20. tekmovanje ACM v znanju računalništva za srednješolce

22. marca 2025

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.");
   }
}
```

20. tekmovanje ACM v znanju računalništva za srednješolce

22. marca 2025

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 in oddaj liste odgovorni osebi v učilnici, kjer si tekmoval. 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.

Na tekmovanju lahko uporabljaš tudi svoje zapiske in literaturo (v papirnati obliki).

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 (npr. Visual Studio Code) 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 in rezultati bodo objavljeni na `https://rtk.ijs.si/`.

Vabimo te, da ob koncu tekmovanja izpolniš tudi **anketo**:

`https://www.1ka.si/a/9474358e`

1. Natakarica

Natakarica nosi hrano in pijačo od šanka do miz. Rada bi se čim manj sprehajala, zato vsakič na svoj pladenj naloži toliko, kolikor gre. Če je pladenj pretežek, ji bo padel in potem bo imela samo še več dela. Postreči mora več miz, z vsake mize pa je dobila več naročil (od različnih ljudi). Na mizo prinese vsa naročila naenkrat, mizam pa jih prinaša v istem vrstnem redu, kot so bila naročila oddana. Ena miza nikoli ne naroči več, kot natakarica lahko nese.

Napiši program, ki izračuna, kolikokrat bo morala natakarica opraviti pot od šanka do miz glede na naročila, ki jih je dobila.

Vhodni podatki: v prvi vrstici sta m, število miz, in t, teža, ki jo lahko natakarica nese. V vsaki od naslednjih m vrstic je najprej n, število naročil s te mize, sledi pa mu n števil, ki predstavljajo težo posameznega naročila.

Izhodni podatki: izpiši število poti.

Tvoj program lahko bere s standardnega vhoda in piše na standardni izhod ali pa bere iz datoteke `vhod.txt` in piše v datoteko `izhod.txt` (kar ti je lažje). Predpostavi, da je lahko vhodnih podatkov zelo veliko; rešitve, ki poskušajo hraniti vse vhodne podatke hkrati v glavnem pomnilniku, lahko dobijo pri tej nalogi največ 16 točk od 20.

Primer vhoda:	Pripadajoči izhod:

`3 10`
`4 1 2 3 1`
`2 2 2`
`1 2`
	`2`

Komentar: prva miza ima skupno težo naročil 7, druga pa 4, kar je skupaj 11. Zato natakarica najprej prinese prvi mizi, potem pa se vrne k šanku in prinese še naročila za drugo in tretjo mizo, ki imata skupno težo 6.

2. Uravnotežena prehrana

Nacionalni inštitut za javno zdravje na svojih straneh objavlja smernice uravnoteženega prehranjevanja, kjer med drugim piše, koliko kalorij (kcal), beljakovin, maščob in ogljikovih hidratov je priporočeno zaužiti na dan. Ker imajo ljudje različne kalorične potrebe glede na spol, mišično maso ali starost, so primerne vrednosti podane kot intervali (tudi spodnja in zgornja meja intervala štejeta za primerni). Pri tej nalogi bomo za primerne dnevne vrednosti vzeli sledeče:

- kalorije: med 1600 in 3000 kcal,
- beljakovine: med 40 in 150 gramov,
- maščobe: med 35 in 100 gramov,
- ogljikovi hidrati: med 220 in 500 gramov.

Prijatelj te je prosil, ali bi preveril, kako kvalitetno se je včeraj prehranjeval. Vestno si je zapisal sestavine vsega, kar je pojedel, in s pomočjo etiket na izdelkih že zabeležil, koliko kalorij, beljakovin, maščob in ogljikovih hidratov je zaužil. Prosi te, da **napišeš program**, ki prebere te podatke in mu pove, ali je dnevni vnos primeren, in če ni, ali lahko morda eno izmed sestavin izpusti, da bi s tem njegova prehrana postala primerna.

Trije primeri spodaj kažejo, v kakšni obliki bo tvoj program dobil vhodne podatke in v kakšni obliki naj izpiše svoj odgovor. Prva vrstica vhoda pove, koliko sestavin sledi, te pa so nato navedene vsaka v svoji vrstici. Vsaka od teh vrstic vsebuje ime sestavine in količino kalorij, beljakovin, maščob in ogljikovih hidratov, ločene z vejicami. Posamezna vrstica vhodne datoteke je dolga največ 100 znakov. Tvoj program naj izpiše, katere vrednosti so zunaj primernih meja (če sploh katere). Če je mogoče vrednosti spraviti znotraj primernih meja tako, da eno od sestavin črtamo, naj izpiše vse sestavine, s katerimi je to mogoče doseči.

Tvoj program lahko bere s standardnega vhoda in piše na standardni izhod ali pa bere iz datoteke `vhod.txt` in piše v datoteko `izhod.txt` (kar ti je lažje).

(*Nadaljevanje na naslednji strani.*)

Vhod 1:

```
8
kos kruha,75,4,1,14
maslo,45,0,5,0
marmelada,50,0,0,13
riz,400,8,2,90
losos,380,50,20,0
brokoli,140,6,0,22
olje,90,0,10,0
cips,300,4,20,30
```

Izhod 1:

```
kalorije: vnos 1480 izven mej [1600, 3000]
ogljikovi hidrati: vnos 169 izven mej [220, 500]
```

Vhod 2:

```
10
kos kruha,75,4,1,14
maslo,45,0,5,0
marmelada,50,0,0,13
riz,400,8,2,90
losos,380,50,20,0
brokoli,140,6,0,22
olje,90,0,10,0
cips,300,4,20,30
cokolada,330,5,20,36
2 kosa kruha,150,8,2,28
```

Izhod 2:

```
Primerna prehrana.
```

Vhod 3:

```
11
kos kruha,75,4,1,14
maslo,45,0,5,0
marmelada,50,0,0,13
riz,400,8,2,90
losos,380,50,20,0
brokoli,140,6,0,22
olje,90,0,10,0
cips,300,4,20,30
cokolada,330,5,20,36
2 kosa kruha,150,8,2,28
sir,350,25,28,2
```

Izhod 3:

```
mascobe: vnos 108 izven mej [35, 100]
Lahko odstranis samo losos.
Lahko odstranis samo olje.
Lahko odstranis samo sir.
```

3. Razpolavljanje torte

V protokolarnem objektu Brdo bodo gostili zabavo z mnogimi pomembnimi osebnostmi, kjer bodo razrezali tudi veliko torto. Da bi bili čim natančnejši, so se tega lotili tako, da torto najprej razrežejo na dve polovici, nato pa tidve polovici razpolovijo naprej in tako dalje, do največ 10 razpolavljanj.

Po nekaj časa organizatorje zanima, koliko torte je še ostalo v kuhinji. Šef kuhinje začne govoriti: „Tam imamo pol in pol od pol od pol in pol od pol od pol od pol in pol od pol od pol od pol od pol od pol . . .“ Prvi pomočnik si je vse lepo zapisal, ampak ne zna oceniti, koliko torte jim je dejansko preostalo v kuhinji.

Napiši program ali podprogram oz. funkcijo, ki iz opisa šefa kuhinje izračuna, kakšen delež celote ta dejansko predstavlja. Opis šefa kuhinje je podan tako, da so našteti vsi kosi torte, ločeni z „`in`“, vsak posamezen kos pa je podan glede na celo torto: „`pol`“ predstavlja polovico, „`pol od pol`“ četrtino itd. Delež torte, ki je ostal v kuhinji, izpiši v odstokih, zaokroženih na 1 % natančno (vseeno je, ali se zaokroža navzgor ali navzdol). Vhodni podatki bodo taki, da bo delež vedno med 0 in 100 %. Podrobnosti tega, v kakšni obliki tvoj (pod)program dobi vhodni niz in kako vrne ali izpiše rezultat, si izberi sam in jih v svoji rešitvi tudi opiši.

Nekaj primerov:

`pol` ⟶ 50 %
`pol od pol` ⟶ 25 %
`pol in pol` ⟶ 100 %
`pol od pol od pol in pol in pol od pol` ⟶ 88 %
`pol od pol od pol od pol od pol od pol od pol` ⟶ 0 %

4. Ultrazvok

Z ultrazvokom opazujemo različna tkiva. Da se bomo bolje znašli, nas zanima, kako debelo je posamezno tkivo.

Opiši postopek (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki iz slike za vsako vrsto tkiva ugotovi, kakšna je njegova največja in najmanjša debelina na tej sliki. Slika prikazuje vertikalen prerez, sestavljen iz različnih tkiv; predstavljamo si jo lahko kot pravokotno karirasto mrežo, v vsakem polju te mreže pa je neka mala črka angleške abecede, ki pove, kakšne vrste tkivo je tam. Vsaka črka abecede predstavlja eno vrsto tkiva.

Za vsako vrsto tkiva, ki se pojavlja na vhodni sliki, izpiši črko, ki jo predstavlja, in dve števili: najmanjšo in največjo debelino. Za debelino tkiva se upošteva maksimalni neprekinjen del tega tkiva (strnjena skupina enakih črk, ki stojijo ena nad drugo vse v istem stolpcu in ki zgoraj in spodaj mejijo na druge črke in/ali rob slike).

Podrobnosti tega, v kakšni obliki tvoj postopek ali (pod)program dobi vhodne podatke in izpiše rezultate, si izberi sam in jih v svoji rešitvi tudi opiši.

Primer vhodne slike:

```
aaaaaaaaaaaaaa
xxxxxxaaxxaxxx
xxxcccxxxxxxxx
xxccccxcxxxxxx
ccceeccccccxxc
ccceeccccccccc
ccccccccccccc
ddddccddddddd
```

Možen pripadajoči izpis:

```
a 1 2
x 1 4
c 2 5
d 1 1
e 2 2
```

5. Prijave na izlet

Tvoj prijatelj Primož bo čez dva tedna organiziral super-kul izlet, za katerega bo naročil avtobus. Ker pričakuje, da bo zanimanja za ta izlet ogromno, ga skrbi, da na avtobusu ne bo dovolj sedežev. Prosi te za pomoč pri vodenju evidence prijav — da bo vedel, komu lahko reče, da ima zanj prostor na avtobusu, in koga bo moral posaditi na čakalno vrsto, ker zanj trenutno ni prostora.

Tega pa kot programer seveda ne boš delal ročno, zato **napiši program**, ki bo za Primoža urejal prijave.

Program naj na začetku prebere število s, ki pove, koliko sedežev ima avtobus, ki ga bo Primož naročil.

Po tem pa naj spremlja prijave in odjave, ki jih bo Primož pisal na naslednji način:

- „`+` ⟨*ime*⟩“ pomeni, da se je prijavila oseba z imenom ⟨*ime*⟩. V odgovor naj tvoj program izpiše, ali je ta oseba dobila prost sedež na avtobusu in če ne, na katerem mestu v čakalni vrsti je.
- „`-` ⟨*ime*⟩“ pomeni, da se je odjavila oseba z imenom ⟨*ime*⟩. Če se s tem sprosti kakšno mesto na avtobusu, izpiši, kdo dobi sedež, ki se je sprostil.
- „`?`“ pomeni, da Primoža zanima, kdo vse je prijavljen na izlet. V tem primeru naj tvoj program izpiše, kdo vse ima trenutno prostor na avtobusu, koliko je na avtobusu prostih mest in koliko ljudi je še v čakalni vrsti (in zanje ni prostora v avtobusu).

Program lahko bere vhod v neskočni zanki ali pa do konca vhodnih podatkov (EOF), karkoli ti je lažje.

Primer vhoda:

```
5
+ Primoz
+ Marko
+ Milan
+ Branko
?
+ Gregor
+ Martin
+ Dejan
+ David
- Martin
- Milan
?
- Marko
?
```

Možen pripadajoči izhod:

```
Primoz dobi prost sedez.
Marko dobi prost sedez.
Milan dobi prost sedez.
Branko dobi prost sedez.
Primoz Marko Milan Branko, prosti sedezi: 1, v vrsti: 0.
Gregor dobi prost sedez.
Martin je na 1. mestu v vrsti.
Dejan je na 2. mestu v vrsti.
David je na 3. mestu v vrsti.
Dejan se premakne iz cakalne vrste na avtobus.
Primoz Marko Branko Gregor Dejan, prosti sedezi: 0, v vrsti: 1.
David se premakne iz cakalne vrste na avtobus.
Primoz Branko Gregor Dejan David, prosti sedezi: 0, v vrsti: 0.
```

(*Nadaljevanje na naslednji strani.*)

Komentar: oblika izpisa ni pomembna, prav tako ne vrstni red imen ljudi, ki imajo trenutno prostor na avtobusu; pomembno je le, da izpis obsega vse potrebne podatke (mesto v vrsti prijavljenih, ki pristanejo v čakalni vrsti; kdo zavzame prosto mesto, ko se nekdo odjavi; kdo vse sedi na avtobusu, koliko je še prostora na njem in koliko ljudi je v čakalni vrsti).

Prav tako ni pomembno, v kakšnem vrstnem redu izpišeš, kdo vse ima prostor na avtobusu (samo, da so imena prava).

Predpostaviš lahko, da so imena enolična: ne bo se prijavilo več ljudi z istim imenom. Tudi se ne bo zgodilo, da se kdo, ki je trenutno že prijavljen, prijavil še enkrat (ne da bi se vmes odjavil) ali da bi se odjavil kdo, ki trenutno ni prijavljen. Imena so dolga največ 20 znakov, sestavljena pa so izključno iz malih in velikih črk angleške abecede ter števk (in vsak prijavljeni ima natanko eno ime).

20. tekmovanje ACM v znanju računalništva za srednješolce

22. marca 2025

NALOGE ZA DRUGO SKUPINO

Pri prvih štirih nalogah lahko odgovore 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 in oddaj liste odgovorni osebi v učilnici, kjer si tekmoval. Pri delu si lahko pomagaš s prevajalniki in razvojnimi orodji, ki so na voljo na tvojem računalniku, vendar bomo tvoje odgovore pri teh štirih nalogah 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 pri nalogah z avtomatskim ocenjevanjem. Če oddaš pri isti nalogi prek računalnika več rešitev, se upošteva **zadnja** od njih.

Pri peti nalogi pa moraš svojo rešitev oddati prek računalnika, naš ocenjevalni sistem pa jo bo samodejno prevedel in preizkusil na več testnih primerih. Če pri tej nalogi oddaš več rešitev, se upošteva **najboljšo** od njih. Podrobnejša navodila v zvezi s to nalogo so na začetku besedila naloge.

Na tekmovanju lahko uporabljaš tudi svoje zapiske in literaturo (v papirnati obliki).

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2025-2/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Uporabniško ime in geslo za Putko sta enaki kot za računalnike. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek gumba „Postavi vprašanje“ pri besedilu posamezne naloge. Na forumu, do katerega prideš s tem gumbom, bomo objavljali 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 pojasnila. 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 in rezultati bodo objavljeni na `https://rtk.ijs.si/`.

Vabimo te, da ob koncu tekmovanja izpolniš tudi **anketo**:
`https://www.1ka.si/a/9474358e`

1. Omrežnina

Pripravlja se nov pravilnik o zaračunavanju električne energije. Izračun se opira na redno merjenje porabe moči v vsakem gospodinjstvu. Del računa za električno energijo je tudi omrežnina, ki je odvisna od dogovorjene moči d in števila presežkov k te moči (k je torej število meritev, pri katerih je bila izmerjena moč strogo višja od dogovorjene moči d). V predlogu pravilnika je cena omrežnine enaka $d \cdot M + k \cdot P$, kjer sta M in P znana faktorja. Dosežene moči se obravnavajo v 15-minutnih intervalih. Zagotovljeno je, da sta M in P nenegativna (torej $M \geq 0$ in $P \geq 0$), pa tudi za d si je treba izbrati neko nenegativno vrednost ($d \geq 0$).

Za posamezno gospodinjstvo imamo podatke o doseženi električni moči v posameznem 15-minutnem intervalu za celo leto, kar znese 35 040 meritev (365 dni $\cdot$ 24 ur $\cdot$ 4 meritve), in zanj želimo na podlagi teh podatkov nastaviti dogovorjeno moč tako, da bo cena omrežnine čim nižja. **Napiši podprogram** (funkcijo), ki bo sprejel faktorja M in P ter seznam s 35 040 meritvami dosežene moči v zadnjem letu in bo izračunal optimalno dogovorjeno moč d, pri kateri bi bila zaračunana omrežnina najnižja. Upoštevaj, da je v Sloveniji približno 860 000 gospodinjstev, zato bo tvoj podprogram klican velikokrat in mora biti čim bolj učinkovit.

2. Trojice

Danih je n nizov, dolgih po k znakov. V njih nastopajo le male črke `a`, `b` in `c`. **Napiši program**, ki prešteje, na koliko načinov lahko izmed teh n nizov izberemo tri nize tako, da bodo na vsakem od k mest imeli bodisi vsi trije enako črko bodisi vsi različne črke (ne pa dva enako črko in tretji neko drugo črko).

Vhodni podatki: v prvi vrstici vhoda sta celi števili n (število nizov) in k (dolžina nizov). Nato sledi n vrstic z nizi, vsak v svoji vrstici. Vsak od teh nizov je dolg k črk in v njih nastopajo le črke `a`, `b` in `c`.

Vsi nizi na vhodu si bodo paroma različni.

Omejitve vhodnih podatkov: $1 \leq n \leq 5000$ in $1 \leq k \leq 100$.

Izhodni podatki: v edini vrstici izhoda izpiši, na koliko načinov je mogoče izbrati tri nize v skladu z omejitvami iz besedila naloge.

Tvoj postopek naj bo čim bolj učinkovit, da bo deloval tudi za velike n.

Primer vhoda:

```
13 4
acaa
ccab
cabc
abbc
bcbc
cbca
caac
cccb
aaac
bacc
abcc
bbbc
cacb
```

Pripadajoči izhod:

```
6
```

Komentar: če nize v tem primeru oštevilčimo po vrsti od 1 do 13, ugotovimo, da je mogoče tri nize izbrati na šest načinov. To so trojice: $\{3, 4, 5\}$, $\{2, 3, 6\}$, $\{1, 5, 8\}$, $\{3, 9, 10\}$, $\{5, 7, 11\}$ in $\{1, 12, 13\}$.

3. Beg

Anton se lovi s svojim prijateljem Borom. Ima nekaj časa, da nabere prednost, Bor pa ga bo lovil s pomočjo drona. Anton se lahko na vsakem koraku premakne za 1 meter proti severu, jugu, vzhodu ali zahodu. Ko bo Anton naredil nekaj korakov, bo proti njemu Bor poslal dron v ravni črti. Dron je hiter, zato se v tem času Anton ne bo mogel več premakniti. Poleg tega ima dron skoraj prazno baterijo, zato lahko preleti največ k metrov. Najmanj koliko korakov mora Anton narediti, da ga dron ne bo dosegel?

Napiši program ali podprogram oz. funkcijo, ki kot vhod dobi naravno število k in pravokoten zemljevid, kjer vsako polje predstavlja kvadratni meter igrišča (glej primer spodaj). Igrišča Anton ne sme zapustiti. Zemljevid je poravnan s smermi neba (sever je gor, vzhod desno). Na začetku se Bor in Anton nahajata na polju, označenem z „`x`“. Če Anton lahko gre na neko polje, je označeno s piko „`.`“, sicer pa je na tem mestu ovira in je označena z „`#`“ (ovire preprečujejo pot le Antonu, dron pa lahko leti čeznje). Izračunaj, najmanj koliko korakov mora Anton narediti, da bo dronu zmanjkalo energije, preden ga doseže. Če to ni mogoče, naj program kot odgovor izpiše -1. Podrobnosti tega, kako tvoj (pod)program dobi vhodne podatke in izpiše rezultat, si izberi sam in jih v svoji rešitvi tudi opiši.

Opomba: še enkrat poudarimo, da dron leti v ravni črti. Razdaljo med dvema poljema v ravni črti izračunamo po Pitagorovem izreku. Če se polji razlikujeta za d_x v smeri vzhod–zahod in d_y v smeri sever–jug, je razdalja med njima enaka $\sqrt{d_x^2 + d_y^2}$.

Primer. Recimo, da je zemljevid igrišča takšen:

```
#....
..##.
...#.
..##.
#..x.
```

Če je $k = 2$, potem je dovolj, da gre Anton dvakrat levo in enkrat gor. Tako bo razdalja enaka $\sqrt{5}$, kar je več od 2. Samo dvakrat levo ne bi bilo dovolj, saj bi dron to polje še dosegel (komajda, pa vendarle). Kakorkoli bo Anton naredil manj kot tri korake, bo razdalja manjša ali enaka 2, zato je odgovor 3 (korake).

Če je $k = 3$, gre lahko enkrat desno in trikrat gor, odgovor je 4. Za $k = 10$ ne more dovolj daleč in je odgovor -1.

4. Tovarna

V tovarni žebljev v Butalah imajo n delavcev in n strojev za izdelavo žebljev, ki so postavljeni v vrsto eden za drugim. Ker so stroje kupili v različnih letih in od različnih proizvajalcev, pa niso vsi stroji enako zmogljivi. Prvi stroj lahko izdela x_1 žebljev vsako minuto, drugi stroj lahko izdela x_2 žebljev na minuto in tako naprej.

V tej tovarni delo poteka na naslednji način: vsako jutro se delavci postavijo vsak pred svoj stroj in k strojem nosijo železne opilke, iz katerih bodo izdelani žeblji; iz vsakega opilka se izdela en žebelj. Delavci niso vsi enako hitri; prvi delavec lahko prinese y_1 opilkov na minuto, drugi delavec lahko prinese y_2 opilkov na minuto in tako naprej. Ker lahko nekateri delavci prinesejo več opilkov, kot njihov stroj naredi žebljev, so se dogovorili, da lahko opilke nesejo tudi k sosednjim strojem (torej stroju takoj na levi in stroju takoj na desni). Posamezni delavec lahko opilke, ki jih je prinesel, poljubno razdeli med te stroje (svoj stroj in njegovega levega in desnega soseda).

Opiši postopek (ali napiši program ali (pod)program oz. funkcijo, če ti je lažje), ki bo izračunal, največ koliko žebljev lahko tovarna proizvede v eni minuti. Kot vhodne podatke tvoj postopek dobi števila $n, x_1, \ldots, x_n, y_1, \ldots, y_n$. V svojem odgovoru tudi **dobro utemelji**, zakaj tvoj postopek res izračuna pravilni rezultat. Upoštevaj, da prvi delavec nima levega soseda, torej lahko nese opilke le k svojemu stroju in stroju neposredno desno od svojega, ter da zadnji delavec nima desnega soseda, torej lahko nese opilke le do svojega stroja ter do stroja na njegovi neposredni levi.

5. Chordpro

Ta naloga se ocenjuje avtomatsko, ne pa ročno kot prve štiri. Izvorna koda tvoje rešitve mora biti napisana v enem od naslednjih programskih jezikov: pascal, C, C++, C#, java, python ali rust. Ocenjevalni strežnik bo tvoj program prevedel in pognal na več testnih primerih. Tvoj program se sme na posameznem testnem primeru izvajati največ pol sekunde in sme porabiti največ 256 MB pomnilnika.

Naloga ima 20 testnih primerov, vsak je vreden po eno točko. Če oddaš pri tej nalogi več rešitev, se upošteva tista, ki doseže največ točk.

Tvoj program naj bere vhodne podatke s standardnega vhoda in izpiše svoje rezultate na standardni izhod. Besedilo naloge natančno določa obliko (format) vhodnih in izhodnih podatkov. Tvoj program lahko predpostavi, 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.

Chordpro je format, v katerem lahko pišemo besedila in akorde za pesmarice (knjige s pesmimi). Pesem v tem formatu je videti kot vsaka druga, le da ima na želenih mestih vstavljene akorde (ki kitaristu povedo, kaj naj igra). Akord je niz znakov v oglatih oklepajih (ki se drugače v besedilu pesmi ne pojavljajo), ki se napiše direktno pred del besedila, nad katerim se mora kasneje pokazati (brez oglatih oklepajev).

Napiši program, ki prevede besedilo v formatu chordpro v berljivo pesmarico. To pomeni, da vsako vrstico na vhodu, ki ima vsaj en akord, prepišeš v dve vrstici; v prvi naj bodo akordi (ki so poravnani glede na položaj v prvotnem besedilu), v drugi pa sam verz pesmi (brez akordov). Vrstice, ki nimajo akordov, pa brez sprememb prepišeš na izhod.

Vhodni podatki: v prvi vrstici se nahaja število vrstic n, temu pa sledi n vrstic v formatu chordpro.

Omejitve vhodnih podatkov: vedno bo veljalo $n \leq 1000$. Poleg tega lahko predpostaviš, da bodo akordi dovolj kratki, da se v končnem izhodu nobena dva ne bosta prekrivala. Posamezna vrstica vhodnih podatkov je dolga največ 100 znakov.

Akordi so sestavljeni izključno iz črk angleške abecede, števk in znakov „/“ ter „#“ in so vedno obdani z oglatimi oklepaji „[“ in „]“. Ostanek besedila pa poleg črk in števk vsebuje tudi presledke in znake `.!?(){}/-:;,`.

Testni primeri pri tej nalogi se razlikujejo po dodatnih omejitvah:

- (13 primerov) Nobena vrstica ne bo prazna in v vsaki vrstici bo vsaj en akord.
- (4 primeri) Nobena vrstica ne bo prazna.
- (2 primera) V vsaki neprazni vrstici bo vsaj en akord.
- (1 primer) Ni dodatnih omejitev.

Izhodni podatki. Vrstice, ki ne vsebujejo nobenega akorda, izpiši brez sprememb take, kakršne so bile na vhodu. Namesto vsake vhodne vrstice z vsaj enim akordom pa izpiši najprej eno vrstico z akordi in eno vrstico z verzom.

Ker se presledkov na koncu vrstice ne vidi, jih bo tudi tekmovalni sistem ignoriral.

Primer vhoda:

```
8
[Am]Muce bele, [E]muce crne, [Am]muce zlate i[C]n srebrne
[Dm]vec ne smejo klepetati, [Esus4]morajo ta[E]koj zaspati[Am].

[Am]Zlato sonce [E]gre za morje, [Am]bele sanje [C]cez obzorje.
[Dm]Sanje so velika skleda, [Esus4]pol iz mleka, [E]pol iz meda.[Am]

Ko jim sanje zadisijo, bele muce brz zaspijo
in za njimi tudi crne in se zlate in srebrne.
```

(*Nadaljevanje na naslednji strani.*)

Pripadajoči izhod:

```
Am         E          Am          C
Muce bele, muce crne, muce zlate in srebrne
Dm                      Esus4    E          Am
vec ne smejo klepetati, morajo takoj zaspati.

Am          E             Am         C
Zlato sonce gre za morje, bele sanje cez obzorje.
Dm                      Esus4         E           Am
Sanje so velika skleda, pol iz mleka, pol iz meda.

Ko jim sanje zadisijo, bele muce brz zaspijo
in za njimi tudi crne in se zlate in srebrne.
```

Komentar. Opazimo, da se akordi ravnajo po znaku, pred katerega so postavljeni. V prvem verzu je akord `Am` v vhodu pred besedo `muce` in zato tudi v izhodu nad to besedo. Akord `Am` na koncu drugega verza druge kitice je na koncu besedila in se zato tako tudi izpiše.

20. tekmovanje ACM v znanju računalništva za srednješolce

22. marca 2025

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, python ali rust.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2025-3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi svoje rešitve 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 pod „Pogovor o nalogi“ v okvirju „Osnovne informacije“ desno od 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 podrobnosti o tem, katere prevajalnike uporablja ocenjevalni strežnik in s kakšnimi nastavitvami, glej stran `https://putka-rtk.acm.si/info/`). Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku in na ocenjevalnem strežniku), prenosnih računalnikov, prenosnih telefonov itd.

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

Vabimo te, da ob koncu tekmovanja izpolniš tudi **anketo**:

`https://www.1ka.si/a/9474358e`

Ocenjevanje

Vsaka naloga ti lahko prinese od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih. Pri prvi nalogi je testnih primerov 10 in vsak je vreden po 10 točk, pri drugi in tretji nalogi pa jih je po 20 in vsak je vreden po 5 točk. Četrta in peta naloga imata točkovanje po podnalogah, kjer dobi program vse točke za posamezno podnalogo, če pravilno reši vse testne primere tiste podnaloge, sicer pa pri tej podnalogi dobi 0 točk.

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

nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

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

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

Primer naloge (ne šteje k tekmovanju)

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

Primer vhoda:

```
123 456
```

Ustrezen izhod:

```
5790
```

Primeri rešitev:

- V pascalu:

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

- V C++:

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

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

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

- 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-ju:

```
#include <stdio.h>

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

- V pythonu:

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

- 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));
  }
}
```

- V rustu:

```
use std::io;

fn main() {
  let mut s = String::new();
  io::stdin().read_line(&mut s);
  let t: Vec<&str> = s.trim().split(' ').collect();
  let i = t[0].parse::<i32>().unwrap();
  let j = t[1].parse::<i32>().unwrap();
  println!("{}", 10 * (i + j));
}
```

20. tekmovanje ACM v znanju računalništva za srednješolce

22. marca 2025

NALOGE ZA TRETJO SKUPINO

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

1. Trki

Simulirati moraš potovanje delca po neskončni karirasti mreži, kjer so na nekaterih poljih ovire. Tvoj program bo na začetku dobil začetne koordinate (x, y) in začetno hitrost (v_x, v_y), torej hitrost v smeri x in hitrost v smeri y. Hitrost je lahko tudi negativna, kar pomeni, da se delec premika proti manjšim x- oz. y-koordinatam. Premike simuliramo na naslednji način, dokler hitrost delca ne pade na $v_x = v_y = 0$:

1. Delec najprej premaknemo v smeri x za v_x, če pa je na tej poti kakšna ovira, se ta premik konča že tik pred prvo oviro.
2. Delec nato premaknemo v smeri y za v_y; v primeru ovire se ta premik konča že tik pred prvo oviro (tako kot v prvi točki).
3. Hitrosti v_x in v_y zaradi upora razpolovimo in (če se deljenje ne izide) zaokrožimo proti 0.

Če se delec s pozitivno hitrostjo v smeri x zaleti v prvem koraku zgoraj opisanega postopka v oviro na (a, b), se bo nehal premikati v smeri x tik pred njo, torej na $(a-1, b)$. Podobno velja za ostale smeri.

Napiši program, ki kot vhodne podatke dobi cela števila x, y, v_x, v_y in izračuna končni položaj delca. Za ugotavljanje položaja ovir lahko tvoj program ocenjevalnemu sistemu postavlja vprašanja, kjer za določeno pravokotno območje vpraša, ali je na njem kakšna ovira ali ne.

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 vrstico, v kateri bodo štiri cela števila x, y, v_x in v_y, ločena s po enim presledkom.
2. Nato lahko izvedeš 0 ali več poizvedb. Posamezno poizvedbo izvedeš tako, da izpišeš na standardni izhod vrstico oblike „`?` *a b c d*“ (brez narekovajev), pri čemer so a, b, c in d cela števila, ločena s presledki (zanje mora veljati $-10^9 \leq a \leq c \leq 10^9$ in $-10^9 \leq b \leq d \leq 10^9$). Nato preberi s standardnega vhoda vrstico, v kateri dobiš odgovor sistema na tvojo poizvedbo; to bo niz „`DA`“ ali „`NE`“ (brez narekovajev), ki pove, ali obstaja kakšna ovira z x-koordinato od vključno a do vključno c in y-koordinato od vključno b do vključno d.
3. Na koncu izpiši na standardni izhod vrstico, v kateri naj bosta dve celi števili, ločeni s presledkom — končna x- in y-koordinata delca. Po tem naj se tvoj program preneha izvajati.

Opozorilo: po vsaki izpisani vrstici splakni standardni izhod (*flush*), da bodo podatki res sproti prišli do ocenjevalnega sistema (navodila za splakovanje v različnih programskih jezikih najdeš na `https://putka-rtk.acm.si/info/`).

(*Nadaljevanje na naslednji strani.*)

Omejitve: $-10^9 \le x \le 10^9$; $-10^9 \le y \le 10^9$; $-10^9 \le x + 2v_x \le 10^9$; $-10^9 \le y + 2v_y \le 10^9$.

Pri prvih 30 % testnih primerov bo veljalo tudi $|x| \le 400$, $|y| \le 400$, $|x + 2v_x| \le 400$, $|y + 2v_y| \le 400$.

Pri naslednjih 20 % testnih primerov bo veljalo tudi $|x| \le 800$, $|y| \le 800$, $|x + 2v_x| \le 800$, $|y + 2v_y| \le 800$.

Tvoj program sme izvesti največ 2000 poizvedb. Če poskusi po tistem izvesti še kakšno, bo ocenjevalni program nanjo odgovoril z nizom `STOP` in se prenehal izvajati. V tem primeru naj se tudi tvoj program pravilno preneha izvajati, če hočeš dobiti od ocenjevalnega strežnika oceno WA (napačen odgovor); če bo tvoj program še naprej pošiljal poizvedbe in čakal na odgovor, ga ne bo dočakal, sčasoma pa bo dobil oceno TLE (prekoračen čas izvajanja).

Točkovanje: če program na koncu izpiše napačne rezultate ali pa se v kakšnem drugem pogledu ne drži prej opisanih navodil za sporazumevanje z ocenjevalnim strežnikom, dobi pri trenutnem testnem primeru 0 točk, sicer pa dobi pri njem vse točke.

Primer:

Tvoj program izpiše:	Sistem izpiše:
	`6 2 -3 2`
`? 2 0 5 2`	
	`NE`
`? 2 3 3 4`	
	`DA`
`? 3 1 4 5`	
	`NE`
`? 2 4 4 4`	
	`DA`
`3 5`	

Komentar: v prvem koraku se je delec premaknil s $(6, 2)$ preko $(3, 2)$ na $(3, 4)$, nakar se mu je hitrost razpolovila z $(-3, 2)$ na $(-1, 1)$. Drugi korak bi se moral začeti z vodoravnim premikom s $(3, 4)$ na $(2, 4)$, vendar je na polju $(2, 4)$ ovira in se delec sploh ne premakne v vodoravni smeri; pač pa nato izvede še navpični premik tega koraka, s $(3, 4)$ na $(3, 5)$. Hitrosti se mu nato razpolovita na $(0, 0)$, torej je delec dosegel svoj končni položaj.

2. Steklenice

V tovarni steklenic proizvajamo steklenice različnih odtenkov med prozorno in temno zeleno, ki se uporabljajo za sokove, pivo in ostale pijače. Odtenek vsake steklenice lahko predstavimo s celim številom med 0 (prozorna) in 10^9 (temno zelena). V skladišču že imamo n paketov; i-ti od njih (za $i = 1, 2, \ldots, n$) ima p_i steklenic odtenka o_i. Vse steklenice v istem paketu imajo isti odtenek, ker smo jih ustvarili iz iste zlitine.

Prejeli smo k naročil za steklenice in izpolniti želimo vsa; j-to izmed teh naročil (za $j = 1, 2, \ldots, k$) je opisano s števili s_j, a_j, b_j, ki pomenijo, da je naročenih s_j steklenic z odtenkom $a_j \leq o \leq b_j$ (pri tem ni nujno, da ima vseh s_j steklenic enak odtenek; lahko imajo različne odtenke, samo da so vsi z območja od a_j do b_j).

Steklenice iz paketov v skladišču lahko uporabimo za izpolnjevanje naročil, pri čemer lahko steklenice iz posameznega paketa razdelimo tudi med več različnih naročil; tudi ni treba, da porabimo vse steklenice iz paketa. Če naročil ne bo mogoče v celoti izpolniti s steklenicami iz skladišča, bomo morali izdelati dodatne steklenice.

Napiši program, ki izračuna, najmanj koliko dodatnih steklenic moramo narediti, da lahko ustrežemo vsem naročilom.

Vhodni podatki: v prvi vrstici sta podani dve s presledkom ločeni celi števili, n in k. Nato sledi n vrstic, ki opisujejo obstoječe pakete steklenic v skladišču; i-ta od njih vsebuje dve celi števili, p_i in o_i (število steklenic in odtenek). Nato sledi k vrstic, ki opisujejo vsa naročila; j-ta izmed njih vsebuje tri s presledkom ločena cela števila: s_j, a_j, b_j.

Omejitve: $1 \leq n \leq 10^5$; $1 \leq k \leq 10^5$; vsa ostala števila so nenegativna in manjša ali enaka 10^9. Pri prvih 40 % testnih primerov velja tudi $n \leq 1000$ in $k \leq 1000$.

Izhodni podatki: izpiši eno samo celo število, namreč najmanjše število steklenic, ki jih je treba še izdelati, da bomo lahko izpolnili vsa naročila.

Primer vhoda:

```
3 3
4 100
100 1000
13 7
10 0 1000
50 50 60
150 900 1100
```

Pripadajoči izhod:

```
100
```

Komentar: porabimo vse steklenice iz skladišča, poleg tega pa moramo izdelati še dodatnih 50 steklenic za drugo naročilo, ki mu ne ustrezajo nobene steklenice iz skladišča, in 50 dodatnih steklenic za tretje naročilo.

3. Zlatarna

Po mnogo tednih načrtovanja in kopanja ste s kolegi tatovi izkopali rov v zlatarno. Zlatarna prodaja nakit, dragulje in mnoge druge dragocenosti. Nagrabiti želite čim višjo vrednost predmetov, vaše oči pa so še posebej uprte v verižice in smaragde, ki so všeč ženi šefa vaše kriminalne združbe. Skozi svojo temeljito raziskavo ste pridobili natančen seznam n predmetov, ki se v zlatarni nahajajo. Za vsakega od njih veste, kakšna je njegova vrednost v_i, ali gre za verižico in ali ima predmet vgrajen smaragd. Zadnji dve lastnosti si nista izključujoči: kos nakita je lahko smaragdna verižica (odkljuka obe kategoriji), zgolj verižica, zgolj nakit s smaragdom ali pa nič od tega. Iz zlatarne lahko odnesete le k predmetov, saj imate v vrečah omejen prostor. Od tega mora biti vsaj a predmetov verižic in vsaj b predmetov s smaragdom. **Napiši program**, ki izračuna, koliko je najvišja skupna vrednost predmetov, ki jo lahko odnesete.

Vhodni podatki. V prvi vrstici so štiri cela števila, ločena s presledkom: n, k, a, b. V drugi vrstici sledi n celih števil $v_1, \ldots, v_n$, ločenih s po enim presledkom; to so vrednosti predmetov. V tretji vrstici je najprej število verižic, nato pa so navedeni njihovi indeksi, ločeni s presledki (indeksi so med 1 in n). V četrti vrstici je najprej število predmetov s smaragdi, nato pa so navedeni njihovi indeksi, ločeni s presledki (indeksi so med 1 in n).

Omejitve: $0 < k < n \leq 5 \cdot 10^5$; $a < n$; $b < n$; za vsak i velja $0 \leq v_i \leq 10^9$ (priporočamo, da za računanje z vrednostmi uporabiš kak 64-bitni podatkovni tip).

V 20 % testnih primerov ne bo noben kos nakita hkrati ogrlica in hkrati smaragden. Pri nadaljnjih 20 % testnih primerov velja $n \leq 1000$.

Zagotovljeno je, da rešitev vedno obstaja.

Izhodni podatki: izpiši le eno celo število, namreč največjo skupno vrednost predmetov, ki jih je mogoče odnesti ob upoštevanju omejitev iz besedila naloge.

Primer vhoda:	Pripadajoči izhod:

`5 3 2 1`
`15 6 3 2 10`
`2 4 5`
`2 3 5`
	`27`

Komentar: izberemo predmete 1, 4 in 5.

4. Urejanje z lažmi

V nekem skladišču je n zabojev, oštevilčenih z naravnimi števili od 1 do n. Nobena dva zaboja nista enako težka, njihovih tež pa ne poznamo. Dobili smo le tabelo, v kateri za vsak par zabojev (i, j), kjer je $1 \leq i < j \leq n$, piše, ali je zaboj i lažji ali težji od zaboja j. Poleg tega vemo, da je eden od teh podatkov v tabeli lahko napačen, torej da za neki konkreten par (i, j) v tabeli piše, da je zaboj i lažji od zaboja j, v resnici pa je težji od zaboja j, ali obratno. Vemo, da taka napaka nastopi pri največ enem paru (i, j), ne vemo pa, pri katerem (če sploh pri katerem; mogoče je tudi, da so vsi podatki v tabeli pravilni).

Radi bi določili vrstni red zabojev po teži (od najlažjega do najtežjega), vendar tega na podlagi zgoraj opisanih podatkov ni vedno mogoče določiti enolično; lahko se zgodi, da obstaja več rešitev (torej več kombinacij vrstnega reda zabojev in položaja napake), ki bi vse dale takšno tabelo, kot jo vidimo v vhodnih podatkih. **Napiši program**, ki za dano tabelo ugotovi, koliko takšnih rešitev obstaja, in nekatere od njih tudi izpiše.

Vhodni podatki: v prvi vrstici je celo število n, torej število zabojev. Sledi $n - 1$ vrstic; j-ta od teh vrstic vsebuje niz, dolg j znakov; i-ti od teh znakov je „<“, če je zaboj i lažji od zaboja $j + 1$, oz. „>“, če je zaboj i težji od zaboja $j + 1$. Še enkrat poudarimo, da je eden od teh podatkov (za en par zabojev) lahko napačen.

Omejitve: $1 \leq n \leq 3000$.

Pri tej nalogi so štiri podnaloge, ki se razlikujejo po dodatnih omejitvah:

- (30 točk) $n < 10$
- (20 točk) $n \leq 100$
- (20 točk) $n \leq 300$
- (30 točk) brez dodatnih omejitev.

Če pri posamezni podnalogi pravilno rešiš vse njene testne primere, dobiš pri tej podnalogi vse točke, sicer pa nobene.

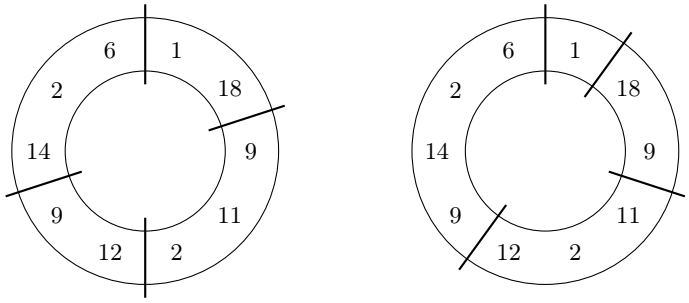
Izhodni podatki: v prvo vrstico izpiši število rešitev (kombinacij vrstnega reda in položaja napake v tabeli), ki so skladne z vhodnimi podatki. Nato izpiši rešitve, vsako v svojo vrstico; če obstaja več kot 10 rešitev, jih izpiši le poljubnih 10 izmed njih. Rešitve lahko izpišeš v poljubnem vrstnem redu. Vsaka vrstica z opisom rešitve mora vsebovati $n + 2$ števil, ločenih s po enim presledkom, recimo $i\ j\ a_1\ a_2\ \ldots\ a_n$, pri čemer naj za i in j velja $1 \leq i < j \leq n$. Takšna vrstica pove, da je pri tej rešitvi v tabeli napaka pri paru zabojev (i, j) in da je vrstni red zabojev po teži $a_1, a_2, \ldots, a_n$ (torej: za vsak k od 1 do n je a_k številka k-tega najlažjega zaboja). Če pri posamezni rešitvi v podatkih ni napake, izpiši za i in j vrednost 0.

Primer vhoda:	Možen pripadajoči izhod:	Primer vhoda:	Pripadajoči izhod:
`2`		`4`	`1`
`>`	`2`	`<`	`1 4 1 2 3 4`
	`0 0 2 1`	`<<`	
	`1 2 1 2`	`><<`	

Komentar: v levem primeru nam vhodni podatki pravijo, da je zaboj 1 težji od zaboja 2. Skladni s tem sta dve rešitvi: bodisi je ta podatek pravilen in je vrstni red zabojev (od najlažjega do najtežjega) enak $[2, 1]$ bodisi je tisti podatek napačen in je zaboj 1 v resnici lažji od zaboja 2, vrstni red pa je torej $[1, 2]$. — V desnem primeru se izkaže, da je edina možna rešitev, ki je skladna z vhodnimi podatki, ta, da je napaka pri paru $(1, 4)$, kjer vhodni podatki pravijo, da je zaboj 1 težji od zaboja 4, v resnici pa je torej lažji od njega in vrstni red zabojev po teži je $[1, 2, 3, 4]$.

5. Razrez kolobarja

Dan je kolobar, na katerem je zapisanih n pozitivnih celih števil. Če ga na k mestih (med dvema sosednjima številoma) prerežemo, nam razpade na k kosov. Izračunajmo vsoto števil na vsakem kosu; največjo od teh vsot (po vseh k kosih) si predstavljajmo kot *ceno* razreza. Odvisno od položaja rezov je lahko cena višja ali nižja; naslednja slika kaže dva primera razrezov istega kolobarja $n = 10$ števil na $k = 4$ kose:



Pri levem razrezu so vsote števil na kosih enake $14+2+6 = 22$, $9+12 = 21$, $1+18 = 19$ in $2 + 11 + 9 = 22$; cena tega razreza je zato enaka $\max\{22, 21, 19, 22\} = 22$. Podoben razmislek pokaže, da je cena desnega razreza enaka 31.

Napiši program, ki za dani kolobar in želeno število kosov k izračuna najmanjšo možno ceno razreza.

Vhodni podatki: vsak testni primer vsebuje več kolobarjev. V prvi vrstici vhoda je število kolobarjev t (zanj velja $1 \leq t \leq 10^4$). Nato sledijo opisi kolobarjev, vsak od njih pa obsega dve vrstici: v prvi vrstici sta celi števili n in k, ločeni s presledkom; v drugi vrstici pa so navedena števila, ki nastopajo na kolobarju (v takem vrstnem redu, v kakršnem si sledijo vzdolž kolobarja); to je n celih števil, ločenih s po enim presledkom.

Vsoto n-jev po vseh kolobarjih v testnem primeru označimo z N. To bo prišlo prav v nadaljevanju pri opisu omejitev.

Omejitve: $1 \leq k \leq n \leq N \leq 10^5$. Vsako izmed števil na kolobarju je večje ali enako 1 in manjše od 10^9.

Pri tej nalogi je pet podnalog, ki se ločijo po dodatnih omejitvah:

- (10 točk) $N \leq 1000$ in $1 \leq a_1 = a_2 = \cdots = a_n \leq 1000$
- (10 točk) $k = n - 1$
- (15 točk) $k = 2$
- (25 točk) $N \leq 1000$
- (40 točk) brez dodatnih omejitev.

Pri posamezni podnalogi dobiš vse točke, če tvoj program pravilno reši vse testne primere te podnaloge, sicer pa pri njej ne dobiš nobene točke.

Izhodni podatki: izpiši t vrstic, po eno za vsak kolobar iz vhodnih podatkov. V tisto vrstico izpiši eno samo celo število, namreč najnižjo ceno razreza, ki jo je mogoče doseči, če po pravilih iz besedila naloge razrežemo tisti kolobar na k kosov.

Primer vhoda:

```
2
10 4
9 18 1 6 2 14 9 12 2 11
6 4
1 1 5 5 5 1
```

Pripadajoči izhod:

```
22
5
```

Komentar: prvi kolobar v tem primeru ustreza levi sliki zgoraj.

20. tekmovanje ACM v znanju računalništva za srednješolce

22. marca 2025

REŠITVE NALOG ZA PRVO SKUPINO

1. Natakarica

Na svoji prvi poti bo natakarica postregla prvih nekaj niz — največ, kolikor je mogoče, ne da bi njihova naročila presegla skupno težo t. Pojdimo torej v zanki po mizah, seštevajmo teže njihovih naročil in pri vsaki mizi preverimo, če bi skupna teža naročila zdaj presegla t ali ne. Če bi ga presegla, moramo trenutno pot zaključiti (in povečati števec poti) in začeti novo pot, pri kateri seštevamo težo naročil spet od 0 naprej. Tako lahko nadaljujemo v zanki po mizah, dokler ne obdelamo vseh. Pri vsaki mizi imamo še vgnezdeno zanko, s katero preberemo naročila te mize in jih seštejemo.

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

int main()
{
  // Preberimo število miz in maksimalno težo.
  int stMiz, maxTeza; cin >> stMiz >> maxTeza;
  // stPoti šteje opravljene poti, tezaPoti je teža pri trenutni poti.
  int stPoti = 0, tezaPoti = 0;
  while (stMiz-- > 0)
  {
    // Preberimo in seštejmo težo naročil pri naslednji mizi.
    int stNarocil, tezaMize = 0; cin >> stNarocil;
    while (stNarocil-- > 0) {
      int narocilo; cin >> narocilo;
      tezaMize += narocilo; }
    // Ali moramo za to mizo začeti novo pot?
    if (tezaPoti + tezaMize > maxTeza)
      tezaPoti = 0, ++stPoti;
    // Dodajmo trenutno mizo v trenutno pot.
    tezaPoti += tezaMize;
  }
  // Prištejmo še pot, ki je trenutno v teku.
  if (tezaPoti > 0) ++stPoti;
  // Izpišimo rezultat.
  cout << stPoti << endl;
  return 0;
}
```

Na koncu glavne zanke pazimo še na to, da bomo šteli tudi zadnjo pot (tisto, pri kateri natakarica postreže zadnjo mizo). Naša spodnja rešitev poveča števec poti takrat, ko se neka pot konča, zato ga mora za zadnjo pot posebej povečati po koncu zadnje zanke (razen če je tezaPoti takrat enaka 0, kar bi se sicer lahko zgodilo le, če bi bila skupna teža vseh naročil po vseh mizah enaka 0). Druga možnost bi bila, da bi števec poti povečali takrat, ko se neka pot začne, torej bi ga že na začetku glavne zanke postavili na 1; potem ga na koncu glavne zanke ne bi bilo treba povečevati, manjša slabost pa bi bila, da bi v primeru, ko je število miz 0 (ali pa je skupna teža vseh naročil enaka 0), še vseeno izpisali 1 namesto bolj primernega rezultata 0.

Zapišimo to rešitev še v pythonu:

```
# Preberimo število miz in maksimalno težo.
stMiz, maxTeza = map(int, input().split())

# stPoti šteje opravljene poti, tezaPoti je teža pri trenutni poti.
stPoti = 0; tezaPoti = 0

for miza in range(stMiz):
  # Preberimo in seštejmo težo naročil pri naslednji mizi.
  tezaMize = sum(tuple(map(int, input().split())))[1:])

  # Ali moramo za to mizo začeti novo pot?
  if tezaPoti + tezaMize > maxTeza:
    tezaPoti = 0; stPoti += 1

  # Dodajmo trenutno mizo v trenutno pot.
  tezaPoti += tezaMize

# Prištejmo še pot, ki je trenutno v teku.
if tezaPoti > 0: stPoti += 1

# Izpišimo rezultat.
print(stPoti)
```

2. Uravnotežena prehrana

Pri tej nalogi je treba le pazljivo slediti navodilom; rešitev ni zapletena, je pa razmeroma dolga. Naša spodnja rešitev si za začetek v konstanti kolicine pripravi podatke o štirih količinah, ki nas pri tej nalogi zanimajo: kalorije, beljakovine, maščobe in ogljikovi hidrati; za vsako imamo ime ter spodnjo in zgornjo mejo. To nam bo omogočilo, da bomo kasneje lahko vse štiri količine obdelali v zanki, namesto da bi morali imeti štiri kopije skoraj enake izvorne kode.

Podatke o pojedenih sestavinah, ki jih beremo z vhoda, si moramo nekje zapomniti, kajti kasneje bomo morali iti čeznje še enkrat, ko bomo razmišljali o tem, ali je mogoče vnos hrane spraviti v priporočene okvire tako, da to ali ono sestavino odstranimo. Deklarirajmo torej strukturo Sestavina, ki lahko hrani podatke o eni sestavini (ime in vse štiri količine); imeli bomo vektor takšnih struktur. Sestavine beremo v zanki, sproti pa jih lahko tudi seštevamo, da dobimo skupni vnos celega dne (v spremenljivki skupaj). Pri branju vhoda je vredno omeniti še to, da ker so podatki v vsaki vrstici ločeni z vejicami in ne s presledki, smo pri klicu funkcije scanf v specifikaciji formata za branje imena uporabili `%[^,]` namesto bolj znanega `%s`.

Preverjanje tega, ali so količine znotraj priporočenih meja, nam bo prišlo prav na dveh mestih: najprej za skupni vnos, nato pa še za skupni vnos brez te ali one sestavine, ko bomo razmišljali o tem, ali lahko kakšno sestavino odstranimo. Zato bomo to preverjanje implementirali s samostojnim podprogramom; v spodnji rešitvi je to metoda Sestavina::Preveri, ki lahko po želji tudi izpiše, katere količine so zunaj predpisanih meja (parameter izpisi). Slednje pride prav pri preverjanju skupnega vnosa, ne želimo pa takšnega izpisa pri preverjanju skupnega vnosa po odstranitvi posamezne sestavine.

Glavni blok programa torej, če se pri preverjanju skupnega vnosa izkaže, da le-ta ni znotraj priporočenih meja, izvede še eno zanko po sestavinah; za vsako sestavino izračunamo, kaj ostane od skupnega vnosa, če to sestavino odstranimo, in preverimo, ali je ta preostanek znotraj predpisanih meja.

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

struct Kolicina { const char *ime; int Min, Max; };
constexpr Kolicina kolicine[] = {
  {"kalorije", 1600, 3000},
  {"beljakovine", 40, 150},
  {"mascobe", 35, 100},
  {"ogljikovi hidrati", 220, 500}
};

struct Sestavina
```

```
{
  string ime; int kol[4] = {0, 0, 0, 0};

  // Vrne logično vrednost, ki pove, ali so vse količine znotraj predpisanih meja.
  // Če je „izpisi“ == true, tudi izpiše tiste, ki so zunaj predpisanih meja.
  bool Preveri(bool izpisi) const
  {
    bool ok = true;
    for (int i = 0; i < 4; ++i)
    {
      int vnos = kol[i]; auto &K = kolicine[i];
      if (vnos >= K.Min && vnos <= K.Max) continue;
      if (! izpisi) return false;
      ok = false; printf("%s: vnos %d zunaj mej [%d, %d]\n", K.ime, vnos, K.Min, K.Max);
    }
    return ok;
  }
};

int main()
{
  int stSestavin; scanf("%d\n", &stSestavin);
  vector<Sestavina> sestavine(stSestavin);
  Sestavina skupaj;

  // Preberimo podatke o sestavinah in jih sproti tudi seštevajmo.
  for (auto &S : sestavine)
  {
    char ime[101]; scanf("%[^,],%d,%d,%d,%d\n",
                         ime, &S.kol[0], &S.kol[1], &S.kol[2], &S.kol[3]);
    S.ime = ime; for (int i = 0; i < 4; ++i) skupaj.kol[i] += S.kol[i];
  }

  // Preverimo skupni vnos; če je kaj zunaj meja, bo to izpisal podprogram Preveri(),
  // sicer pa izpišimo, da je vse v redu.
  if (skupaj.Preveri(true)) printf("Primerna prehrana.\n");

  // Če je kakšna količina zunaj meja, preglejmo vse sestavine še enkrat.
  else for (auto &S : sestavine)
  {
    // Odštejmo trenutno sestavino od skupnega vnosa in preverimo,
    // ali bi bil skupni vnos potem znotraj predpisanih meja.
    for (int i = 0; i < 4; ++i) S.kol[i] = skupaj.kol[i] - S.kol[i];
    if (S.Preveri(false)) printf("Lahko odstranis samo %s.\n", S.ime.c_str());
  }
  return 0;
}
```

Zapišimo podobno rešitev še v pythonu:

```
kolicine = [
  ("kalorije", 1600, 3000),
  ("beljakovine", 40, 150),
  ("mascobe", 35, 100),
  ("ogljikovi hidrati", 220, 500)
]

class Sestavina:
  def __init__(self, ime, kol): self.ime = ime; self.kol = kol

  # Vrne logično vrednost, ki pove, ali so vse količine znotraj predpisanih meja.
  # Če je „izpisi“ == true, tudi izpiše tiste, ki so zunaj predpisanih meja.
  def Preveri(self, izpisi: bool) -> bool:
    ok = True
    for i, (ime, Min, Max) in enumerate(kolicine):
      vnos = self.kol[i]
      if Min <= vnos <= Max: continue
      if not izpisi: return False
```

```
            ok = False; print(f"{ime}: vnos {vnos} zunaj mej [{Min}, {Max}]")
        return ok

stSestavin = int(input())
sestavine = []; skupaj = Sestavina("", [0, 0, 0, 0])
# Preberimo podatke o sestavinah in jih sproti tudi seštevajmo.
for _ in range(stSestavin):
    L = input().strip().split(',')
    S = Sestavina(L[0], list(map(int, L[1:])))
    sestavine.append(S)
    for i in range(4): skupaj.kol[i] += S.kol[i]

# Preverimo skupni vnos; če je kaj zunaj meja, bo to izpisal podprogram Preveri(),
# sicer pa izpišimo, da je vse v redu.
if skupaj.Preveri(True): print("Primerna prehrana.")

# Če je kakšna količina zunaj meja, preglejmo vse sestavine še enkrat.
else:
    for S in sestavine:
        # Odštejmo trenutno sestavino od skupnega vnosa in preverimo,
        # ali bi bil skupni vnos potem znotraj predpisanih meja.
        for i in range(4): S.kol[i] = skupaj.kol[i] - S.kol[i]
        if S.Preveri(False): print(f"Lahko odstranis samo {S.ime}.")
```

3. Razpolavljanje torte

Opazimo lahko, da se črka `i` pojavlja v vhodnem nizu le v besedi „`in`“, ki ločuje posamezne kose, in da se črka `p` pojavlja le v besedi „`pol`“ v opisu posameznega kosa; velikost posameznega kosa pa je odvisna le od tega, kolikokrat se v njegovem opisu pojavlja beseda „`pol`“: če se pojavlja enkrat, je ta kos velik 1/2 torte, če se pojavlja dvakrat, je kos velik 1/4 torte in tako naprej.

Vhodni niz (recimo, da ga naš podprogram dobi kot parameter `s`) lahko torej obdelujemo po črkah; ko vidimo `p`, delimo velikost trenutnega kosa z 2, ko vidimo `i`, pa prištejemo trenutni kos k vsoti in začnemo nov kos (vsak kos se začne z velikostjo 1, ki jo bomo seveda takoj nato zmanjšali na 1/2, ko bomo naleteli na prvi `p` v opisu tistega kosa). Na koncu ne pozabimo k vsoti prišteti še zadnjega kosa (ker za njim ni besede „`in`“, pri kateri bi sicer prišteli tisti kos k vsoti). Pred izpisom pomnožimo vsoto s 100 in jo zaokrožimo na celo število, da dobimo odstotke.

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

void Delez(const char *s)
{
    double skupaj = 0, trenKos = 1;
    // Obdelajmo vhodni niz po črkah.
    for (; *s; ++s)
        // Pri "pol" se trenutni kos prepolovi.
        if (*s == 'p') trenKos /= 2;
        // Pri "in" prištejemo trenutni kos vsoti.
        else if (*s == 'i') skupaj += trenKos, trenKos = 1;
    // Prištejmo še zadnji kos.
    skupaj += trenKos;
    // Izpišimo rezultat v odstotkih.
    printf("%d %%\n", int(100 * skupaj));
}
```

Zapišimo takšno rešitev še v pythonu:

```
def Delez(s: str) -> None:
    skupaj = 0; trenKos = 1
    # Obdelajmo vhodni niz po črkah.
```

```
  for c in s:
    # Pri "pol" se trenutni kos prepolovi.
    if c == 'p': trenKos /= 2
    # Pri "in" prištejemo trenutni kos vsoti.
    elif c == 'i': skupaj += trenKos; trenKos = 1
  # Prištejmo še zadnji kos.
  skupaj += trenKos;
  # Izpišimo rezultat v odstotkih.
  print("%d %%" % int(100 * skupaj))
```

Nalogo lahko rešimo tudi tako, da vhodni niz razrežemo pri znakih `i` na krajše dele, ki opisujejo posamezne kose; v vsakem kosu preštejmo znake `p` in upoštevajmo, da če je bilo teh znakov k, obsega ta kos $1/2^k$ torte. To moramo nato le še sešteti po vseh kosih in pretvoriti v odstotke:

```
def Delez2(s: str) -> None:
  print("%d %%" % int(100 * sum(0.5**t.count("p") for t in s.split('i'))))
```

Opazimo lahko tudi, da če v našem vhodnem nizu zamenjamo vsako besedo `"pol"` z nizom `"0.5"`, vsako besedo `"od"` z znakom `"*"` in vsako besedo `"in"` z znakom `"+"`, dobimo veljaven aritmetični izraz; na primer, iz `"pol in pol od pol"` nastane `"0.5 + 0.5 * 0.5"`. Vrednost takega izraza lahko v pythonu izračunamo s funkcijo `eval`.

```
def Delez3(s: str) -> None: print("%d %%" % int(100 * eval(
                                s.replace("pol", "0.5").replace("od", "*").replace("in", "+"))))
```

4. Ultrazvok

Recimo, da dobimo sliko kot zaporedje (vektor ali tabelo) nizov, pri čemer vsak niz predstavlja po eno vrstico slike. Najlažje je, če jo obdelamo v zanki po stolpcih. Pri vsakem stolpcu, recimo x, začnimo na vrhu slike, pri polju $(x, 0)$, in poglejmo, do katere globine (x, y) sega strnjena skupina enakih znakov, ki se začne na $(x, 0)$. To je debelina tega tkiva na tem mestu v sliki; ko jo poznamo, lahko pogledamo, ali je to morda največja ali najmanjša debelina te vrste tkiva doslej, in če je, si jo zapomnimo (spodnja rešitev ima v ta namen vektorja `dMin` in `dMax`). Potem nadaljujemo z zanko dol po stolpcu, da ugotovimo debelino naslednje plasti tkiva in tako naprej vse do dna stolpca.

Ko na ta način obdelamo vse stolpce, gremo na koncu v zanki po vseh 26 možnih vrstah tkiv in izpišemo rezultate (tista tkiva, ki se na sliki sploh niso pojavljala, prepoznamo na primer po tem, da je maksimalna globina 0 oz. da je manjša od minimalne).

```
void ObdelajSliko(const vector<string>& slika)
{
  int w = slika[0].length(), h = slika.size(); // Velikost slike.
  // dMin[c], dMax[c] = najmanjša in največja debelina tkiva c doslej.
  vector<int> dMin(26, h + 1), dMax(26, 0);
  // Obdelajmo sliko po stolpcih.
  for (int x = 0; x < w; ++x)
    for (int y = 0; y < h; )
    {
      char c = slika[y][x]; int yOd = y;
      // Kako globoko sega tkivo, ki se začne v (x, y)?
      while (y < h && slika[y][x] == c) ++y;
      c -= 'a'; int debelina = y - yOd;
      // Če je to največja ali najmanjša debelina doslej, si jo zapomnimo.
      if (debelina < dMin[c]) dMin[c] = debelina;
      if (debelina > dMax[c]) dMax[c] = debelina;
    }
  // Izpišimo rezultate.
  for (int c = 0; c < 26; ++c) if (dMin[c] <= dMax[c])
    cout << char('a' + c) << ' ' << dMin[c] << ' ' << dMax[c] << endl;
}
```

Manjša slabost gornje rešitve je, da mora biti celotna slika prisotna v pomnilniku. Nalogo lahko rešimo tudi tako, da sliko beremo iz datoteke vrstico po vrstico in v pomnilniku vedno hranimo le dve vrstici: trenutno in prejšnjo. Poleg tega za vsak stolpec x vzdržujmo tudi podatek o tem, kako debelo je doslej (do trenutne vrstice) trenutno tkivo v tistem stolpcu (spodnja rešitev to hrani v d[x]). Ko se tkivo v nekem stolpcu konča, poznamo njegovo pravo debelino in lahko po potrebi ustrezno popravimo vrednosti v tabelah dMin in dMax. To, da se je dosedanje tkivo končalo, prepoznamo načeloma po tem, da je istoležni znak trenutne vrstice drugačen od istoležnega znaka prejšnje vrstice, paziti pa moramo na dva robna primera: na vrhu slike (ko beremo prvo vrstico) prejšnje vrstice sploh še ni (in se tudi ne konča nobeno tkivo), na dnu slike pa se končajo vsa tkiva (in naslednje vrstice sploh ni). Slednje obravnava naša rešitev tako, da se glavna zanka (po y) izvede po koncu slike še enkrat (pri $y = h$, kjer je h višina slike), vendar takrat ne prebere še ene vrstice, pač pa za vsak stolpec upošteva, da se je tkivo v njem zdaj končalo. Lepo pri tem pristopu je, da hranimo v pomnilniku le dve vrstici in tabelo s trenutno debelino tkiva za vsak stolpec; poraba pomnilnika je zato $O(w)$ namesto $O(wh)$, če imamo sliko širine w in višine h. Oglejmo si implementacijo te rešitve v C++; predpostavili bomo, da dobimo sliko na standardnem vhodu, pri čemer sta v prvi vrstici navedeni širina in višina slike:

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

int main()
{
  // Preberimo širino in višino slike.
  int w, h; cin >> w >> h;
  string s(w, ' '), prej; // Trenutna in prejšnja vrstica.
  // d[x] = trenutna debelina tkiva v stolpcu x;
  // dMin[c], dMax[c] = najmanjša in največja debelina tkiva c doslej.
  vector<int> d(w, 0), dMin(26, h + 1), dMax(26, 0);
  // Obdelajmo sliko.
  for (int y = 0; y <= h; ++y)
  {
    // Preberimo naslednjo vrstico.
    swap(s, prej);
    if (y < h) cin >> s;
    // Primerjajmo istoležne znake trenutne in prejšnje vrstice.
    for (int x = 0; x < w; ++x) {
      // Ali se tu nadaljuje tkivo iz prejšnje vrstice?
      if (y < h && s[x] == prej[x]) { ++d[x]; continue; }
      // Če ne, se tu prejšnje tkivo konča.
      if (y > 0) { int c = prej[x] - 'a';
        if (d[x] < dMin[c]) dMin[c] = d[x];
        if (d[x] > dMax[c]) dMax[c] = d[x]; }
      // In začne se novo tkivo.
      d[x] = 1; }
  }
  // Izpišimo rezultate.
  for (int c = 0; c < 26; ++c) if (dMin[c] <= dMax[c])
    cout << char('a' + c) << ' ' << dMin[c] << ' ' << dMax[c] << endl;
  return 0;
}
```

Zapišimo prvo od gornjih dveh rešitev še v pythonu:

```
def ObdelajSliko(slika):
  w = len(slika[0]); h = len(slika) # Velikost slike.
```

```
# dMin[c], dMax[c] = najmanjša in največja debelina tkiva c doslej.
dMin = [h + 1] * 26; dMax = [0] * 26

# Obdelajmo sliko po stolpcih.
for x in range(w):
  y = 0
  while y < h:
    c = slika[y][x]; yOd = y

    # Kako globoko sega tkivo, ki se začne v (x, y)?
    while y < h and slika[y][x] == c: y += 1
    c = ord(c) - ord('a'); debelina = y - yOd

    # Če je to največja ali najmanjša debelina doslej, si jo zapomnimo.
    if debelina < dMin[c]: dMin[c] = debelina
    if debelina > dMax[c]: dMax[c] = debelina

# Izpišimo rezultate.
for c in range(26):
  if dMin[c] <= dMax[c]: print("%c %d %d" % (chr(ord('a') + c), dMin[c], dMax[c]))
```

5. Prijave na izlet

Med obravnavanjem vhodnih podatkov je koristno vzdrževati dve množici (oz. slovarja ali razpršeni tabeli; v C++ lahko uporabimo razred unordered_set): eno z imeni ljudi, ki so trenutno na avtobusu, in eno z imeni ljudi, ki so trenutno v čakalni vrsti. Tako bomo lahko v konstantno mnogo časa preverili, ali je neki potnik na avtobusu ali v vrsti oz. ga od tam pobrisali ali ga tja dodali. Množica pa ima to pomanjkljivost, da ne moremo posebej vplivati na to, v kakšnem vrstnem redu hrani svoje elemente. Pri potnikih na avtobusu to ni težava, saj naloga pravi, da jih smemo (pri ukazu „?“) našteti v poljubnem vrstnem redu; pač pa je vrstni red pomemben pri čakalni vrsti, kajti ko se (zaradi odjave) sprosti sedež na avtobusu, ga mora dobiti tisti potnik, ki je že najdlje v vrsti.

Koristno bi bilo torej vzdrževati spisek čakajočih tudi v vrsti kot podatkovni strukturi (v C++ uporabimo razred queue), kjer lahko pobiramo potnike iz vrste v enakem vrstnem redu, v kakršnem smo jih dodajali vanjo. Težavo nam povzročajo le še odjave; ob odjavi nekoga, ki trenutno čaka v vrsti, bi ga morali od tam načeloma pobrisati, toda iz vrste lahko brišemo le na začetku. To nevšečnost lahko zaobidemo tako, da ob odjavi takega potnika pobrišemo le iz množice čakajočih, ne pa iz vrste čakajočih; kasneje pa, ko se bo sprostil kak sedež na avtobusu in bomo hoteli potnika z začetka vrste premakniti na avtobus, imejmo v mislih, da so v vrsti lahko še vedno tudi nekateri že odjavljeni potniki (prepoznamo jih po tem, da jih v množici čakajočih ni več); z začetka vrste take potnike brišemo, dokler ne pridemo do takega, ki res še vedno čaka.

Oglejmo si implementacijo te rešitve v C++:

```
#include <unordered_set>
#include <queue>
using namespace std;

int main()
{
  unordered_set<string> naAvtobusu, vVrsti;
  queue<string> vrsta; " čakalna vrsta, morda z nekaterimi že odjavljenimi "

  // Preberimo število sedežev.
  int stSedezev; cin >> stSedezev;

  // Preberimo prijave in odjave.
  string kaj;
  while (cin >> kaj)
  {
    if (kaj == "?") // Izpis stanja.
    {
      // Izpišimo ljudi na avtobusu.
      bool prvi = true;
      for (const auto &ime : naAvtobusu) {
```

```
      if (prvi) prvi = false; else cout << " ";
      cout << ime; }
    // Izpišimo proste sedeže in število čakajočih.
    cout << ", prosti sedezi: " << (stSedezev - naAvtobusu.size())
         << ", v vrsti: " << vVrsti.size() << endl;
    continue;
  }

  // Sicer imamo prijavo ali odjavo; preberimo ime.
  string ime; cin >> ime;
  if (kaj == "+") // Prijava.
  {
    // Ali dobi prost sedež?
    if (naAvtobusu.size() < stSedezev) {
      cout << ime << " dobi prost sedez." << endl;
      naAvtobusu.emplace(ime); continue; }
    // Sicer gre v čakalno vrsto.
    vVrsti.emplace(ime); vrsta.emplace(ime);
    cout << ime << " je na " << vVrsti.size() << ". mestu v vrsti." << endl;
  }
  else // Odjava.
  {
    // Če je bil v vrsti, ga le pobrišemo iz nje.
    if (vVrsti.erase(ime) > 0) continue;
    // Sicer je moral biti na avtobusu.
    naAvtobusu.erase(ime);
    // Poglejmo, kdo pride iz vrste na avtobus.
    while (! vrsta.empty())
    {
      // Pobrišimo človeka z začetka vrste.
      string ime2 = vrsta.front(); vrsta.pop();
      // Morda se je ta človek že prej odjavil in ga v resnici
      // sploh ni bilo več v vrsti.
      if (vVrsti.erase(ime2) == 0) continue;
      // Sicer se zdaj premakne na avtobus.
      cout << ime2 << " se premakne iz cakalne vrste na avtobus." << endl;
      naAvtobusu.emplace(ime2); break;
    }
  }
}
return 0;
}
```

Zapišimo takšno rešitev še v pythonu:

```
import queue, sys

# Preberimo število sedežev.
stSedezev = int(input())
# Preberimo prijave in odjave.
naAvtobusu = set(); vVrsti = set()
vrsta = queue.SimpleQueue()
while True:
  kaj = sys.stdin.readline()
  if not kaj: break

  if kaj[0] == "?": # Izpis stanja.
    print("%s, prosti sedezi: %d, v vrsti: %d" % (" ".join(naAvtobusu),
      stSedezev - len(naAvtobusu), len(vVrsti)))
    continue

  # Sicer imamo prijavo ali odjavo.
```

```
ime = kaj[1:].strip()
if kaj[0] == "+": # Prijava.
    # Ali dobi prost sedež?
    if len(naAvtobusu) < stSedezev:
        naAvtobusu.add(ime); print(f"{ime} dobi prost sedez."); continue
    # Sicer gre v čakalno vrsto.
    vVrsti.add(ime); vrsta.put(ime)
    print(f"{ime} je na {len(vVrsti)}. mestu v vrsti.")
else: # Odjava.
    # Če je bil v vrsti, ga le pobrišemo iz nje.
    if ime in vVrsti: vVrsti.remove(ime); continue
    # Sicer je moral biti na avtobusu.
    naAvtobusu.remove(ime)
    # Poglejmo, kdo pride iz vrste na avtobus.
    while not vrsta.empty():
        # Pobrišimo človeka z začetka vrste.
        ime2 = vrsta.get()
        # Morda se je ta človek že prej odjavil in ga v resnici
        # sploh ni bilo več v vrsti.
        if ime2 not in vVrsti: continue
        # Sicer se zdaj premakne na avtobus.
        print(f"{ime2} se premakne iz cakalne vrste na avtobus.")
        vVrsti.remove(ime2); naAvtobusu.add(ime2); break
```

Opisana rešitev pa ima naslednjo pomanjkljivost: če se nekdo odjavi, medtem ko čaka v čakalni vrsti, in se takoj zatem spet prijavi, bomo zanj na konec vrste dodali nov element, še preden je njegov stari element prišel na začetek vrste in bil pobrisan. Tako imamo lahko za istega človeka dva ali tudi več (poljubno veliko) zapisov v vrsti hkrati. Ko pride najstarejši od njih na začetek vrste, bomo v množici vVrste videli, da ta človek trenutno čaka v vrsti, in bomo mislili, da se zapis z začetka vrste nanaša nanj, čeprav je v resnici tisto neki star zapis, ki bi ga bilo treba pobrisati, medtem ko je aktualni zapis za tega čakajočega tisti med njegovimi zapisi, ki je najbližje koncu vrste. Tako se torej lahko zgodi, da bomo iz vrste na avtobus premaknili napačnega potnika. To lahko preprečimo tako, da vsaki prijavi pripišemo enolično zaporedno številko; naAvtobusu in vVrsti naj namesto množic postaneta slovarja (razpršeni tabeli), kjer so ključi še vedno imena, spremljevalna vrednost pa naj bo številka najnovejše prijave tistega človeka; v vrsti čakajočih pa poleg njihovih imen hranimo tudi številke prijav. Tako ne bo težko preveriti, ali se zapis na začetku vrste nanaša na trenutno aktualno prijavo nekega človeka ali pa na neko staro prijavo (in je zato treba ta zapis pobrisati).

Še ena možnost bi bila, da namesto vrste uporabimo seznam, v katerem so elementi povezani s kazalci (*linked list*; v C++ lahko uporabimo razred list), namesto množice čakajočih pa imejmo slovar oz. razpršeno tabelo (v C++ uporabimo unordered_map), kjer kot ključi nastopajo imena čakajočih, pripadajoča vrednost pa je kazalec (oz. iterator) na pripadajoči člen v seznamu. Tako bomo lahko čakajočega potnika, ki se odjavi, nemudoma pobrisali tako iz slovarja kot iz seznama.

REŠITVE NALOG ZA DRUGO SKUPINO

1. Omrežnina

Za začetek je koristno urediti meritve naraščajoče; recimo torej, da si po vrsti sledijo meritve $a_1 \leq a_2 \leq \cdots \leq a_n$. V formuli za omrežnino, $d \cdot M + k \cdot P$, nastopata konstanti M in P (ki ju naš podprogram dobi kot parametra) in spremenljivki d in k; toda slednji nista neodvisni, saj k pomeni število meritev, ki so večje od d.

Opazimo lahko, da če si izberemo katerikoli d z območja $a_i \leq d < a_{i+1}$, so od tega d-ja večje meritve $a_{i+1}, \ldots, a_n$, druge pa ne; za vse d-je z omenjenega območja je torej

$k = n - i$. Omrežnina $d \cdot M + k \cdot P$ je potem odvisna le še od d-ja in je (ker je $M \geq 0$) najmanjša takrat, ko je tudi d najmanjši, to pa je pri $d = a_i$.

Podoben razmislek pokaže, da če si izberemo katerikoli d z območja $0 \leq d < a_1$, so od tega d-ja večje vse meritve, torej je $k = n$ in najmanjšo omrežnino (po d-jih s tega območja) tudi v tem primeru dobimo pri najmanjšem d-ju, to je $d = 0$.

Podobno tudi, če si izberemo katerikoli d z območja $d \geq a_n$, ni od njega večja nobena meritev, torej je $k = 0$ in med vsemi temi d-ji bo omrežnina najmanjša spet pri najmanjšem izmed njih, to je pri $d = a_n$.

Tako torej vidimo, da je dovolj, če kot možne vrednosti d preizkusimo le $0, a_1, a_2, \ldots, a_n$; pri $d = 0$ imamo omrežnino $n \cdot P$, pri $d = a_i$ pa omrežnino $a_i \cdot M + (n - i) \cdot P$. Te kandidate lahko pregledamo v zanki in vrnemo tistega, ki dá najnižjo omrežnino.

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

int DolociMoc(vector<int> meritve, int M, int P)
{
  // Uredimo meritve naraščajoče.
  const int n = meritve.size();
  sort(meritve.begin(), meritve.end());

  // En kandidat za rešitev je, ko je dogovorjena moč 0 in so vse meritve nad njo (k = n).
  int najMoc = 0, najOmreznina = n * P;

  // Preglejmo še rešitve z višjo dogovorjeno močjo in manj presežki (manjšim k).
  for (int i = 0; i < n; ++i)
  {
    // Če za dogovorjeno moč vzamemo meritve[i], nastopijo presežki pri meritvah
    // od i + 1 do n − 1, torej n − 1 − i presežkov.
    int omreznina = meritve[i] * M + (n − 1 − i) * P;

    // Če je to najboljša rešitev doslej, si jo zapomnimo.
    if (omreznina < najOmreznina) najMoc = meritve[i], najOmreznina = omreznina;
  }

  return najMoc; // Vrnimo najboljšo rešitev.
}
```

Če je slučajno več zaporednih meritev enakih, so nekateri kandidati, ki jih ta postopek pregleda, sicer nesmiselni; na primer, če je $a_i = a_{i+1}$ in če vzamemo $d = a_i$, potem ni res, da so meritve $a_{i+1}, \ldots, a_n$ večje od d, saj je meritev a_{i+1} enaka meritvi a_i in s tem tudi d-ju. Toda to pomeni le, da bomo pri takih nesmiselnih kandidatih precenili število presežkov k in zato dobili previsoko omrežnino; pri $i + 1$ bomo dobili enak d kot pri i (ker je obakrat $d = a_{i+1} = a_i$), vendar manjši k (namreč $k = n - (i + 1)$ namesto $n - i$) in zato nižjo omrežnino. Nesmiselni kandidat i torej ne bo dal najnižje omrežnine in zato ne bo vplival na rezultat naše funkcije.

2. Trojice

Preprosta rešitev je, da s tremi gnezdenimi zankami pregledamo vse trojice nizov iz našega vhodnega seznama in za vsako preverimo, ali ustreza zahtevam naloge; slednje zahteva še eno zanko po črkah, kjer primerjamo istoležne črke vseh treh nizov. Ta rešitev ima časovno zahtevnost $O(n^3\, k)$, kar je neugodno veliko. Ker naloga posebej poudarja, naj bo naš postopek čim bolj učinkovit, razmislimo o boljši rešitvi.

Recimo, da sta prva dva niza v trojici s in t. Oglejmo si dva njuna istoležna znaka, recimo $s[r]$ in $t[r]$. Če sta tadva znaka enaka, mora imeti tudi tretji niz v trojici na tem indeksu enak znak kot onadva. Če pa sta $s[r]$ in $t[r]$ različna, mora imeti tretji niz na tem mestu neki znak, ki je različen tako od $s[r]$ kot od $t[r]$, za ta znak pa je takrat ena sama možnost, namreč tista izmed črk `a`, `b` in `c`, ki ni enaka niti $s[r]$ in $t[r]$ (saj naloga pravi, da v naših nizih nastopajo le te tri črke).

Vidimo torej, da če poznamo dva niza trojice, je tretji niz s tem že enolično določen. Tako smo prišli do naslednje rešitve: z dvema gnezdenima zankama pojdimo po vseh parih nizov iz našega vhodnega seznama; za vsak par pojdimo po znakih in izračunajmo,

kakšen bi moral biti tretji niz, ki bi tvoril z omenjenima dvema nizoma veljavno trojico; potem pa moramo le še preveriti, ali takšen tretji niz tudi zares imamo na našem vhodnem seznamu. Da bomo to preverili hitro, je koristno shraniti nize tudi v razpršeno tabelo oz. množico,[1] kjer lahko niz poiščemo v času, ki je bolj ali manj neodvisen od števila nizov n. Tako imamo rešitev s časovno zahtevnostjo $O(n^2\,k)$.

Pazimo še na to, da če se neka primerna trojica zares pojavlja v našem vhodnem seznamu, jo bomo opazili trikrat, ker lahko na tri načine izberemo dva od treh nizov trojice (in potem izračunamo, kakšen mora biti tretji, in opazimo, da takega tudi zares imamo). Število trojic moramo torej na koncu deliti s tri.

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

int PrestejTrojice(const vector<string> &v)
{
  const int k = v[0].length(), n = v.size();
  // Pripravimo si množico vseh nizov iz „v“.
  unordered_set<string> h { v.begin(), v.end() };
  // Preglejmo vse pare nizov.
  string u(k, ' '); int stTrojic = 0;
  for (int i = 1; i < n; ++i) for (int j = 0; j < i; ++j)
  {
    // Poglejmo, kateri niz bi skupaj z nizoma v[i] in v[j] primerno trojico.
    auto &s = v[i], &t = v[j];
    for (int r = 0; r < k; ++r)
      if (s[r] == t[r]) u[r] = s[r];
      else u[r] = 'a' + 3 - (s[r] - 'a') - (t[r] - 'a');
    // Če tak niz imamo, smo našli trojico.
    if (h.count(u) == 1) ++stTrojic;
  }
  return stTrojic / 3;  // Upoštevajmo, da smo vsako trojico šteli trikrat.
}

int main()
{
  // Preberimo vhodne podatke.
  int n, k; cin >> n >> k;
  vector<string> v(n);
  for (auto &s : v) cin >> s;
  // Izračunajmo in izpišimo rezultat.
  cout << PrestejTrojice(v) << endl; return 0;
}
```

Za določanje tega, kakšen mora biti znak v tretjem nizu (recimo mu u), da bo različen od istoležnih znakov v prvih dveh nizih (recimo s in t), smo si v gornji rešitvi pomagali z naslednjim opažanjem: če si znake namesto kot črke `a`, `b` in `c` predstavljamo kot števila 0, 1 in 2, potem se trije različni znaki vedno seštejejo v $0 + 1 + 2 = 3$. Če torej dva znaka že poznamo in vemo, da sta različna, recimo s_r in t_r (torej r-ti znak nizov s oz. t), potem mora biti tretji enak $u_r := 3 - s_r - t_r$.

S tem razmislekom gremo lahko še korak dlje. Če imajo na nekem mestu trije nizi tri različne znake, se ti znaki seštejejo v $0 + 1 + 2 = 3$; če pa imajo vsi trije enak znak,

[1] Še ena možnost je, da bi nize hranili v drevesu po znakih (*trie*), kjer bi se lahko že sproti, medtem ko računamo znake tretjega niza, tudi spuščali po drevesu; z malo sreče bi pri marsikaterem paru nizov že po prvih nekaj črkah opazili, da se v drevesu ne dá nadaljevati spuščanja s tisto črko, torej primernega tretjega niza nimamo in lahko nad trenutnim parom takoj obupamo. Pri večini parov torej ne bi porabili $O(k)$ časa, ampak bolj $O(1)$. Res pa je, da ni težko sestaviti patoloških primerov, kjer bi pri vsakem paru porabili $O(k)$ časa, na primer tako, da se vsi nizi našega seznama ujemajo v vseh znakih razen v zadnjih nekaj.

je njihova vsota 0, 3 ali 6. Če bi torej u_r računali po formuli $u_r := 6 - s_r - t_r$, bi dobili včasih ravno pravo vrednost, včasih pa za 3 ali 6 preveliko (odvisno od tega, ali je prava vsota vseh treh znakov enaka 6, 3 ali 0). Pravo vrednost torej dobimo tako, da od omenjene razlike obdržimo le ostanek po deljenju s 3, torej $u_r := (6 - s_r - t_r) \bmod 3$. Ta formula zdaj deluje tako v primerih, ko sta s_r in t_r enaka, kot v primerih, ko sta različna.

Pravkar dobljeno formulo lahko izkoristimo za še eno izboljšavo, s katero postane naša rešitev za neki konstanten faktor hitrejša. Niz dolžine k lahko predstavimo s $3k$-bitnim celim številom; vsako črko torej predstavljajo trije biti: $\mathtt{a} = 0 = 000_2$, $\mathtt{b} = 1 = 001_2$ in $\mathtt{c} = 2 = 010_2$. Niz s, ki ga tvorijo znaki $s_0, s_1, \ldots, s_{k-1}$, je tako predstavljen kot celo število $(s_0 s_1 \ldots s_{k-1})_8 = \sum_{r=0}^{k-1} s_r \cdot 8^{k-1-r}$.

Recimo, da imamo niza s in t predstavljena s takšnima $3k$-bitnima številoma in bi radi zdaj izračunali $3k$-bitno število za tretji niz, u, ki tvori skupaj s s in t takšno trojico, po kakršnih sprašuje naša naloga. Da bo šlo to čim hitreje, bi radi izračun $u_r := (6 - s_r - t_r) \bmod 3$ izvedli hkrati na vseh k skupinah treh bitov, ki predstavljajo posamezne črke niza. Omenjeno formulo lahko razdelimo na tri korake:

1 $u_r := 6 - s_r - t_r$;
2 **if** $u_r \geq 4$ **then** $u_r := u_r - 3$;
3 **if** $u_r = 3$ **then** $u_r := 0$;

Najprej torej izračunamo $6 - s_r - t_r$, potem pa enkrat ali dvakrat odštejemo 3, dokler vrednost ne pride na območje $\{0, 1, 2\}$.

Razmislimo o tem, kako naj te tri korake izvedemo z operacijami na številih s in t. Če bi na vsakem mestu r (od 0 do $k-1$) izvedli prvi korak, $u_r := 6 - s_r - t_r$, bi tako dobljene trojice bitov tvorile število $u := \sum_r u_r 8^{k-1-r} = (66\ldots6)_8 - s - t$. Tega ni težko izračunati; konstanto $(66\ldots6)_8$, torej število, ki je v osmiškem zapisu sestavljeno iz k šestic, si lahko pripravimo vnaprej.

Pri drugem koraku upoštevajmo, da je u_r med 0 in 6; pogoj $u_r \geq 4$ je torej izpolnjen natanko tedaj, ko je v dvojiškem zapisu števila u_r prižgan bit 2 (to je tretji bit z desne). Če torej vrednost tega bita pomnožimo s 3 in jo odštejemo od u_r, bo učinek ta, da če je bil u_r prej ≥ 4, ga bomo zdaj zmanjšali za 3, če pa je bil u_r prej manjši od 4, se ne bo nič spremenil (ker bomo od njega odšteli $3 \cdot 0 = 0$). Drugega od gornjih treh korakov lahko torej predelamo v:

$$u_r := u_r - \big((u_r \textbf{ shr } 2) \textbf{ and } 1\big) \cdot 3;$$

Če izvedemo to pri vsaki trojici bitov, torej vsakem r od 0 do $k-1$, je učinek na u kot celoto naslednji:

$$u := u - \big((u \textbf{ shr } 2) \textbf{ and } (11\ldots1)_8\big) \cdot 3;$$

Spet vidimo, da je to nekaj, kar lahko izračunamo s peščico aritmetičnih operacij na $3k$-bitnih številih. Konstanta $(11\ldots1)_8$, torej število, ki ga v osmiškem zapisu sestavlja k enic, ima vrednost $(11\ldots1)_8 = \sum_{r=0}^{k-1} 1 \cdot 8^r = (8^k - 1)/7$. Malo prej omenjena $(66\ldots6)_8$ je njen šestkratnik.

Ostane nam še tretji korak; ko pridemo do njega, ima u_r že vrednost z območja $\{0, 1, 2, 3\}$, mi pa moramo zdaj preveriti, če je enak 3, in ga v tem primeru postaviti na 0. Ker je u_r zdaj manjši od 4, si ga lahko predstavljamo kot dvobitno število, sestavljeno iz spodnjega bita x_r in zgornjega bita y_r, torej $u_r = 2y_r + x_r$. Operacijo, ki nas zanima — „če je u_r enak 3, ga postavi na 0“ — lahko zdaj razumemo tudi takole: če je bil bit x_r od prej ugasnjen, bo tak tudi ostal; če pa je bil prižgan, bo ostal prižgan, razen če je bil prižgan tudi y_r — slednje namreč pomeni, da je imel u_r vrednost 3 in bo treba bit x_r zdaj ugasniti. Po novem bo torej x_r prižgan natanko v primeru, če je bil prižgan že prej in če je bil y_r prej ugasnjen. Enak razmislek velja seveda tudi, če vlogo obeh bitov obrnemo. Korak 3 lahko torej predelamo takole:

$x_r := u_r$ **and** 1; $y_r := (u_r$ **shr** 1) **and** 1;
$\hat{x}_r := x_r$ **and not** y_r; $\hat{y}_r := y_r$ **and not** x_r;
$u_r := 2 \cdot \hat{y}_r + \hat{x}_r$;

Spet vidimo, da tega ni težko izvesti na vseh k trojicah bitov v naših $3k$-bitnih številih naenkrat:

$$x := u \textbf{ and } (11\ldots1)_8;\ y := (u \textbf{ shr } 1) \textbf{ and } (11\ldots1)_8;$$
$$\hat{x} := x \textbf{ and not } y;\ \hat{y} := y \textbf{ and not } x;$$
$$u := 2 \cdot \hat{y} + \hat{x};$$

Oglejmo si implementacijo te rešitve v C++.

```
#include <cstddint>
typedef uint_fast64_t myint;

constexpr int z = (sizeof(myint) * 8) / 3; // tako dolg niz lahko predstavimo v enem številu
constexpr myint enice = ((myint(1) << (3 * z)) - 1) / 7; // število (11...1)_8


myint TretjiNiz(myint s, myint t)
{
  // Na vsakem mestu r izračunajmo u_r = 6 - s_r - t_r.
  myint u = 6 * enice - s - t;

  // Kjer je u_r >= 4, ga zmanjšajmo za 3.
  u -= ((u >> 2) & enice) * 3;

  // Kjer je u_r = 3, ga postavimo na 0.
  myint spodnji = u & enice, zgornji = (u >> 1) & enice;
  return (spodnji &  zgornji) | ((zgornji &  spodnji) << 1);
}

int PrestejTrojice2(const vector<string> &vs)
{
  const int k = vs[0].length(), n = vs.size();

  // Predelajmo nize v števila in jih shranimo v množico h.
  vector<myint> v(n); unordered_set<myint> h;
  for (int i = 0; i < n; ++i) {
    myint s = 0; for (char c : vs[i]) { s <<= 3; s |= c - 'a'; }
    v[i] = s; h.emplace(s); }

  // Preglejmo vse pare nizov.
  int stTrojic = 0;
  for (int i = 1; i < n; ++i) for (int j = 0; j < i; ++j) {
    myint u = TretjiNiz(v[i], v[j]);
    if (h.count(u) == 1) ++stTrojic; }

  return stTrojic / 3; // Upoštevajmo, da smo vsako trojico šteli trikrat.
}
```

Ker smo uporabili 64-bitna cela števila, lahko z enim številom predstavimo niz dolžine največ $z := \lfloor 64/3 \rfloor = 21$ znakov. Pri naših poskusih z nekaj tisoč nizi dolžine 21 znakov je bila pravkar opisana funkcija PrestejTrojice2 približno sedem- do osemkrat hitrejša od PrestejTrojice; pri nizih dolžine 15 je bila približno šestkrat hitrejša, pri nizih dolžine 10 pa tri- do štirikrat hitrejša. Če bi hoteli na ta način obravnavati tudi daljše nize, bi lahko vsak niz dolžine k predstavili z zaporedjem $\lceil k/21 \rceil$ števil (64-bitnih) in potem klicali funkcijo TretjiNiz na istoležnih elementih teh zaporedij.

3. Beg

Nalogo lahko rešimo z iskanjem v širino: najprej poiščemo na zemljevidu začetni položaj (polje z znakom „x“) in ga dodamo v vrsto; nato pa na vsakem koraku vzamemo eno polje iz vrste in dodamo v vrsto njegove štiri sosede, razen smo jih v vrsto dodali že kdaj prej ali pa so na njih ovire. Tako bomo sčasoma pregledali vsa polja, ki so dosegljiva iz začetnega položaja; in pregledali jih bomo po naraščajoči oddaljenosti (merjeno s številom korakov) od začetnega položaja. Prvo med njimi, ki je dovolj oddaljeno od začetnega položaja (po evklidski razdalji), da ga dron ne bo mogel doseči, je torej ravno polje, po katerem sprašuje naloga; koristno je torej vsakič, preden dodamo novo polje v vrsti, preveriti ta pogoj in če je izpolnjen, lahko z iskanjem v širino takoj končamo in vrnemo pravkar odkrito polje.

Da ne bomo istega polja po večkrat dodajali v vrsto, si spodnja rešitev v vektorju vVrsti zapisuje, katera polja je že kdaj dodala v vrsto. Še ena možnost bi bila, da bi spreminjali sam zemljevid: ko dodamo neko polje v vrsto, ga v zemljevidu spremenimo na „#“; tako ga kasneje nikoli več ne bomo poskušali dodati v vrsto, ker ga bomo šteli za neprehodno.

Še eno vprašanje je, kako naj vemo, na kakšni oddaljenosti (merjeno s številom korakov) od začetnega položaja leži neko polje; to je namreč vrednost, ki jo bomo morali na koncu tudi vrniti. Lahko bi te podatke hranili v vrsti skupaj s številko posameznega polja; lahko bi imeli ločen vektor ali tabelo, kjer bi za vsako polje zemljevida zapisali oddaljenost, ko tisto polje prvič dosežemo. Še ena možnost pa je naslednja (ki smo jo uporabili tudi v našem spodnjem programu): pri iskanju v širino vedno velja, da je prvih nekaj polj v vrsti na neki oddaljenosti d, preostala pa so na oddaljenosti $d + 1$. Dovolj je torej, če vzdržujemo trenutni d — to je oddaljenost prvega polja v vrsti od začetnega položaja — in številko prvega takega polja v vrsti, ki je na oddaljenosti $d + 1$ (spodnji program jo hrani v spremenljivki dNasl).

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

// Vsak niz v vektorju „zemljevid“ predstavlja eno vrstico zemljevida.
int KolikoKorakov(int k, const vector<string> &zemljevid)
{
  if (k == 0) return 0;
  int w = zemljevid[0].length(), h = zemljevid.size();
  vector<bool> vVrsti(w * h, false);
  queue<int> vrsta;

  int d = 0; // število korakov do prvega polja v vrsti
  int dNasl = -1; // prvo polje v vrsti na oddaljenosti d + 1 korakov

  // Poiščimo začetni položaj in ga dodajmo v vrsto.
  int x0 = -1, y0 = -1;
  for (int y = 0; y < h; ++y) for (int x = 0; x < w; ++x)
    if (zemljevid[y][x] == 'x') {
      vrsta.emplace(y * w + x); vVrsti[y * w + x] = true;
      x0 = x; y0 = y; break; }

  // Preiskujmo zemljevid v širino.
  const int DX[] = {-1, 1, 0, 0}, DY[] = {0, 0, -1, 1};
  while (! vrsta.empty())
  {
    // Vzemimo naslednje polje iz vrste.
    int z = vrsta.front(); vrsta.pop();
    int x = z % w, y = z / w;

    // Ali se tu število korakov od začetnega polja poveča?
    if (z == dNasl) ++d, dNasl = -1;

    // Preglejmo sosede tega polja.
    for (int smer = 0; smer < 4; ++smer)
    {
      int xx = x + DX[smer], yy = y + DY[smer];

      // Če ta sosed ni prehodno polje ali pa smo ga že videli, ga preskočimo.
      if (xx < 0 || xx >= w || yy < 0 || yy >= h) continue;
      if (zemljevid[yy][xx] != '.') continue;
      int zz = yy * w + xx; if (vVrsti[zz]) continue;

      // Če je dovolj daleč od začetnega polja, smo našli rešitev.
      if ((xx - x0) * (xx - x0) + (yy - y0) * (yy - y0) > k * k) return d + 1;

      // Sicer ga dodajmo v vrsto.
      vrsta.emplace(zz); vVrsti[zz] = true;
      if (dNasl < 0) dNasl = zz;
    }
```

```
  }
  return −1; // Če pridemo do sem, rešitve nismo našli.
}
```

4. Tovarna

K prvemu stroju lahko prinašata opilke le prva dva delavca; prvi delavec lahko poleg tega prinaša opilke tudi k drugemu stroju, drugi delavec pa k drugemu in tretjemu stroju. Koristno je torej, če k prvemu stroju prinese čim več opilkov prvi delavec, ker bo tako drugemu delavcu ostalo več opilkov, ki jih lahko morda koristno uporabi pri tretjem stroju (kjer mu prvi delavec ne more pomagati).

Če s tako prejetimi opilki prvi stroj še ni polno zaseden, je spet koristno, če mu čim več opilkov prinese še drugi delavec; nobene koristi ni od tega, da bi drugi delavec hranil nekaj opilkov za drugi in tretji stroj, kajti pri tisth dveh mu lahko pomaga tudi tretji delavec, pri prvem stroju pa mu ne more pomagati nihče več (prvi delavec je tja že prinesel največ, kar se je dalo).

Ker ni več nobenega delavca, ki bi lahko še kaj prinesel prvemu stroju, razmislimo zdaj o drugem stroju. Prvi delavec lahko prinaša opilke le še njemu, tako da je najbolje, če mu prinese največ, kar je še mogoče, saj bodo drugače ti njegovi opilki ostali neizkoriščeni.

Zdaj smo ostali brez prvega stroja in tudi brez prvega delavca, tako da imamo pred seboj problem točno enake oblike kot na začetku, le z enim delavcem manj in enim strojem manj; nadaljujemo lahko torej po enakem razmisleku kot doslej. Tako smo dobili naslednji požrešni algoritem (z a_{ij} bomo označili količino opilkov, ki jih delavec i prinese stroju j):

$A := 0$;
for $i := 1$ **to** n:
  (* *Delavec i prinese čim več opilkov stroju i.* *)
  $a_{ii} := \min\{y_i, x_i\}$; $x_i \mathrel{-}= a_{ii}$; $y_i \mathrel{-}= a_{ii}$; $A \mathrel{+}= a_{ii}$;
  if $i < n$:
    (* *Delavec $i+1$ prinese čim več opilkov stroju i.* *)
    $a_{i+1,i} := \min\{y_{i+1}, x_i\}$; $x_i \mathrel{-}= a_{i+1,i}$; $y_{i+1} \mathrel{-}= a_{i+1,i}$; $A \mathrel{+}= a_{i+1,i}$;
    (* *Delavec i prinese čim več opilkov stroju $i+1$.* *)
    $a_{i,i+1} := \min\{y_i, x_{i+1}\}$; $x_{i+1} \mathrel{-}= a_{i,i+1}$; $y_i \mathrel{-}= a_{i,i+1}$; $A \mathrel{+}= a_{i,i+1}$;
return A;

Na koncu tega postopka nam vrednosti a_{ij} povedo, koliko opilkov prinesejo delavci posameznim strojem, v x_j nam ostane še prosta kapaciteta stroja j (koliko opilkov bi še lahko sprejel), v y_i pa prosta kapaciteta delavca i (koliko opilkov bi še lahko prenesel). Vsota vseh a_{ij}, ki smo jo izračunali v A, pa je skupna količina proizvedenih žebljev — to je rezultat, po katerem sprašuje naloga.

Prepričajmo se, da naš postopek res vedno najde najboljšo rešitev (torej tako z največjo skupno proizvodnjo A). Pa recimo, da ni tako; vzemimo neki tak primerek našega problema, pri katerem je rešitev našega požrešnega postopka slabša od optimalne. Število opilkov, ki jih prinese delavec i stroju j v požrešni rešitvi, označimo z a_{ij}, v optimalni rešitvi pa z $\hat{a}_{ij}$.

Naš postopek je po vrsti računal vrednosti a_{11}, a_{21}; a_{12}, a_{22}, a_{32}; a_{23}, a_{33}, a_{43}; in tako naprej, torej računamo a_{ij} po naraščajočih j, pri posameznem j pa po naraščajočih i. Prvih nekaj vrednosti v tem zaporedju se morda ujema s tistimi iz optimalne rešitve, prej ali slej pa mora nastopiti razlika: $a_{ij} \neq \hat{a}_{ij}$. Ker naš postopek izbere a_{ij} tako, da v celoti zasede kapaciteto, ki je takrat še ostala delavcu i in/ali stroju j (odvisno od tega, kateri od njiju ima manj proste kapacitete), je lahko $\hat{a}_{ij}$ le manjši od a_{ij}, večji pa ne more biti; recimo, da je manjši za $d := a_{ij} - \hat{a}_{ij}$. Po dodelitvi vrednosti $\hat{a}_{ij}$ je delavcu i in stroju j ostalo v optimalni rešitvi za d več proste kapacitete, kot sta jo imela v naši rešitvi po dodelitvi vrednosti a_{ij}.

(1) Če je $i = j$: vidimo torej, da je $a_{ii} = d + \hat{a}_{ii}$. V isti iteraciji glavne zanke je nato naš postopek določil še vrednosti $a_{i,i+1}$ in $a_{i+1,i}$.

(1.1) Ali je mogoče, ali bi zanju veljalo $a_{i,i+1} \geq \hat{a}_{i,i+1}$ in $a_{i+1,i} \geq \hat{a}_{i+1,i}$? To bi pomenilo, da tistih d enot proste kapacitete, ki so pri optimalni rešitvi ostale delavcu i

in stroju i po dodelitvi vrednosti $\hat{a}_{ii}$, ostane proste tudi po dodelitvi vrednosti $\hat{a}_{i,i+1}$ in $\hat{a}_{i+1,i}$; po tistem pa z delavcu i in stroju i ne dodelimo ničesar več, torej tistih d enot proste kapacitete ostane do konca. Optimalno rešitev bi torej lahko izboljšali, če bi $\hat{a}_{ii}$ povečali za d, kar pa je protislovje.

(1.2) Recimo, da je $a_{i,i+1} \geq \hat{a}_{i,i+1}$. Potem mora gotovo veljati $a_{i+1,i} < \hat{a}_{i+1,i}$, sicer bi zapadli pod točko (1.1); recimo, da je $\hat{a}_{i+1,i} = d' + \hat{a}_{i+1,i}$. (1.2.1) Če je $d' < d$, lahko v optimalni rešitvi povečamo $\hat{a}_{ii}$ za d' in zmanjšamo $\hat{a}_{i+1,i}$ za d'; rešitev ostane veljavna in enako dobra, torej še vedno optimalna, vendar pa zdaj zapade pod točko (1.1), torej smo v protislovju. (1.2.2) Če pa je $d' \geq d$, lahko v optimalni rešitvi povečamo $\hat{a}_{ii}$ za d in zmanjšamo $\hat{a}_{i+1,i}$ za d; rešitev ostane veljavna in enako dobra, vendar pa zanjo zdaj velja $\hat{a}_{ii} = a_{ii}$; prva razlika v primerjavi z rešitvijo našega postopka mora torej nastopiti kasneje in lahko z razmislekom nadaljujemo tam.

(1.3) Ostane še možnost, da je $a_{i,i+1} < \hat{a}_{i,i+1}$; razliki med desno in levo stranjo recimo d'', torej je $\hat{a}_{i,i+1} = d'' + a_{i,i+1}$. (1.3.1) Če je $d'' < d$, lahko v optimalni rešitvi povečamo $\hat{a}_{ii}$ za d'' in zmanjšamo $\hat{a}_{i,i+1}$ za d''; rešitev ostane veljavna in enako dobra, torej še vedno optimalna, vendar zdaj zanjo velja $\hat{a}_{i,i+1} = a_{i,i+1}$, torej zapade pod točko (1.2) in lahko z razmislekom nadaljujemo tam. (1.3.2) Če pa je $d'' \geq d$, lahko v optimalni rešitvi povečamo $\hat{a}_{ii}$ za d in zmanjšamo $\hat{a}_{i,i+1}$ za d; rešitev ostane veljavna in enako dobra, vendar pa zanjo zdaj velja $\hat{a}_{ii} = a_{ii}$; prva razlika v primerjavi z rešitvijo našega postopka mora torej nastopiti kasneje in lahko z razmislekom nadaljujemo tam.

(2) Če je $i = j + 1$: vidimo torej, da je $a_{j+1,j} = d + \hat{a}_{j+1,j}$. Ker je prva razlika med rešitvama nastopila šele tu, se morata do vključno vrednosti a_{jj} še ujemati; velja torej $a_{jj} = \hat{a}_{jj}$ in nekoč pred tem tudi $a_{j-1,j} = \hat{a}_{j-1,j}$. Stroj j je torej v optimalni rešitvi dobil d opilkov manj kot v naši požrešni rešitvi. Če v optimalni rešitvi delavec $j + 1$ prinese manj opilkov, kot jih zmore (torej manj kot y_{j+1}), bi lahko to rešitev izboljšali tako, da bi povečali $\hat{a}_{j+1,j}$; to bi bilo protislovje. Torej v optimalni rešitvi delavec $j + 1$ prinese natanko y_{j+1} opilkov; ker jih prinese stroju j za d manj kot v požrešni rešitvi, jih mora zato prinesti strojema $j + 1$ in $j + 2$ vsaj za d več kot v požrešni rešitvi. V optimalni rešitvi lahko torej $\hat{a}_{j+1,j+1}$ in $\hat{a}_{j+1,j+2}$ zmanjšamo tako, da se bo njuna vsota zmanjšala za d, nato pa $\hat{a}_{j+1,j}$ povečamo za d; rešitev bo še vedno veljavna in enako dobra kot prej, torej še vedno optimalna, obenem pa se bo s požrešno zdaj ujemala tudi pri $a_{j+1,j} = \hat{a}_{j+1,j}$. Prvo neujemanje med njima torej zdaj nastopi kasneje kot prej in lahko z razmislekom nadaljujemo tam.

(3) Ostane še primer, ko je $i = j - 1$; torej je $a_{i,i+1} = d + \hat{a}_{i,i+1}$. Ker je prva razlika med rešitvama nastopila tu, se morata pred tem še ujemati, med drugim pri $a_{ii} = \hat{a}_{ii}$ in $a_{i,i-1} = \hat{a}_{i,i-1}$. Delavec i je torej v optimalni rešitvi prinesel za d opilkov manj kot v naši požrešni. Če stroj $i + 1$ v optimalni rešitvi prejme manj opilkov, kot jih zmore obdelati, torej manj kot x_{i+1}, bi se dalo optimalno rešitev izboljšati, če bi v njej $\hat{a}_{i,i+1}$ povečali; to bi bilo protislovje. Stroj $i + 1$ v optimalni rešitvi prejme natanko x_{i+1} opilkov. Ker jih od delavca i prejme d manj kot pri požrešni rešitvi, jih mora od delavcev $i+1$ in $i+2$ prejeti vsaj d več kot pri požrešni rešitvi. Torej lahko v optimalni rešitvi zmanjšamo $\hat{a}_{i+1,i+1}$ in $\hat{a}_{i+2,i+1}$ tako, da se njuna vsota zmanjša za d, nato pa povečamo $\hat{a}_{i,i+1}$ za d. Rešitev bo še vedno veljavna in enako dobra kot prej, torej še vedno optimalna, obenem pa se bo s požrešno zdaj ujemala tudi pri $a_{i,i+1} = \hat{a}_{i,i+1}$. Prvo neujemanje med njima torej zdaj nastopi kasneje kot prej in lahko z razmislekom nadaljujemo tam.

Vidimo torej, da lahko optimalno rešitev vedno popravimo tako, da poiščemo prvo neujemanje med njo in požrešno rešitvijo in ga odpravimo, ne da bi se rešitev pri tem kaj poslabšala (in ne da bi nastalo kakšno novo neujemanje pred tistim, ki smo ga pravkar odpravili). Tako lahko korak za korakom predelamo optimalno rešitev v požrešno, ne da bi se rešitev kdaj poslabšala; torej je bila požrešna rešitev že tudi optimalna. □

5. Chordpro

Vhodne podatke berimo vrstico po vrstico. Pri vsaki vrstici se v zanki zapeljimo po znakih in jih kopirajmo v dva niza, enega za akorde in enega za običajno besedilo. Ko pridemo do konca vhodne vrstice, izpišemo na izhod najprej niz z akordi in nato niz z besedilom, vsakega v svojo vrstico. Če je niz z akordi prazen, ga seveda ne izpišemo.

Pri kopiranju znakov iz vhodnega niza v izhodna niza pazimo še na to, da morajo biti akordi pravilno poravnani z besedilom. Ko pridemo v vhodnem nizu do oglatega

oklepaja, ki označuje začetek akorda, dodajmo na konec niza z akordi toliko presledkov, da bo enako dolg kot niz z besedilom; tako se bo akord začel točno nad tistim znakom besedila, pred katerim stoji v vhodnem nizu.

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

int main()
{
  int n; cin >> n; // Preberimo število vrstic.
  string s; getline(cin, s); // Preberimo konec vrstice, v kateri je bil n.
  while (n-- > 0)
  {
    getline(cin, s); // Preberimo naslednjo vrstico.
    string a, b; // Ločili jo bomo na akorde a in besedilo b.
    for (int i = 0, d = s.length(); i < d; )
    {
      // Zunaj akordov kopirajmo znake s-ja v b.
      if (s[i] != '[') { b += s[i++]; continue; }
      // Ko pridemo do akorda, dodajmo v a toliko presledkov,
      // da bo enako dolg kot b; tako bo akord pravilno poravnan.
      int j = b.length();
      while (a.length() < j) a += ' ';
      // Znake akorda skopirajmo iz s v a.
      for (++i; i < d && s[i] != ']'; ++j, ++i) a += s[i];
      ++i; // Preskočimo oglati zaklepaj na koncu akorda.
    }
    if (! a.empty()) cout << a << endl; // Izpišimo akorde (če jih je kaj bilo).
    cout << b << endl; // Izpišimo vrstico z besedilom.
  }
  return 0;
}
```

Še ena možnost je, da se (za vsako vrstico vhodnih podatkov) po črkah vhodnega niza dvakrat zapeljemo v zanki. Pri prvem prehodu izpisujemo akorde, pri tem pa ločemo štejemo izpisane znake ter znake običajnega besedila (ki jih ne izpisujemo); s tema števcema (v spodnjem programu sta to spremenljivki a in b) si pomagamo, da na začetku akorda izpišemo še primerno število presledkov, da bo akord pravilno poravnan z besedilom. V drugem prehodu čez vhodni niz izpišemo znake besedila, akorde pa le preskočimo.

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

int main()
{
  int n; cin >> n; // Preberimo število vrstic.
  string s; getline(cin, s); // Preberimo konec vrstice, v kateri je bil n
  while (n-- > 0)
  {
    getline(cin, s); // Preberimo naslednjo vrstico.
    int a = 0, b = 0, d = s.length();
    // Izpišimo akorde.
    for (int i = 0; i < d; ) {
      // Znake besedila le štejmo.
      if (s[i++] != '[') { ++b; continue; }
      // Izpišimo presledke, da bo akord poravnan z besedilom.
      while (a < b) cout << ' ', ++a;
      // Izpišimo znake trenutnega akorda.
```

```
      while (s[i] != ']') { cout << s[i++]; ++a; }
      // Preskočimo oglati zaklepaj.
      ++i; }
    if (a > 0) cout << endl;

    // Izpišimo besedilo.
    for (int i = 0; i < d; ++i) {
      if (s[i] != '[') { cout << s[i]; continue; }
      // Preskočimo akorde.
      while (s[i] != ']') ++i; }
    cout << endl;
  }
  return 0;
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Trki

To, kako naj simuliramo gibanje delca, nam pravzaprav precej podrobno opisuje že besedilo naloge; v zanki lahko delec izmenično premikamo vodoravno in navpično, po vsakem premiku pa razpolovimo hitrost v tisti smeri; ko obe hitrosti padeta na 0, se ustavimo. Edino, česar nam naloga ne pove, je to, kako ugotoviti, kako daleč se bo delec na posameznem koraku premaknil; z drugimi besedami, kje bo prva ovira, na katero bo naletel (če sploh bo naletel na oviro).

Pri tem si lahko pomagamo z bisekcijo. Recimo, da razmišljamo o premiku v desno. Vemo, da se lahko premaknemo vsaj 0 enot daleč, in vemo, da se ne moremo premakniti $v_x + 1$ enot daleč; med tema dvema mejama lahko zdaj najdaljši možni premik poiščemo z bisekcijo. Pri tem vzdržujemo spodnjo mejo d_1 in zgornjo mejo d_2, za kateri vedno velja, da je premik dolžine d_1 mogoč, premik dolžine d_2 pa ne. Na vsakem koraku bisekcije vzamemo d na pol poti med spodnjo in zgornjo mejo ter preverimo, ali je mogoč premik dolžine d, torej ali obstaja kakšna ovira v pravokotniku $[x, x+d] \times [y, y]$; če je premik mogoč, dvignemo spodnjo mejo na d, sicer pa spustimo zgornjo mejo na d. To ponavljamo, dokler se meji ne zbližata: ko velja $d_2 = d_1 + 1$, takrat vemo, da je premik dolžine d_1 mogoč, premik dolžine $d_2 = d_1 + 1$ pa ne, torej je d_1 točno prava dolžina premika. Doslej smo govorili o premiku v desno, na enak način pa seveda lahko obravnavamo tudi premike v druge smeri.

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

bool SoOvire(int x1, int y1, int x2, int y2)
{
  // Pošljimo ocenjevalnemu sistemu poizvedbo.
  cout << "? " << min(x1, x2) << " " << min(y1, y2) << " " <<
    max(x1, x2) << " " << max(y1, y2) << endl << flush;
  // Preberimo odgovor.
  string s; cin >> s; return s == "DA";
}

int main()
{
  // Preberimo začetno stanje.
  int x, y, vx, vy; cin >> x >> y >> vx >> vy;
  // Odsimulirajmo gibanje delca.
  while (vx != 0 || vy != 0)
  {
    if (vx != 0) // Premik gor/dol.
```

```
    {
      int smer = (vx > 0) ? 1 : -1;
      int d1 = 0, d2 = vx * smer + 1;
      while (d2 - d1 > 1) {
        // Vemo, da se lahko premaknemo za d1 daleč, ne pa za d2.
        int d = (d1 + d2) / 2;
        if (SoOvire(x, y, x + smer * d, y)) d2 = d; else d1 = d; }
      // Vemo, da se lahko premaknemo za d1 daleč, ne pa za d1 + 1.
      x += smer * d1; vx /= 2;
    }
    if (vy != 0) // Premik levo/desno.
    {
      int smer = (vy > 0) ? 1 : -1;
      int d1 = 0, d2 = vy * smer + 1;
      while (d2 - d1 > 1) {
        // Vemo, da se lahko premaknemo za d1 daleč, ne pa za d2.
        int d = (d1 + d2) / 2;
        if (SoOvire(x, y, x, y + smer * d)) d2 = d; else d1 = d; }
      // Vemo, da se lahko premaknemo za d1 daleč, ne pa za d1 + 1.
      y += smer * d1; vy /= 2;
    }
  }
  // Izpišimo rezultat.
  cout << x << " " << y << endl << flush; return 0;
}
```

Razmislimo še o tem, koliko poizvedb bo ta postopek izvedel v najslabšem primeru (spomnimo se, da jih smemo izvesti največ 2000). Razmišljajmo le o x-koordinatah, kajti za y-koordinate je vse enako in bomo na koncu število poizvedb le podvojili.

Recimo, da je hitrost v trenutni smeri enaka v; bisekcijo torej začnemo z $d_1 = 0$ in $d_2 = v + 1$, razlika $r := d_2 - d_1$ je enaka $v + 1$. Na vsakem koraku bisekcije se razlika zmanjša z r na $\lfloor r/2 \rfloor$ ali $\lceil r/2 \rceil$, odvisno od tega, katero mejo premaknemo. Z indukcijo po r se lahko prepričamo, da če smo bisekcijo začeli z razliko r, se bo izvedlo v najslabšem primeru $\lceil \log_2 r \rceil$ iteracij; v našem primeru je to naprej enako $\lceil \log_2(v+1) \rceil$.

Naloga pravi, da za začetno stanje velja $|x| \le M$ in $|x + 2v_x| \le M$ za $M = 10^9$. Iz tega sledi, da je tudi $|v_x| \le M$. Največja možna hitrost pri prvem premiku je torej M, od tam naprej pa se hitrost vsakič razpolovi po formuli $v \mapsto \lfloor v/2 \rfloor$. Po t premikih imamo torej hitrost $\lfloor M/2^t \rfloor$. Pri $t = T + 1$ za $T = \lfloor \log_2 M \rfloor$ je ta hitrost gotovo že 0 in premikanje se ustavi. Skupno število iteracij bisekcije (in s tem skupno število poizvedb) je zato kvečjemu

$$\begin{aligned}
&\sum_{t=0}^{T} \lceil \log_2(1 + \lfloor M/2^t \rfloor) \rceil \le \sum_{t=0}^{T} \lceil \log_2(1 + M/2^t) \rceil \\
&= \sum_{t=0}^{T} \lceil \log_2 \left((M + 2^t)/2^t \right) \rceil = \sum_{t=0}^{T} \lceil \left(\log_2(M + 2^t) \right) - t) \rceil = \sum_{t=0}^{T} \left(\lceil \log_2(M + 2^t) \rceil - t \right) \\
&= \left(\sum_{t=0}^{T} \lceil \log_2(M + 2^t) \rceil \right) - \sum_{t=0}^{T} t = \left(\sum_{t=0}^{T} \lceil \log_2(M + 2^t) \rceil \right) - T(T+1)/2.
\end{aligned}$$

V našem primeru je $M = 10^9$, kar leži med 2^{29} in 2^{30}, zato je $T = 29$. M-ju do 2^{30} manjka še dobrih 73 milijonov, kar leži med 2^{26} in 2^{27}; pri $t = 0, \ldots, 25$ je zato $M + 2^t$ še manjši od 2^{30} in za $\lceil \log_2(M + 2^t) \rceil$ dobimo 30; pri $t = 26, \ldots, T$ pa že dobimo $\lceil \log_2(M + 2^t) \rceil = 31$. Omenjena vsota je zato naprej enaka $26 \cdot 30 + 4 \cdot 31 - 29 \cdot 30/2 = 469$. Ker smo doslej šteli le premike v vodoravni smeri, moramo to še podvojiti in dobimo 938; vidimo torej, da bomo gotovo porabili manj kot polovico od dovoljenih 2000 poizvedb.

2. Steklenice

Za začetek našega razmisleka odmislimo, da so steklenice v paketih; recimo, da imamo

m steklenic, ki jih uredimo naraščajoče po odtenku, tako da ima i-ta najsvetlejša med njimi odtenek $\hat{o}_i$ (lahko ima seveda več zaporednih enak odtenek).

Nalogo lahko rešujemo s požrešnim algoritmom. Steklenice obravnavajmo po naraščajočem odtenku, pri vsaki steklenici $\hat{o}_i$ pa med tistimi naročili, ki bi lahko sprejela to steklenico (torej za katera velja $a_j \leq \hat{o}_i \leq b_j$ in ki še niso povsem izpolnjena), izberimo tisto z najnižjim b_j ter mu dodelimo trenutno steklenico. Če naročil, ki bi ustrezala opisanemu pogoju, sploh ni, naj ta steklenica pač ostane neuporabljena. Ko na ta način obdelamo vse steklenice, moramo le še pri vsakem naročilu sešteti, koliko steklenic mu še manjka do s_j, in vrniti vsoto tega po vseh naročilih.

Prepričajmo se, da je ta rešitev res najboljša možna. Pa recimo, da ni tako in da je najboljša rešitev (taka, ki uporabi največje možno število izmed m obstoječih steklenic) drugačna od požrešne rešitve. Posamezno rešitev lahko opišemo s tem, da za vsako steklenico povemo, kateremu naročilu (če sploh kateremu) je bila dodeljena. Požrešna in najboljša rešitev se pri prvih nekaj steklenicah morda ujemata, prej ali slej pa mora nastopiti razlika; recimo, da prva razlika nastopi pri i-ti steklenici (tisti z odtenkom $\hat{o}_i$). Če obstaja več (enako dobrih) najboljših rešitev, vzemimo med njimi tisto z največjim i (torej tisto, pri kateri prvo neujemanje nastopi najkasneje). Ker se do sem obe rešitvi ujemata, imata obe enake možnosti glede tega, katera naročila bi lahko sprejela i-to steklenico.

Ali je mogoče, da požrešna rešitev te steklenice ne dodeli nobenemu naročilu? Tako bi naredila le, če je nobeno naročilo ne bi moglo sprejeti; toda tedaj bi enako veljalo tudi pri najboljši rešitvi in tudi ona ne bi dodelila te steklenice nobenemu naročilu; potem pa se obe rešitvi pri tej steklenici sploh ne bi razlikovali, kar je v protislovju s tem, kako smo to steklenico sploh izbrali. Neizogibno je torej, da požrešna rešitev to steklenico dodeli nekemu naročilu, recimo j-temu. Najboljša rešitev pa, ker tu razlikuje od požrešne, steklenice i bodisi sploh ne uporabi bodisi jo dodeli nekemu drugemu naročilu $j' \neq j$.

(1) Če pri najboljši rešitvi j-to naročilo na koncu ni povsem izpolnjeno, lahko to rešitev spremenimo tako, da i-to steklenico dodelimo j-temu naročilu. Če je bila i-ta steklenica pred to spremembo v najboljši rešitvi sploh neuporabljena, se je rešitev s tem izboljšala, kar je protislovje; torej je v resnici morala biti ta steklenica prej dodeljena nekemu drugemu naročilu j', s premikom te steklenice na naročilo j pa ostane enako dobra, torej še vedno najboljša, vendar pa se zdaj s požrešno rešitvijo ujema še v eni steklenici več kot prej, to pa je v protislovju s predpostavko, da smo med najboljšimi rešitvami prej izbrali tisto z največjim i (torej največjim indeksom prvega neujemanja s požrešno rešitvijo).

(2) Ostane še možnost, da je pri najboljši rešitvi j-to naročilo na koncu povsem izpolnjeno. Ker je požrešna rešitev dodelila i-to steklenico naročilu j, to pomeni, da je to naročilo takrat (pred dodelitvijo te steklenice) še ni bilo povsem izpolnjeno; enako je torej veljalo tudi pri najboljši rešitvi (saj se do pred steklenice i obe rešitvi ujemata); ker pa ona ni dodelila steklenice i naročilu j, to naročilo tudi po obravnavi i-te steklenice še ni bilo v celoti izpolnjeno. Ker pa smo videli, da je na koncu pri tej rešitvi to naročilo vendarle izpolnjeno, to pomeni, da mu je nekoč kasneje dodelila neko drugo steklenico $i' > i$. Ker so steklenice urejene po odtenku, velja $\hat{o}_{i'} \geq \hat{o}_i$.

(2.1) Če najboljša rešitev ni dodelila steklenice i nobenemu naročilu, lahko to rešitev spremenimo tako, da steklenico i dodelimo naročilu j, steklenice i' pa po novem ne dodelimo nobenemu naročilu. Rešitev ostane veljavna in enako dobra kot prej, torej še vedno najboljša, obenem pa se s požrešno ujema v eni steklenici več, kar je protislovje.

(2.2) Sicer pa je najboljša rešitev dodelila steklenico i nekemu drugemu naročilu j'. Spomnimo se, da imata požrešna in najboljša rešitev pri steklenici i na voljo obe ista naročila in da požrešna rešitev med njimi izbere tisto z najmanjšo b_j. Ker je najboljša rešitev izbrala neko drugo naročilo j', mora za le-to očitno veljati $b_{j'} \geq b_j$. Spremenimo požrešno rešitev tako, da steklenico i' preselimo iz j v j', steklenico i pa iz j' v j. Za steklenico j že vemo, da ustreza naročilu i; prepričajmo se še, da tudi steklenica i' ustreza naročilu j'. Ker je najboljša rešitev prej dodelila steklenico i' naročilu j, je očitno moralo veljati $\hat{o}_{i'} \leq b_j$, kar nam skupaj z $b_{j'} \geq b_j$ dá $\hat{o}_{i'} \leq b_{j'}$; in ker je najboljša rešitev dodelila steklenico i naročilu j', je očitno moralo veljati $\hat{o}_i \geq a_{j'}$, kar nam skupaj z $\hat{o}_{i'} \geq \hat{o}_i$ dá $\hat{o}_{i'} \geq a_{j'}$. Obe ugotovitvi lahko združimo v $a_{j'} \leq \hat{o}_{i'} \leq b_{j'}$, torej steklenica i' res ustreza naročilu j'; torej je rešitev po opisani spremembi še vedno veljavna in

enako dobra kot prej, torej še vedno najboljša, poleg tega pa se ujema s požrešno tudi v steklenici i; spet protislovje.

Vidimo torej, da nas predpostavka, da požrešna rešitev ni bila najboljša, neizogibno pripelje v protislovje; torej je požrešna rešitev najboljša. □

Razmislimo zdaj še o tem, kako učinkovito implementirati opisani postopek; zdaj imejmo v mislih tudi to, da bomo morali steklenice obravnavati v paketih, ne pa posamično. Lahko si predstavljamo takšen postopek:

za vsak paket (p_i, o_i) po naraščajoči vrednosti o_i:
  $M :=$ množica naročil, ki lahko sprejmejo steklenice z odtenkom o_i;
  while $p_i > 0$ **and** M ni prazna:
    naj bo (s_j, a_j, b_j) tisto naročilo iz M, ki ima najmanjši b_j;
    $\triangle := \min\{p_i, s_j\}$;
    $s_j := s_j - \triangle$; $p_i := p_i - \triangle$;
    if $s_j = 0$ **then** pobriši naročilo j iz M;

V vsaki iteraciji notranje zanke torej dodelimo $\triangle$ steklenic iz paketa i naročilu j; po tem se bodisi paket izprazni (p_i pade na 0) in zanka se konča bodisi je naročilo povsem izpolnjeno (s_j pade na 0) in ga pobrišemo iz M.

Seveda si ne moremo privoščiti, da bi pri vsakem paketu i pregledali vsa naročila, da bi videli, katera tvorijo množico M; pomagati si moramo z dejstvom, da pakete pregledujemo po naraščajoči o_i. Ko gremo na naslednji paket z večjo o_i, nekatera naročila izpadejo iz M (ker imajo prenizko zgornjo mejo, $b_j < o_i$), vanjo pa pridejo nekatera nova (katerih spodnja meja a_j je bila previsoka za staro o_i, pri novi o_i pa velja $a_j \le o_i$).

Koristno je torej imeti seznam vseh naročil, urejen naraščajoče po a_j; s pomočjo takega seznama bomo zlahka videli, katera naročila pridejo na novo v množico M, ko se premaknemo na naslednji paket. Množico M sámo pa lahko predstavimo s kopico, kjer so naročila z manjšim b_j višje v kopici (v korenu je tisto z najmanjšim b_j). V prejšnjem odstavku smo rekli, da nekatera naročila izpadejo iz M, ko se premaknemo na naslednji paket, vendar jih v resnici ni treba takoj pobrisati iz kopice; to lahko počaka do takrat, ko se bodo znašla v korenu kopice (od tam jih je namreč najlažje pobrisati).

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

typedef int_fast64_t myint;

struct Paket { int pi, oi; };
struct Narocilo { mutable int sj; int aj, bj;
  bool operator < (const Narocilo &N) const { return bj > N.bj; } };

int main()
{
  // Preberimo vhodne podatke.
  int n, k; cin >> n >> k;
  vector<Paket> paketi(n);
  for (auto &P : paketi) cin >> P.pi >> P.oi;
  vector<Narocilo> narocila(k);
  for (auto &N : narocila) cin >> N.sj >> N.aj >> N.bj;
  myint manjkajoce = 0; for (auto &N : narocila) manjkajoce += N.sj;

  // Uredimo pakete po odtenku in naročila po spodnji meji.
  sort(paketi.begin(), paketi.end(), [] (auto &x, auto &y) { return x.oi < y.oi; });
  sort(narocila.begin(), narocila.end(), [] (auto &x, auto &y) { return x.aj < y.aj; });
  priority_queue<Narocilo> M; // Naročila, ki lahko sprejmejo trenutno steklenico.
  int nasl = 0;               // Naslednje naročilo, ki bo prišlo v M.

  // Dodelimo obstoječe steklenice naročilom.
```

```
  for (auto &P : paketi)
  {
    // Nekaj naročil na novo pride v M.
    while (nasl < k && narocila[nasl].aj <= P.oi) M.push(narocila[nasl++]);
    // Razdeljujmo steklenice tega paketa, dokler gre.
    while (P.pi > 0 && ! M.empty())
    {
      // Oglejmo si v M naročilo z najmanjšim bj.
      auto &N = M.top();
      // Morda to naročilo v resnici ne sme več biti v M.
      if (N.bj < P.oi) { M.pop(); continue; }
      // Dodelimo mu čim več steklenic.
      int delta = min(P.pi, N.sj);
      P.pi -= delta; N.sj -= delta; manjkajoce -= delta;
      if (N.sj <= 0) M.pop();
    }
  }
  // Izpišimo, koliko steklenic nam še manjka.
  cout << manjkajoce << endl; return 0;
}
```

3. Zlatarna

Množico vseh verižic označimo z A, vseh predmetov s smaragdom pa z B. Recimo, da se odločimo odnesti natanko r predmetov iz $A \cap B$ (torej takih, ki so hkrati verižice in vsebujejo smaragde); smiselno je seveda vzeti najdražjih toliko predmetov iz omenjene množice. Da bomo izpolnili zahteve naloge, moramo potem vzeti še $\alpha := \max\{a - r, 0\}$ predmetov iz $A - B$ (da bomo skupaj imeli a predmetov iz A) in $\beta := \max\{b - r, 0\}$ predmetov iz $B - A$ (da bomo skupaj imeli b predmetov iz B). Spet bomo seveda vzeli najdražjih α iz $A - B$ ter najdražjih β iz $B - A$. Lahko se tudi izkaže, da to sploh ni mogoče, ker ima $A - B$ morda manj kot α elementov ali pa ima $B - A$ manj kot β elementov; to je znak, da moramo vzeti večji r.

Doslej se nam je nabralo $r + \alpha + \beta$ predmetov; če je to morda večje od k, je to spet znak, da je problem pri tem r-ju nerešljiv in moramo poskusiti z večjim r. Drugače pa moramo zdaj vzeti še $c := k - r - \alpha - \beta$ predmetov izmed vseh tistih, ki niso iz $A \cap B$ (kajti za take smo rekli, da jih hočemo izbrati točno r in prav toliko smo jih res že izbrali) in ki jih doslej še nismo izbrali (ne glede na to, ali pripadajo množici A ali B ali nobeni); recimo tej množici še neizbranih predmetov C. Vzeti moramo torej najdražjih c predmetov iz C. (Tudi tu se lahko zgodi, da v C sploh ni toliko predmetov, kar je znak, da moramo vzeti večji r.)

Da dobimo najboljšo rešitev, moramo preizkusiti vse možne r (od 0 do k, pri čemer se bo, kot smo že videli, pri nekaterih izkazalo, da rešitev ne obstaja); paziti pa moramo, da ne porabimo pri vsakem r preveč časa. Kaj se zgodi, ko povečamo r za 1? Na začetku izberemo iz $A \cap B$ en element več kot prej, zato v nadaljevanju izberemo iz $A - B$ ter iz $B - A$ po en element manj kot prej (razen če že prej nismo izbrali nobenega, torej pri $\alpha = 0$ oz. $\beta = 0$); torej se α in β zmanjšata za 1 ali (redkeje) ostaneta enaka. To tudi pomeni, da v množico neizbranih elementov C prideta (največ) dva nova elementa. Število že izbranih elementov, $r + \alpha + \beta$, se ponavadi zmanjša za 1, lahko pa ostane nespremenjeno ali se celo poveča za 1 (če sta α in β ostala enaka); število c, ki pove, koliko najdražjih elementov iz C moramo še izbrati, pa se bo zato ponavadi povečalo za 1, lahko pa ostane nespremenjeno ali se celo zmanjša za 1.

Koristno je torej imeti za vsako od množic $A \cap B$, $A - B$ in $B - A$ seznam vrednosti elementov v njej, urejen padajoče; tako ne bo težko računati vsote najdražjih nekaj elementov, ko se ta „nekaj“ poveča ali zmanjša za 1. Za množico C pa, ki se bo ves čas po malem spreminjala, potrebujemo neko bolj dinamično strukturo, kjer bomo lahko poceni dodajali elemente, poleg tega pa vzdrževali vsoto največjih c elementov. Primerna struktura je par kopic; v prvi hranimo največjih c elementov, pri čemer so pri vrhu kopice najmanjši med njimi; v drugi kopici pa hranimo vse ostale elemente, pri čemer so pri vrhu kopice največji med njimi. Za prvo kopico tudi vzdržujemo vsoto vseh elementov

v njej (to je potem vsota najdražjih c predmetov iz C). Ko je treba dodati nov element, lahko s pomočjo korenov obeh kopic preverimo, ali je dovolj velik za v prvo kopico, sicer ga dodamo v drugo; če se potem izkaže, da v prvi kopici ni natanko c elementov, jih moramo nekaj preseliti iz ene kopice v drugo (najmanjši element prve kopice preselimo v drugo ali največji element druge kopice v prvo); ker izvedemo po vsakem povečanju r-ja v množici C le $O(1)$ dodajanj in ker se tudi c spremeni le za $O(1)$, bo treba tudi samo $O(1)$ takšnih selitev. Ker vsaka operacija na kopici vzame $O(\log n)$ časa, imamo tako z vsakim r-jem le $O(\log n)$ dela in postopek kot celota ima časovno zahtevnost $O(n \log n)$.

```
#include <cstdio>
#include <vector>
#include <algorithm>
#include <queue>
#include <cstdint>
using namespace std;
typedef int_fast64_t myint;

int c; // Toliko predmetov bi želeli pobrati iz C.
priority_queue<int, vector<int>, greater<int>> Cveliki; // Največjih min(c, |C|) predmetov iz C.
priority_queue<int> Cmajhni; // Ostali predmeti C-ja.
myint vsotaC = 0; // Vsota predmetov iz Cveliki.

// Po potrebi premesti predmete med kopicama, tako da bo v Cveliki točno c predmetov.
void PrerazporediC()
{
  while (! Cveliki.empty() && Cveliki.size() > c) {
    int x = Cveliki.top(); Cveliki.pop(); vsotaC -= x; Cmajhni.emplace(x); }
  while (! Cmajhni.empty() && Cveliki.size() < c) {
    int x = Cmajhni.top(); Cmajhni.pop(); vsotaC += x; Cveliki.emplace(x); }
}

// Dodaj vrednost „v“ v eno od kopic Cveliki in Cmajhni.
void DodajVC(int v)
{
  if (! Cveliki.empty() && v > Cveliki.top()) { Cveliki.emplace(v); vsotaC += v; }
  else Cmajhni.push(v);
  PrerazporediC(); // Poskrbimo, da bo v Cveliki spet c elementov.
}

int main()
{
  // Preberimo vhodne podatke.
  int n, k, a, b; scanf("%d %d %d %d", &n, &k, &a, &b);
  vector<int> vrednosti(n);
  for (auto &vi : vrednosti) scanf("%d", &vi);
  vector<bool> jeVA(n, false), jeVB(n, false);
  int stVerizic; scanf("%d", &stVerizic);
  while (stVerizic-- > 0) { int x; scanf("%d", &x); jeVA[--x] = true; }
  int stSmaragdov; scanf("%d", &stSmaragdov);
  while (stSmaragdov-- > 0) { int x; scanf("%d", &x); jeVB[--x] = true; }

  // Pripravimo sezname predmetov, urejene padajoče po vrednosti.
  vector<int> presek, AbrezB, BbrezA;
  for (int i = 0; i < n; ++i)
    if (jeVA[i]) (jeVB[i] ? presek : AbrezB).emplace_back(vrednosti[i]);
    else if (jeVB[i]) BbrezA.emplace_back(vrednosti[i]);
  sort(presek.begin(), presek.end(), greater<int>());
  sort(AbrezB.begin(), AbrezB.end(), greater<int>());
  sort(BbrezA.begin(), BbrezA.end(), greater<int>());
  int nP = presek.size(), nAB = AbrezB.size(), nBA = BbrezA.size(); // Velikosti množic.

  // Pripravimo rešitev pri r = 0.
  int r = 0, aa = a, bb = b; // Toliko predmetov bi želeli pobrati iz A ∩ B, A − B, B − A,
  c = k − aa − bb;           // toliko pa iz C.
```

```
  // Vsota najdražjih r predmetov iz A ∩ B, najdražjih min(aa, nAB) iz A − B
  // in najdražjih min(bb, nBA) iz B − A.
  myint vsotaP = 0, vsotaAB = 0, vsotaBA = 0;

  for (int i = 0; i < AbrezB.size(); ++i)
    if (i < aa) vsotaAB += AbrezB[i]; else DodajVC(AbrezB[i]);
  for (int i = 0; i < BbrezA.size(); ++i)
    if (i < bb) vsotaBA += BbrezA[i]; else DodajVC(BbrezA[i]);
  for (int i = 0; i < n; ++i) if (! jeVA[i] && ! jeVB[i]) DodajVC(vrednosti[i]);
  myint najResitev = (Cveliki.size() == c && aa <= nAB && bb <= nBA) ?
                                                  vsotaP + vsotaAB + vsotaBA + vsotaC : −1;

  // Preglejmo večje r.
  for (int r = 1; r <= k && r <= nP; ++r)
  {
    // Zdaj izberemo en predmet več iz preseka, zato bo treba izbrati
    vsotaP += presek[r − 1]; −−c;       // enega manj iz A − B, B − A in C.

    // Predmeta, ki ju ne izberemo več iz A − B in B − A, gresta v C.
    if (0 < aa) { ++c; if (−−aa < nAB) {
      vsotaAB −= AbrezB[aa]; DodajVC(AbrezB[aa]); } }
    if (0 < bb) { ++c; if (−−bb < nBA) {
      vsotaBA −= BbrezA[bb]; DodajVC(BbrezA[bb]); } }

    // Poskrbimo, da bomo imeli pri roki vsoto največjih c elementov množice C.
    PrerazporediC();

    // Če je rešitev veljavna in najboljša doslej, si jo zapomnimo.
    if (Cveliki.size() == c && aa <= nAB && bb <= nBA)
      najResitev = max(najResitev, vsotaP + vsotaAB + vsotaBA + vsotaC);
  }

  // Izpišimo rezultat.
  printf("%jd\n", intmax_t(najResitev)); return 0;
}
```

4. Urejanje z lažmi

Recimo za začetek, da v vhodnih podatkih ne bi bilo napake. Zaboje si lahko predstavljamo kot točke grafa; in za vsak par zabojev, če v vhodnih podatkih piše, da je zaboj u lažji od zaboja v, imejmo v grafu usmerjeno povezavo od u do v. V točko u torej kažejo povezave iz vseh zabojev, ki so lažji od u, iz nje pa kažejo povezave na vse zaboje, ki so težji od u. Vhodna stopnja točke u nam torej pove, koliko zabojev je lažjih od u. Najlažji zaboj ima vhodno stopnjo 0, drugi najlažji ima vhodno stopnjo 1 in tako naprej; v splošnem ima k-ti najlažji zaboj vhodno stopnjo $k-1$. Vrstni red zabojev po teži, po katerem sprašuje naloga, lahko dobimo torej preprosto tako, da točke grafa uredimo po njihovi vhodni stopnji. V mislih lahko k-ti najlažji zaboj označimo z u_k.

Razmislimo zdaj o tem, kaj se spremeni, če se v vhodni tabeli pojavi napaka, recimo pri paru zabojev u_i in u_j, kjer je $i < j$. Ker je $i < j$, je zaboj u_i lažji od u_j, torej je bila v prvotnem grafu povezava usmerjena od u_i do u_j, napaka na tem mestu pa pomeni, da ta povezava zdaj kaže v nasprotno smer: $u_j \to u_i$. Zaradi te spremembe se je točki u_i vhodna stopnja povečala za 1, torej z $i-1$ na i; točki u_j pa se je vhodna stopnja zmanjšala za 1, torej z $j-1$ na j. Za nadaljevanje našega razmisleka ločimo nekaj primerov:

(1) Če je $j = i+1$, to pomeni, da je imela točka u_i prej stopnjo $i-1$, zdaj pa ima stopnjo i, medtem ko je imela točka u_{i+1} prej stopnjo i, zdaj pa ima stopnjo $i-1$. Še vedno imamo torej po natanko eno točko vsake možne stopnje od 0 do $n-1$, zato takšnega grafa ne moremo ločiti od grafa za primer, ko v podatkih ni napake. Če torej na vhodu dobimo graf, v katerem imajo točke stopnje $0, 1, 2, \ldots, n-1$, so možne naslednje rešitve: lahko v podatkih ni napak in pravi vrstni red je $u_1, \ldots, u_n$, kjer je (za vsak k) u_k točka z vhodno stopnjo k; lahko pa je (za poljuben i od 1 do $n-1$) napaka med zabojema u_i in u_{i+1} in je pravi vrstni red v resnici $u_1, \ldots, u_{i-1}, u_{i+1}, u_i, u_{i+2}, \ldots, u_n$. To je skupno n možnih rešitev (ena brez napake in $n-1$ z napako), pazimo pa še na to, da jih moramo izpisati največ 10.

(2) Če je $j = i + 2$, to pomeni, da se je točki u_i stopnja povečala z $i - 1$ na i, točki u_{i+2} pa se je zmanjšala z $i+1$ na i. Med njima je točka u_{i+1}, ki je napaka ni prizadela in ima še vedno vhodno stopnjo i. Tudi ostalim točkam se vhodna stopnja ni spremenila: za $k = 1, \ldots, i, i + 3, \ldots, n$ ima točka u_k ima vhodno stopnjo $k - 1$. Ta tip napake torej zlahka prepoznamo po tem, da imajo tri točke enako vhodno stopnjo. V grafu tvorijo cikel: pred nastankom napake smo imeli povezave $u_i \to u_{i+1} \to u_{i+2}$ in $u_i \to u_{i+2}$, pri napaki pa se je slednja povezava obrnila in je nastal cikel $u_i \to u_{i+1} \to u_{i+2} \to u_i$. Ker v vhodnih podatkih vidimo le stanje po nastanku napake, pa zdaj ne moremo več ugotoviti, katera od treh povezav na ciklu je bila napačna.[2] Tako so možne tri rešitve; v vseh se vrstni red začne z zaboji $u_1, \ldots, u_{i-1}$ in konča z zaboji $u_{i+3}, \ldots, u_n$, vmes pa so trije zaboji s cikla v enem od treh možnih vrstnih redov: bodisi u_i, u_{i+1}, u_{i+2} (pri čemer je v podatkih napaka pri povezavi $u_{i+2} \to u_i$) bodisi u_{i+1}, u_{i+2}, u_i (napaka pa je na povezavi $u_i \to u_{i+1}$) bodisi u_{i+2}, u_i, u_{i+1} (napaka pa je na povezavi $u_{i+1} \to u_{i+2}$).

(3) Ostane še možnost, da je $j \geq i + 3$. Točki u_i se je stopnja povečala z $i - 1$ na i, tako da imamo zdaj v grafu dve točki s stopnjo i (poleg u_i je taka še u_{i+1}, ki je imela to stopnjo že od prej); točki u_j pa se je stopnja zmanjšala z $j - 1$ na $j - 2$, tako da imamo zdaj v grafu tudi dve točki s stopnjo $j - 2$ (poleg u_j je taka še u_{j-1}, ki je imela to stopnjo že od prej). Ta primer torej prepoznamo po tem, da imamo dva para točk z enako stopnjo: en par, z nižjo stopnjo (namreč i), tvorita točki u_i in u_{i+1}, drugi par, z višjo stopnjo (namreč $j - 2$), pa tvorita točki u_{j-1} in u_j. V prvotnem grafu, pred nastankom napake, so bile med tema paroma točk povezave $u_i \to u_{j-1}$, $u_i \to u_j$, $u_{i+1} \to u_{j-1}$ in $u_{i+1} \to u_j$; zdaj pa se je zaradi napake povezava $u_i \to u_j$ obrnila v $u_j \to u_i$. Ta povezava torej zdaj kaže iz para z višjo stopnjo v par z nižjo stopnjo, ostale tri povezave pa torej še vedno kažejo iz para z nižjo v par z višjo stopnjo. Tako ni težko ugotoviti, katera povezava je napačna: to je edina povezava, ki kaže iz para z višjo v par z nižjo stopnjo. Ko tisto povezavo v mislih obrnemo, dobimo spet prvotni graf brez napak, v katerem je za vsako stopnjo od 0 do $n-1$ prisotna natanko ena točka. Pri tem tipu napake obstaja torej natanko ena rešitev.

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

int n; vector<string> tabela; // Vhodni podatki.
enum { MaxIzpisanihResitev = 10 };

bool Lazji(int u, int v) { // Ali je zaboj u lažji od zaboja v?
  return (u < v) ? tabela[v - 1][u] == '<' : tabela[u - 1][v] == '>'; }

void IzpisiResitev(int u, int v, const vector<int> &L)
{
  cout << min(u, v) + 1 << ' ' << max(u, v) + 1;
  for (auto w : L) cout << ' ' << w + 1;
  cout << endl;
}

int main()
{
  // Preberimo vhodne podatke.
  cin >> n; tabela.resize(n); for (auto &vrstica : tabela) cin >> vrstica;
  // Izračunajmo vhodne stopnje točk.
  vector<int> D(n, 0); // D[u] = vhodnja stopnja točke u.
  for (int v = 1; v < n; ++v) for (int u = 0; u < v; ++u)
    ++D[tabela[v - 1][u] == '<' ? v : u];
```

[2] Pri pregledovanju tega cikla moramo paziti še na naslednjo podrobnost: točke cikla moramo imeti zapisane v takem vrstnem redu, v kakrkšnem si sledijo na ciklu, in ne v nasprotnem vrstnem redu. Če na primer točke le uredimo po vhodni stopnji, bodo točke cikla sicer prišle skupaj (ker imajo vse tri enako vhodno stopnjo), vendar ne nujno v pravem vrstnem redu. Takrat na primer preverimo, če je v vhodnih podatkih prva točka cikla manjša od druge; če ni, je to znak, da moramo cikel obrniti. V pravem vrstnem redu je namreč vsaka točka cikla manjša od naslednje.

```
  // Uredimo točke po vhodni stopnji.
  vector<int> L(n); // vrstni red točk po vhodni stopnji
  for (int u = 0; u < n; ++u) L[u] = u;
  sort(L.begin(), L.end(), [&D] (int u, int v) { return D[u] < D[v]; });

  // Ali imajo tri točke enako stopnjo?
  for (int i = 0; i < n - 2; ++i) if (D[L[i]] == D[L[i + 1]] && D[L[i]] == D[L[i + 2]])
  {
    // Postavimo jih v tak vrstni red, da kažejo povezave cikla v smeri
    // L[i] → L[i + 1] → L[i + 2] → L[i] in ne obratno.
    if (! Lazji(L[i], L[i + 1])) swap(L[i], L[i + 1]);

    cout << 3 << endl;
    for (int r = 0; r < 3; ++r) {
      IzpisiResitev(L[i], L[i + 2], L);

      // Zamaknimo cikel za eno mesto, da pripravimo naslednjo rešitev.
      swap(L[i], L[i + 1]); swap(L[i + 1], L[i + 2]); }
    return 0;
  }

  // Ali obstajata dva para točk z enako stopnjo?
  int i = -1;
  for (int j = 1; j < n; ++j) if (D[L[j - 1]] == D[L[j]])
  {
    // Našli smo par točk z enako stopnjo. Če je prvi, si ga le zapomnimo.
    if (i < 0) { i = j - 1; continue; }

    // Če pa je drugi, poglejmo, katera povezava je napačna; to je tista, ki kaže
    // iz para L[j - 1], L[j] v par L[i], L[i + 1].
    int ii = -1, jj = -1;
    for (int ic = i; ic <= i + 1; ++ic) for (int jc = j - 1; jc <= j; ++jc)
      if (Lazji(L[jc], L[ic])) ii = ic, jj = jc;

    // Napačna povezava je med točkama ii in jj. Ko jo obrnemo, ima ii nižjo stopnjo
    // od tiste, s katero je bila prej v paru, jj pa višjo,
    if (ii != i) swap(L[i], L[i + 1]);    // zato mora biti ii prva v svojem paru,
    if (jj != j) swap(L[j - 1], L[j]);    // jj pa druga v svojem.

    // Izpišimo rešitev.
    cout << 1 << endl; IzpisiResitev(L[i], L[j], L); return 0;
  }

  // Sicer imamo n možnih rešitev.
  cout << n << endl;
  IzpisiResitev(-1, -1, L);
  for (int i = 1; i < n && i < MaxIzpisanihResitev; ++i)
  {
    swap(L[i - 1], L[i]);
    IzpisiResitev(L[i - 1], L[i], L);
    swap(L[i - 1], L[i]);
  }
  return 0;
}
```

Gornja rešitev ima časovno zahtevnost $O(n^2)$, bolje kot to pa pri tej nalogi ne gre, saj porabimo toliko časa že samo za branje vhodnih podatkov.

Oglejmo si zdaj še eno rešitev s časovno zahtevnostjo $O(n^2)$; v primerjavi s prvo rešitvijo ima to prednost, da zahteva manj razmišljanja, in to slabost, da postane število rešitev (ki ga moramo izpisati v prvo vrstico izhoda) znano šele na koncu, po tistem, ko vse rešitve tudi zares najdemo in preštejemo; morali si jih bomo torej nekje (vsaj prvih deset) zapomniti, da jih bomo lahko na koncu izpisali.

Spomnimo se, da če v vhodnih podatkih ni napak, nam nastane graf, v katerem za vsako možno vhodno stopnjo od 0 do $n - 1$ obstaja natanko ena točka s takšno vhodno stopnjo; in pravi vrstni red zabojev dobimo preprosto tako, da izpišemo točke naraščajoče po vhodni stopnji.

Vse ostale rešitve lahko poiščemo tako, da se za vsako od $n(n-1)/2$ povezav v grafu

(ali, z drugimi besedami, za vsak par zabojev) vprašamo, ali bi graf dobil obliko, kot smo jo opisali v prejšnjem odstavku, če bi to povezavo obrnili. Vprašanje je le, kako to (za posamezno povezavo) preveriti v $O(1)$ časa, da bomo lahko vseh $O(n^2)$ povezav obdelali v $O(n^2)$ časa.

Za vsako stopnjo d od 0 do $n-1$ naj bo $A[d]$ število točk z vhodno stopnjo natanko d. Teh števil ni težko vzdrževati: ko obrnemo povezavo $u \to v$ in iz nje nastane $v \to u$, se točki u vhodna stopnja poveča za 1, točki v pa izhodna stopnja zmanjša za 1; in če se neki točki stopnja spremeni z d na d', moramo zmanjšati $A[d]$ za 1 in povečati $A[d']$ za 1.

Poleg tabele A vzdržujmo tudi števec R, ki naj pove, pri koliko d-jih je $A[d] > 0$. Tudi tega ni težko vzdrževati, ko se vrednosti v tabeli spreminjajo. Graf ima n točk in možnih je n stopenj (od 0 do $n-1$); če ima graf obliko, ki nas zanima, torej če ima za vsako stopnjo od 0 do $n-1$ natanko eno točko, to potem pomeni, da so vse $A[d]$ enake 1, vrednost R pa bo zato enaka n. In po drugi strani, če je $R = n$, se to lahko (pri n točkah in n stopnjah) zgodi le tako, da je za vsako stopnjo po ena točka, tak graf pa ima iskano obliko. Vidimo torej, da imamo pred seboj veljavno rešitev natanko tedaj, ko je $R = n$. Takrat povečamo števec rešitev, prvih deset rešitev pa si tudi zapomnimo, da jih bomo lahko na koncu izpisali. Oglejmo si še implementacijo tega postopka:

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

int n; vector<string> tabela; // Vhodni podatki.
enum { MaxIzpisanihResitev = 10 };

int stResitev = 0;
stringstream resitve; // Rešitve, ki jih bomo na koncu izpisali.

void IzpisiResitev(int u, int v, const vector<int> &D)
{
  if (stResitev++ >= MaxIzpisanihResitev) return;

  // V tabeli L pripravimo točke, urejene po stopnji.
  vector<int> L(n); for (int w = 0; w < n; ++w) L[D[w]] = w;

  // Izpišimo to rešitev v tok „resitve“.
  resitve << min(u, v) + 1 << ' ' << max(u, v) + 1;
  for (auto w : L) resitve << ' ' << w + 1;
  resitve << endl;
}

int main()
{
  // Preberimo vhodne podatke.
  cin >> n; tabela.resize(n); for (auto &vrstica : tabela) cin >> vrstica;

  // Izračunajmo vhodne stopnje točk.
  vector<int> D(n, 0); // D[u] = vhodnja stopnja točke u
  for (int v = 1; v < n; ++v) for (int u = 0; u < v; ++u)
    ++D[tabela[v - 1][u] == '<' ? v : u];

  // A[d] = število točk z vhodno stopnjo d.
  vector<int> A(n, 0); for (int d : D) ++A[d];

  // R = število neničelnih elementov v tabeli A.
  int R = 0; for (int a : A) if (a > 0) ++R;
  if (R == n) IzpisiResitev(-1, -1, D); // Rešitev brez napak v podatkih.

  // Poskusimo obračati povezave.
  for (int vv = 1; vv < n; ++vv) for (int uu = 0; uu < vv; ++uu)
  {
    int u = uu, v = vv; if (tabela[v - 1][u] == '>') swap(u, v);
    int Rprej = R;
```

```
    // Obrnimo povezavo u → v, da bo postala v → u.
    int du = D[u]; if (--A[du] == 0) --R; if (++A[du + 1] == 1) ++R;
    int dv = D[v]; if (--A[dv] == 0) --R; if (++A[dv - 1] == 1) ++R;
    if (R == n) { ++D[u]; --D[v]; IzpisiResitev(uu, vv, D); --D[u]; ++D[v]; }
    // Povrnimo povezavo v prvotno stanje.
    --A[du + 1]; ++A[du]; --A[dv - 1]; ++A[dv]; R = Rprej;
  };
  // Izpišimo rezultate.
  cout << stResitev << endl << resitve.str(); return 0;
}
```

5. Razrez kolobarja

Na vhodu smo dobili števila $a_1, \ldots, a_n$, toda ker je kolobar cikličen, bo za naš opis rešitve prikladno, če si bomo mislili, da obstajajo tudi $a_0 = a_n$ in $a_t = a_{t \bmod n}$ za vsa naravna števila t.

Gotovo obstaja razrez s ceno kvečjemu $\sum_{i=1}^{n} a_i$ (takšno ceno bi dobili, če bi vsa števila pripadala enemu kosu); in gotovo ne obstaja razrez s ceno 0 (kajti vsa števila na našem kolobarju so večja od 0); med tema dvema mejama pa lahko najmanjšo še dosegljivo ceno poiščemo z bisekcijo. Na vsakem koraku bisekcije moramo torej za neko ceno C preveriti, ali obstaja razrez na k kosov s ceno kvečjemu C.

Recimo, da bi začeli seštevati števila na kolobarju od a_i naprej in šli s tem tako daleč, kolikor je le mogoče, ne da bi vsota presegla C; številu tako uporabljenih seštevancev recimo $f(i)$. Izračunajmo to za vse i, pri čemer imejmo v mislih, da ni treba računati vsakega $f(i)$ posebej z zanko od a_i naprej. Recimo, da smo že izračunali $f(i)$ in nas zdaj zanima $f(i+1)$. Vemo že, da je vsota $a_i + a_{i+1} + \ldots + a_{i+f(i)-1}$ manjša ali enaka C; če zdaj gledamo podobno vsoto, le da se začne šele pri a_{i+1}, bo tudi ona manjša ali enaka C (saj je od prejšnje vsote manjša za a_i, ta pa je pozitiven). Izračun $f(i+1)$ lahko torej začnemo tako, da vsoti, do katere smo prišli pri izračunu $f(i)$, odbijemo prvi člen in potem pogledamo, ali ji smemo na desnem koncu kak člen še dodati. Tako lahko v $O(n)$ časa izračunamo $f(i)$ za vse i skupaj (pri vsakem i smo en člen odšteli iz vsote in na desni morda nekaj členov dodali, toda pri tem dodajanju nikoli ne moremo iti več kot n števil naprej od števila a_i, pri katerem smo začeli — meja C, ki je naša vsota ne sme preseči, bo namreč vedno manjša od vsote vseh števil na kolobarju).

Funkcija f nam torej pove, da če se začne neki kos našega razreza pri številu a_i, bo ta kos pokril največ $f(i)$ števil; najkasneje pri $a_{i+f(i)}$ se mora torej začeti naslednji kos. Če hočemo sestaviti razrez za ceno kvečjemu C, moramo s k kosi pokriti celoten kolobar. Definirajmo torej $f_r(i)$ kot vrednost, ki pove, koliko števil lahko pokrijemo z r kosi, če začnemo pri številu a_i. Pri $r = 1$ je $f_1(i) = f(i)$ za funkcijo f iz prejšnjega odstavka; kaj pa za večje r? Recimo, da je $r = p + q$; potem, če začnemo pri a_i in narežemo r kosov, je učinek enak, kot če bi najprej narezali p kosov in od tam naprej nadaljevali s še q kosi. Vemo pa, da s p kosi, če začnemo pri a_i, pokrijemo $f_p(i)$ števil; preostalih q kosov se torej začne pri številu a_j za $j := (i + f_p(i)) \bmod n$; zato vemo, da teh q kosov pokrije še $f_q(j)$ števil. Tako torej vidimo:

$$f_{p+q}(i) = f_p(i) + f_q\big((i + f_p(i)) \bmod n\big).$$

S to formulo lahko v $O(n)$ časa izračunamo f_{p+q} (za vse i) iz funkcij f_p in f_q. V našem primeru to pomeni, da lahko iz f_1 po vrsti izračunamo f_2, f_4, f_8 in tako naprej za vse potence števila 2, ki so manjše ali enake k; nato pa pogledamo, katere od teh potenc je treba sešteti med sabo, da dobimo k (torej kateri biti so prižgani v dvojiškem zapisu števila k), in iz ustreznih f_{2^r} izračunamo f_k.

Nato moramo le še preveriti, če pri vsaj kakšnem i velja $f_k(i) \geq n$ (torej ali je mogoče začetek razreza izbrati tako, da potem s k kosi pokrijemo celoten kolobar).[3] Če to drži, je razrez mogoč, sicer pa (pri trenutnem C) ni; tako bomo vedeli, ali naj pri bisekciji popravimo zgornjo ali spodnjo mejo.

[3] To lahko preverjamo tudi že prej; jasno je, da če je $r < k$, je mogoče s k kosi (pri istem začetku i) pokriti vsaj toliko števil kot z r kosi; če torej že pri r opazimo, da je $f_r(i) \geq n$, potem lahko takoj zaključimo, da bo tudi $f_k(i) \geq n$ in da torej razrez, kakršnega iščemo, res obstaja.

Ker smo morali izračunati funkcije f_r za $O(\log k)$ različnih r-jev, vsak tak izračun pa je vzel $O(n)$ časa, nam je en korak bisekcije vzel $O(n \log k)$ časa. Če so števila na kolobarju med 1 in A, je bila zgornja meja naše bisekcije na začetku $\sum_{i=1}^{n} a_i \leq nA$, torej se bisekcija konča po $O(\log(nA))$ korakih. Tako ima torej naš postopek časovno zahtevnost $O(n(\log k)(\log nA))$; če gledamo le odvisnost od n-ja in upoštevamo, da je $k < n$, je časovna zahtevnost naprej enaka $O(n(\log n)^2)$.

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

typedef int_fast64_t myint;

int main()
{
  int stTestov; cin >> stTestov;
  while (stTestov-- > 0)
  {
    // Preberimo naslednji kolobar.
    int n, k; cin >> n >> k;
    vector<myint> A(n); myint spodnja = 0, zgornja = 0;
    for (auto &ai : A) { cin >> ai; zgornja += ai; }
    vector<int> f(n), fk(n), ff(n);

    // Z bisekcijo poiščimo najnižjo ceno razreza.
    while (zgornja - spodnja > 1)
    {
      // Gotovo obstaja razrez s ceno kvečjemu „zgornja“
      // in ne obstaja tak s ceno „spodnja“ ali manj.
      myint C = (zgornja + spodnja) / 2;

      // Naj bo f[i] = koliko števil po kolobarju, začenši z A[i],
      // lahko največ seštejemo, ne da bi presegli mejo M.
      int F = 0; myint vsota = 0;
      for (int i = 0; i < n; ++i) {
        if (F > 0) { vsota -= A[i - 1]; --F; }
        while (vsota + A[(i + F) % n] <= C) vsota += A[(i + F++) % n];
        f[i] = F; fk[i] = 0; }

      // Naj bo f_k(i) = koliko števil po kolobarju, začenši z A[i],
      // lahko poberemo, če smemo sestaviti k kosov (pri meji M).
      // Trehutno imamo v f[] funkcijo f_1, v fk[] pa funkcijo f_0.
      bool resljivo = false;
      for (int K = k; K > 0; K >>= 1)
      {
        // Če smo v u-ti iteraciji te zanke, je K = (k >> u), v f[] imamo funkcijo f_{2^u}
        // in v fk[] imamo funkcijo f_{k mod 2^u}. Izračunajmo zdaj v fk[] funkcijo f_{k mod 2^{u+1}}.
        // Če je bit u v k-ju (oz. najnižji bit v K-ju) ugasnjen,
        // je k mod 2^{u+1} = k mod 2^u in fk[] ni treba spreminjati,
        // sicer pa je k mod 2^{u+1} = (k mod 2^u) + 2^u, zato bomo
        // novo fk[] dobili iz stare fk[] in iz f[].
        if (K & 1) for (int i = 0; i < n; ++i) fk[i] = fk[i] + f[(i + fk[i]) % n];

        // Zdaj v ff[] izračunajmo funkcijo f_{2^{u+1}}.
        for (int i = 0; i < n; ++i) ff[i] = f[i] + f[(i + f[i]) % n];
        swap(f, ff);

        // Če je mogoče izbrati začetek i tako, da je f_k(i) >= n,
        // to pomeni, da obstaja razrez s ceno kvečjemu M.
        int maxF = 0; for (int F : fk) if (F > maxF) maxF = F;
        if (K > 1) for (int F: f) if (F > maxF) maxF = F;
        if (maxF >= n) { resljivo = true; break; }
      }
      (resljivo ? zgornja : spodnja) = C;
    }
    cout << zgornja << endl; // Izpišimo rezultat.
```

```
    }
    return 0;
}
```

Ta rešitev je čisto dovolj dobra za testne primere na našem tekmovanju, kot zanimivost pa si vseeno oglejmo, kako jo lahko še izboljšamo. Videli bomo, da lahko (pri danem C) funkcijo $f_k(i)$ izračunamo za vse i že v $O(n)$ časa, tako da bo imela rešitev kot celota časovno zahtevnost $O(n \log n)$ namesto $O(n(\log n)^2)$.

Mislimo si graf s točkami $0, \ldots, n-1$, torej po eno točko za vsako število našega kolobarja; iz vsake točke i naj kaže natanko ena izhodna povezava, namreč na $(i + f(i)) \bmod n$, dolžina te povezave pa naj bo $f(i)$. Povezava $i \to j$ torej pove, da če se en kos začne z i-tim številom na kolobarju, se mora naslednji začeti z j-tim.

Ker gre iz vsake točke natanko ena izhodna povezava, ima naš graf naslednjo strukturo: sestavljen je iz ene ali več šibko povezanih komponent, vsaka taka komponenta pa vsebuje en cikel (v splošnem lahko tudi zanko, torej cikel, v katerem neka točka kaže samo nase; vendar pa v našem primeru do zanke ne bo prišlo, saj bi to pomenilo, da je $f(i) = n$, torej je mogoče z enim kosom pokriti celoten kolobar; s tako velikimi C-ji, pri katerih bi se to lahko zgodilo, pa se pri naši bisekciji ne bomo ukvarjali), na katerega je lahko pripetih še 0 ali več dreves, v katerih vse povezave kažejo k ciklu.[4]

Poleg tega pa, ker ima vsaka točka natanko eno izhodno povezavo, je možna iz vsake točke natanko ena pot določene dolžine. Če začnemo v i in naredimo k korakov, bo skupna dolžina tako prehojenih povezav ravno $f_k(i)$; to je tisto, kar bi radi izračunali za vse i. Takšna pot dolžine k se najprej nekaj časa (0 ali več korakov) vzpenja po drevesu, nato pa se bodisi konča bodisi (če je dosegla cikel in še ni naredila k korakov) še nekaj časa teče po ciklu.

Recimo, da gledamo neki cikel iz m točk; izberimo si poljubno od njih kot začetek cikla in jih od nje naprej oštevilčimo kot $u_0, u_1, \ldots, u_{m-1}$. Ker se cikel po m korakih vrne nazaj v začetno točko, to pomeni, da če bi začeli na kolobarju pri u_0-tem številu in narezali m kosov, bi se zadnji od njih končal tik pred tem številom; tako smo torej naredili enega ali morda celo več celih obhodov po kolobarju. Če je $m \le k$, smo tako pokrili cel kolobar z manj kot k kosi (morda celo večkrat) in lahko takoj zaključimo, da je razrez pri trenutnem C mogoč. V nadaljevanju smemo torej predpostaviti, da je $m > k$.

Definirajmo $s_0 = 0$ in $s_{t+1} = s_t + f(u_t)$; tako je torej s_t dolžina poti, ki se začne v u_0 in naredi t korakov; s_m je dolžina celotnega cikla. S pomočjo teh vsot lahko hitro izračunamo tudi dolžino poljubne druge poti na ciklu; na primer, pot od u_r do u_t je dolga $s_t - s_r$, če je $r < t$, oz. $s_t + s_n - s_r$, če je $r > t$. Tako ne bo težko izračunati f_k za točke na ciklu: $f_k(u_t)$ je preprosto dolžina poti od u_t do $u_{(t+k) \bmod m}$ (spomnimo se, da je $m > k$, zato taka pot naredi manj kot en cel obhod po ciklu).

Kaj pa f_k za ostale točke grafa? Vsako u_t si lahko predstavljamo kot koren drevesa, v katerem vsaka povezava kaže od otroka na starša in kjer z vzpenjanjem po teh povezavah vedno sčasoma dosežemo točko u_t. Preglejmo to drevo z iskanjem v globino. Med iskanjem v globino vzdržujmo seznam oz. sklad točk na poti od korena (torej u_t) do trenutne točke (z drugimi besedami, do so trenutna točka in vsi njeni predniki), pri vsaki od teh točk pa hranimo tudi dolžino poti od nje do korena. Točke na tem seznamu označimo z $v_0, v_1, \ldots, v_h$, pri čemer je $v_0 = u_t$ koren drevesa, v_h pa je točka (na globini h), s katero se trenutno ukvarjamo pri iskanju v globino. Dolžino poti od v_r do v_0 označimo z D_r; tega ni težko računati, ko se iz neke točke v_{r-1} spustimo v njenega otroka v_r: dolžino D_r dobimo tako, da D_{r-1} prištejemo dolžino povezave $v_r \to v_{r-1}$, to pa je $f(v_r)$.

Recimo torej, da smo trenutno pri iskanju v globino dosegli točko v_h, ki leži v drevesu na globini h, torej h korakov daleč od korena. Če je $h \ge k$, je k korakov dolga pot iz v_h sestavljena le iz vzpenjanja po drevesu in se konča v točki v_{h-k}; tedaj je torej $f_k(v_h) = D_h - D_{h-1}$. Če pa je $h < k$, bomo iz v_h po h korakih dosegli cikel (točko $v_0 = u_t$; dolžina poti do sem je D_h) in morali nato narediti še $h - k$ korakov po ciklu, torej od u_t do $u_{(t+h-k) \bmod m}$ (dolžino takšne poti po ciklu pa smo se naučili računati že malo prej).

[4] S takšnimi grafi smo se na naših tekmovanjih že srečali; gl. nalogo 2018.3.5 (*Bilten* 2018, str. 79) in nalogo J s CERC 2023 (*Bilten* 2023, str. 179).

Vidimo torej, da smo se s ciklom dolžine m ukvarjali $O(m)$ časa in da smo nato za vsako točko v drevesih, pripetih na ta cikel, porabili še $O(1)$ časa; tako lahko izračunamo $f_k(i)$ za vse točke i našega grafa (in s tem za vsa števila na našem kolobarju) v $O(n)$ časa.

Naloge so sestavili: omrežnina — Tomaž Hočevar; steklenice, zlatarna — Vid Kocijan; natakarica, ultrazvok — Ella Potisek; prijave na izlet, trojice, chordpro — Jakob Schrader; uravnotežena prehrana, razpolavljanje torte — Jure Slak; beg, trki — Jošt Smrtnik; tovarna — Patrik Žnidaršič; razrez kolobarja — Jakob Žorž; urejanje z lažmi — Janez Brank.

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