x)$ in $h_k(s) = H$;

Zdaj torej znamo izračunati g_k in h_k za daljše nize iz njunih vrednosti za posamezne črke abecede; slednje pa dobimo tako, da upoštevamo, da je $f^k(t_i) = f^{k-1}(f(t_i))$:

za vsako črko t_i naše abecede:
  $g_0(t_i) = t_i$;
  if $p = t_i$ **then** $h_0(t_i) := 1$ else $h_0(t_i) := 0$;
for $r := 0$ **to** $k-1$:
  za vsako črko t_i naše abecede:
    za $x = f(t_i)$ pokliči KORAK(x, r, g_r, h_r);
    vrednosti $g_r(x)$ in $h_r(x)$, ki ju je KORAK vrnil,
      si zapomni kot $g_{r+1}(t_i)$ in $h_{r+1}(t_i)$;
pokliči KORAK(s, k, g_k, r_k) in vrni $h_k(s)$, ki jo je izračunal;

V praksi moramo pri izračunu paziti še na to, da moramo na koncu vrniti število pojavitev po modulu M in da je pametno zato že med izračunom ves čas delati le z ostanki po deljenju s M, saj bi bilo pravo število pojavitev lahko preveliko in bi prekoračili obseg celoštevilskih spremenljivk.

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

string p; int pd, a, M; // pd je dolžina niza p

// Struktura Opis opisuje vse, kar moramo vedeti o nizih oblike f^r(x):
// stPojavitev = h_r(x), število pojavitev p-ja v nizu;
// s = g_r(x) = γ(f^r(x)) = prvih in zadnjih |p| − 1 znakov niza f^r(x), vmes pa
// ločilni znak (število a). Če je niz dolg ≤ 2(|p| − 1) znakov, je v s prisoten v celoti.
struct Opis { int stPojavitev; string s; };

// Izračuna opis niza f^r(x), če so znani opisi f^r(t_i) za posamezne črke abecede.
void Korak(const vector<Opis>& opisi, const string &x, Opis &O)
{
  O.stPojavitev = 0; string &fx = O.s; fx.clear();
  // V „fx“ pripravimo stik nizov γ(f^r(x[i])) in seštejmo pojavitve p-ja znotraj nizov f^r(x[i])
  // za posamezne črke niza x. Če je neki f^r(x[i]) dovolj kratek, da dobimo celega v fx,
```

„histogram“ — za vsako črko imamo število pojavitev te črke v oknu. Teh števil ni težko vzdrževati: ko se okno premakne za en znak naprej, pride na desni ena črka vanj, na levi pa pade ena črka iz njega, torej moramo popraviti največ dva elementa v histogramu. Vnaprej preštejmo tudi pojavitve vsake črke v nizu p, pa bomo lahko sproti preverjali, koliko črk se v oknu pojavlja manjkrat kot v p, koliko pa večkrat kot v p. Če ni nobene take črke, lahko zaključimo, da se vsaka črka v oknu pojavlja natanko tolikokrat kot v p, torej je v oknu nek anagram p-ja.

```
  // pojavitev p-ja v njem ne upoštevajmo, saj jih bomo dobili kasneje, ko bomo pregledali fx.
  for (char c : x) { const Opis &Oc = opisi[c]; fx += Oc.s;
    if (Oc.s.length() == 2 * pd - 1 && Oc.s[pd - 1] == a)
      O.stPojavitev = (O.stPojavitev + Oc.stPojavitev) % M; }

  // Upoštevajmo pojavitve p-ja, ki prečkajo kakšno od meja med f^r(x[i]) in f^r(x[i+1]).
  // Po nizu fx se premikamo z oknom dolžine |p|; stCrk[c] je razlika med številom pojavitev
  // črke c (za c = 0, ..., a; c = a predstavlja ločilni znak) v oknu in v vzorcu p.
  // „stPremalo" pove, pri koliko c-jih je stCrk[c] < 0; „stPrevec" pa, pri koliko c-jih
  int fsd = fx.length(), stPremalo = 0, stPrevec = 0;              // je stCrk[c] > 0.
  static int stCrk[27]; for (int i = 0; i <= a; ++i) stCrk[i] = 0;
  for (char c : p) if (--stCrk[c] == -1) ++stPremalo;
  for (int i = 0; i < fsd; ++i) {

    // Na desni je v okno fx[i - pd + 1, ..., i] prišel znak fx[i].
    int x = ++stCrk[fx[i]]; if (x == 0) --stPremalo; else if (x == 1) ++stPrevec;

    // Na levi je iz okna izpadel znak fx[i - pd].
    if (i >= pd) { x = --stCrk[fx[i - pd]];
      if (x == 0) --stPrevec; else if (x == -1) ++stPremalo; }

    // Ali je v oknu ravno anagram p-ja?
    if (stPremalo == 0 && stPrevec == 0) ++O.stPojavitev; }

  O.stPojavitev %= M;

  // Če je f^r(x) daljši od 2(|p| - 1), obdržimo le prvih in zadnjih |p| - 1 črk,
  // vmes pa damo ločilni znak (število a). Tako dobimo γ(f^r(x)).
  if (fsd > 2 * (pd - 1))
    fx = fx.substr(0, pd - 1) + char(a) + fx.substr(fsd - pd + 1);
};

int main()
{
  // Preberimo abecedo.
  string abeceda; getline(cin, abeceda); a = abeceda.length();

  // Pripravimo funkcijo, s katero bomo znake abecede prevedli v števila od 0 do a - 1.
  string xlat(26, ' '); for (int i = 0; i < a; ++i) xlat[abeceda[i] - 'a'] = i;
  auto Prevedi = [&xlat] (string &s) { for (char &c : s) c = xlat[c - 'a']; };

  // Preberimo nize f(t_i) za vse črke abecede in jih prevedimo.
  vector<string> f(a); for (auto &fa : f) { getline(cin, fa); Prevedi(fa); }

  // Preberimo ostale podatke.
  string s; int k; cin >> s >> k >> M >> p;
  Prevedi(s); Prevedi(p); pd = p.length();

  // Pripravimo opise nizov f^0(t_i).
  vector<Opis> opisi(a), noviOpisi(a);
  for (int i = 0; i < a; ++i) opisi[i] = { (pd == 1 && p[0] == i ? 1 : 0), string(1, char(i)) };

  // Izračunajmo opise f^r(t_i) za r = 1, ..., k.
  for (int r = 0; r < k; ++r) {
    for (int i = 0; i < a; ++i) Korak(opisi, f[i], noviOpisi[i]);
    swap(opisi, noviOpisi); }

  // Izračunajmo opis f^k(s) in izpišimo rezultat.
  Opis O; Korak(opisi, s, O);
  cout << O.stPojavitev << endl; return 0;
}
```

Naloge so sestavili: mastermind — Urban Duh; stonoge, sedežni red — Bor Grošelj Simić; žabe, breakout — Tomaž Hočevar; planinarjenje — Mark Martinec; snežinke — Polona Novak; neprevidni poeti — Ella Potisek; dolgovi — Jakob Schrader; varnostno kopiranje — Jure Slak; semafor, iskanje kvadrata — Borut in Peter Žnidar; luči, pijansko urejanje, L-sistem — Janez Brank.

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