19. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2024

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.");
  }
}
```

19. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2024

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 (Visual Studio Code, Geany, Lazarus) in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

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

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

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite; bolj učinkovite rešitve dobijo več točk (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

Rešitve in rezultati bodo objavljeni na `https://rtk.ijs.si/`.

Vabimo te, da ob koncu tekmovanja izpolniš tudi **anketo**:
`https://www.1ka.si/a/4298cad5`

1. Budilka

Napiši tri **podprograme** (funkcije), ki jih bo sistem klical, da mu bodo pomagali upravljati z budilko. Tvoji podprogrami naj bodo takšne oblike:

- **void** NastaviBudilko(**int** t) — sistem ga pokliče, ko želi uporabnik nastaviti, naj zvoni vsak dan ob času t. Če je bila budilka pred tem nastavljena na zvonjenje ob drugem času, naj se stari čas pobriše. Čas je podan kot celo število v minutah od polnoči (torej od 0 do $24 \cdot 60 - 1 = 1439$).
- **bool** JeCasZaAlarm() — sistem ga pokliče vsako minuto točno ob začetku minute. Če naj budilka to minuto zvoni, naj ta funkcija vrne **true**, sicer **false**.
- **void** Dremez() — sistem ga pokliče, ko pritisne uporabnik tipko za dremež. Če se to zgodi manj kot pet minut po originalnem zvonjenju, naj tvoj program poskrbi, da bo budilka še enkrat zazvonila pet minut po originalnem zvonjenju. Morebitne nadaljnje klice tega podprograma ignoriraj (prav tako tudi klice, ki nastopijo več kot pet minut po originalnem zvonjenju) — z drugimi besedami, poskrbeti moraš, da bo učinek takih klicev enak, kot če se podprogram takrat sploh ne bi klical. Nova nastavitev časa bujenja (s podprogramom NastaviBudilko) naj hkrati tudi prekliče trenutno aktivno funkcijo dremeža.

Poleg teh podprogramov lahko tvoja rešitev uporablja tudi globalne spremenljivke, ki jim lahko tudi določiš začetne vrednosti.

Predpostavi, da se bo tvoj program začel izvajati točno opolnoči in da bo takrat sistem najprej poklical tvoj podprogram JeCasZaAlarm. Ob začetku izvajanja naj se program obnaša tako, kot da budilka ni nastavljena.

Še deklaracije v pythonu:

```
def NastaviBudilko(t: int) -> None: ...
def JeCasZaAlarm() -> bool: ...
def Dremez() -> None: ...
```

2. Odbojkaške točke

Tvoj prijatelj Jan z obale v svojem prostem času sodi na odbojkaških tekmah. Seveda pa bi on raje igral odbojko in na plaži ribaril, zato bi rad razmišljajoči del sojenja preložil nate. On bo zapisal zaporedje dogodkov, ti pa mu izračunaj, ali to točko dobi ekipa 1 ali 2.

Dogodki, ki jih bo Jan napisal, so treh vrst:

1. `O1` oz. `O2` pomeni **o**dboj ekipe 1 oz. 2. Na začetku vsake točke mora nekdo iz ene ekipe žogo servirati (ima en odboj) na drugo stran mreže. Od takrat naprej mora vsaka ekipa žogo odbiti vsaj enkrat in največ trikrat ter jo z zadnjim odbojem spraviti nazaj čez mrežo.
2. `P1` oz. `P2` pomeni, da je žoga padla v **p**olje ekipe 1 oz. 2. Če ekipi žoga pade v lastno polje, točko izgubi, če pa jo spravi v polje druge ekipe, točko dobi.
3. `I` pomeni, da je žoga šla **i**zven igrišča (v out) ali neveljavno letela pod mrežo. Ekipa, ki se ji to zgodi, izgubi točko.

Napiši program (ali podprogram oz. funkcijo), ki iz seznama dogodkov izračuna, katera ekipa dobi prvo točko. Podrobnosti tega, v kakšni obliki je seznam dogodkov podan, določi sam in jih v svoji rešitvi tudi opiši.

Zaporedje se vedno začne z dogodkom `O1` ali `O2`, kar nam pove, katera ekipa je servirala žogo. Zaporedje se ne konča nujno takoj po tistem, ko ena od ekip dobi točko.

3. Tekstonim

Na telefonski številčni tipkovnici imamo na vsaki tipki poleg številke zapisane še tri ali štiri črke, kar nam včasih olajša zapomniti si kakšno telefonsko številko.

Črkam ustrezajo števke takole:

`a`, `b`, `c`	$= 2$
`d`, `e`, `f`	$= 3$
`g`, `h`, `i`	$= 4$
`j`, `k`, `l`	$= 5$
`m`, `n`, `o`	$= 6$
`p`, `q`, `r`, `s`	$= 7$
`t`, `u`, `v`	$= 8$
`w`, `x`, `y`, `z`	$= 9$

(števki 1 in 0 nimata pripadajočih črk)

Za neko podano telefonsko številko (ki bi si jo radi lažje zapomnili) poišči med možnimi besedami iz podanega seznama take, ki (predelane v številke) predstavljajo strnjen podniz podane telefonske številke. Za vsak tak podniz izpiši niz, ki nastane, če v telefonski številki zamenjaš ta podniz z besedo iz seznama.

Primer: če imamo telefonsko številko 0586326 in seznam, v katerem so tudi besede `june`, `junec`, `kunec`, `nebo` in `voda`, bomo našli naslednje možne zapise:

```
0june26
0junec6
0kunec6
058nebo
05voda6
```

Napiši program, ki bo z vhodne datoteke (ali standardnega vhoda) prebral eno telefonsko številko, potem pa pregledal vse besede v preostalih vrsticah vhodne datoteke in izpisal najdene zapise.

Če se beseda (predelana v številko) pojavi kot podniz večkrat, izpiši vse tako dobljene zapise.

V prvi vrstici vhodne datoteke je zapisana telefonska številka (zaporedje števk med 0 in 9), v preostalih vrsticah vhoda so besede slovarja, vsaka v svoji vrstici, sestavljajo jih le male črke angleške abecede.

Primer vhoda:

```
0586326
june
junec
zmagovalec
kunec
nebo
vedro
voda
```

Pripadajoči izhod:

```
0june26
0junec6
0kunec6
058nebo
05voda6
```

4. Premešani mozaik

Umetnik Polde je sestavil mozaik v obliki pravokotne kariraste mreže, široke w stolpcev in visoke h vrstic; v vsaki celici mreže je kvadratna ploščica v eni od B možnih barv, zato lahko barve predstavimo kar s števili od 1 do B. Na Poldetovem mozaiku je bila j-ta ploščica v i-ti vrstici barve b_{ij}.

Žal so v noči pred otvoritvijo Poldetove razstave nepridipravi vdrli v galerijo in premešali ploščice njegovega mozaika; zdaj je j-ta ploščica v i-ti vrstici barve a_{ij}. Poudarimo, da gre še vedno za iste ploščice, le drugače so razporejene po pravokotni mreži (torej se število ploščic posamezne barve ni spremenilo).

Polde se je odločil, da bo povrnil mozaik v prvotno stanje, vendar bo iz tega naredil performans. Mozaik bo spreminjal po korakih, pri čemer v vsakem koraku zamenja med seboj dve sosednji ploščici (to sta taki, ki imata skupno eno od stranic).

Opiši postopek (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki kot vhodne podatke dobi trenutno stanje mreže (vse a_{ij}) in njeno prvotno stanje (vse b_{ij}) ter izpiše poljubno zaporedje takšnih zamenjav, s katerim lahko Polde povrne mozaik v prvotno stanje. Podrobnosti izpisa niso pomembne, da bo le razvidno, kateri dve ploščici naj na posameznem koraku zamenja.

5. Rokomet

Spremljamo rokometno tekmo med dvema moštvoma, A in B. Ker je moštvo A precej boljše od B, o rezultatih poročajo na nenavaden način. Namesto da bi poročali o vsakem golu posebej, poročajo samo o golih moštva B, hkrati pa povejo, koliko golov je vmes doseglo moštvo A.

Primer poročanja: „Ekipa B je končno dosegla gol, vmes je ekipa A dala 5 golov. Ponovno gol za B, vmes 2 gola za A. Od zadnjega gola za B do konca tekme je padlo še 7 golov za A.“

Napiši program ali podprogram oz. funkcijo, ki izpiše končni rezultat tekme in še to, kolikokrat se je vmes zamenjalo vodstvo.

Predpostaviš lahko, da so podatki podani kot seznam števil, npr. $[5, 2, 7]$ (kar ustreza gornjemu primeru). Vsa števila razen zadnjega predstavljajo en gol moštva B, velikost števila pa pove, koliko golov je vmes dalo moštvo A. Zadnje število predstavlja število golov od zadnjega gola B do konca tekme.

Za primer $[5, 2, 7]$ so posamezni rezultati (A : B) naslednji: $0:0$, $1:0$, $2:0, \ldots, 5:0$, $5:1$, $6:1$, $7:1$, $7:2$, $8:2, \ldots, 13:2$, $14:2$. Končni rezultat je 14 proti 2, vodstvo pa se ni nikoli zamenjalo.

Še en primer: pri seznamu $[2, 0, 0, 0, 1, 3]$ se je rezultat spreminjal takole: $0:0$, $1:0$, $2:0$, $2:1$, $2:2$, $2:3$, $2:4$, $3:4$, $3:5$, $4:5$, $5:5$, $6:5$. Končni rezultat je 6 proti 5, vodstvo pa se je zamenjalo dvakrat: najprej je nekaj časa vodilo moštvo A, nato nekaj časa B in nato spet A.

Opomba: če nekaj časa vodi eno moštvo, pa potem drugo izenači in nato prvo spet povede, se to ne šteje za spremembo vodstva.

19. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2024

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-2024-2/` 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/4298cad5`

1. Disleksija

Podan je seznam n števil $x_1, x_2, \ldots, x_n$, ki bi moral biti urejen naraščajoče ($x_i \le x_{i+1}$). Ker je imel avtor seznama manjše težave s števkama 6 in 9, temu ni nujno tako. Zaradi nezanesljivosti teh dveh števk smo jih povsod v seznamu števil zamenjali z zvezdico. Posamezna števila so lahko zelo dolga (m-mestna). Nobeno od števil se ne začne na ničlo.

Napiši čim bolj učinkovit **program** (ali podprogram oz. funkcijo), ki bo zvezdice v seznamu zamenjal s števkama 6 ali 9 tako, da bo nastal naraščajoč seznam, ali javil, da to ni mogoče. Podrobnosti tega, v kakšni obliki tvoj (pod)program dobi vhodne podatke, si izberi sam in jih v svoji rešitvi tudi opiši. **Utemelji** pravilnost in učinkovitost svojega postopka — zakaj najde pravilen odgovor in koliko operacij potrebuje v odvisnosti od parametrov n in m.

Če naloge ne znaš rešiti v splošnem, jo lahko za 50 % točk rešiš z dodatno predpostavko, da je v vsakem številu največ ena zvezdica.

Primer. Recimo, da vhodni seznam sestavlja naslednjih pet nizov:

```
7      ***      88*      23*5*3      23*551
```

Potem je eden od možnih izhodnih seznamov (ni pa edini) tale:

```
7      696      889      236563      239551
```

2. Preurejanje

Podan imamo seznam n števil $x_1, x_2, \ldots, x_n$, ki bi ga radi uredili od manjših proti večjim. Predpostavite lahko, da bo n potenca števila 2 in $n \ge 4$. Poznamo vsebino seznama, vendar elementov ne moremo poljubno prestavljati. Edina operacija, s katero lahko spreminjamo seznam, je sledeča: izberemo neki podseznam (lahko nestrnjen) dolžine $n/2$ in ta podseznam na mestu uredimo naraščajoče. Povedano bolj natančno, izberemo indekse $1 \le i_1 < i_2 < \ldots < i_{n/2} \le n$ in z eno operacijo preuredimo elemente na teh indeksih tako, da bodo urejeni naraščajoče. Na primer, iz seznama $[7, \mathbf{2}, \mathbf{8}, 3, 1, \mathbf{9}, 2, \mathbf{4}]$ bi ob izbiri indeksov $[2, 3, 6, 8]$ dobili seznam $[7, \mathbf{2}, \mathbf{4}, 3, 1, \mathbf{8}, 2, \mathbf{9}]$ (v krepkem tisku so napisani tisti elementi, ki so sodelovali v tej operaciji). **Opiši postopek**, ki bo s čim manj takimi operacijami uredil seznam. Svoj odgovor dobro **utemelji** — zakaj tvoj postopek uspešno uredi seznam in koliko operacij potrebuje v najslabšem primeru.

3. Boggle

Pri igri Boggle dobimo mrežo $n \times n$ črk, na kateri nato iščemo besede. Besede zaporedoma sestavljamo iz črk, ki mejijo druga na drugo (levo/desno, gor/dol in po diagonalah). Smer, v kateri sestavljamo besedo, lahko poljubnokrat spremenimo. Posamezno črko mreže lahko pri posamezni besedi uporabimo največ enkrat, lahko pa isto črko uporabimo spet pri kakšni drugi besedi.

Napiši program ali podprogram oz. funkcijo, ki kot vhodne podatke dobi mrežo črk in seznam k besed dolžine največ 8 znakov ter izpiše, koliko izmed danih besed lahko najdemo na dani mreži. Podrobnosti tega, v kakšni obliki dobi vhodne podatke, določi sam in jih v svoji rešitvi tudi opiši. Predpostaviš lahko, da v mreži in besedah nastopajo le male črke angleške abecede.

Primer: če dobimo naslednjo mrežo 3×3:

```
ekz
ato
lva
```

in če dobimo seznam besed `avto`, `voz`, `teta` in `zlo`, lahko najdemo v mreži prvi dve, drugih dveh pa ne.

4. Prepisovanje

Pri dodatnem pouku računalništva je učiteljica vsakemu učencu oz. učenki dodelila svoj računski problem, za katerega so morali poiskati čim bolj učinkovit algoritem. Janko rešuje problem osvetlitve cestnega omrežja, kjer je podanih C dvosmernih cest, ki se med seboj stikajo v K križiščih. Za vsako cesto c_i je podano, kateri dve križišči u_i in v_i povezuje med seboj. Na nekatera križišča želimo postaviti luči tako, da bodo osvetljene vse ceste. Posamezna cesta je osvetljena, če se nahaja luč na vsaj enem izmed obeh križišč, ki ju povezuje. Janko išče algoritem, ki bo izračunal najmanjše število postavljenih luči, s katerimi bomo osvetlili vse ceste.

Poleg Janka sedi Metka, ki se ukvarja s problemom zbiranja sličic, v katerem obstaja S različnih sličic, ki so naprodaj v P različnih paketih, ki vsebujejo neko podmnožico sličic. Za vsak paket je znano, katere sličice so v njem. Metka išče algoritem, ki bo izračunal, kakšno je najmanjše število paketov, s katerimi bi lahko dobila vseh S sličic (pri tem se lahko kakšne sličice tudi ponovijo).

Metki je uspelo rešiti svoj problem v času $t(S, P)$ — z drugimi besedami, za rešitev problema s S različnimi sličicami in P paketi potrebuje njen algoritem $t(S, P)$ operacij. V kakšnem času lahko reši svoj problem Janko, če si pri reševanju lahko pomaga z rešitvijo Metke, ki sedi poleg njega? Dobro utemelji svoj odgovor — kako in zakaj si lahko Janko pomaga z Metkino rešitvijo ter kako učinkovito lahko s tem reši svoj problem.

5. Laserji

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 2 sekundi in sme porabiti največ 256 MB pomnilnika.

Naloga je razdeljena na štiri podnaloge, ki se razlikujejo po omejitvah vhodnih podatkov. Točke pri posamezni podnalogi dobiš, če tvoj program pravilno reši vse testne primere pri tisti podnalogi. 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.

Veliko napihnjenih balonov imamo razporejenih na različna mesta na karirasti mreži. Z laserjem lahko ustrelimo navpično ali vodoravno in tako popokamo vse balone na poti laserja.

Naredili bomo nekaj strelov, po vsakem strelu pa nas zanima, koliko balonov je še celih. **Napiši program**, ki to ugotovi.

Vhodni podatki. V prvi vrstici se nahajata dve s presledkom ločeni števili, N (število balonov) in S (število strelov z laserjem). Sledi N vrstic; i-ta od njih vsebuje celi števili x_i in y_i (koordinati i-tega balona). Sledi še S vrstic; j-ta od njih vsebuje celo število s_j in črko „v“ ali „h“. Črka „v“ pomeni, da smo z laserjem ustrelili navpično (po premici $x = s_j$), črka „h“ pa, da smo ustrelili vodoravno (po premici $y = s_j$).

Omejitve vhodnih podatkov: $1 \leq N \leq 10^5$; $1 \leq S \leq 10^5$; vse x_i, y_i in s_j so po absolutni vrednosti manjše ali enake 10^9. Nobena dva balona ne bosta na istih koordinatah.

Pri nekaterih podnalogah veljajo še dodatne omejitve.

- Podnaloga 1 (5 točk): $N \leq 100$; $S \leq 100$; $0 \leq x_i \leq 100$; $y_i = 0$; $-1000 \leq s_j \leq 1000$.
- Podnaloga 2 (5 točk): $N \leq 500$; $S \leq 500$.
- Podnaloga 3 (5 točk): $0 \leq x_i \leq 1000$; $0 \leq y_i \leq 1000$.
- Podnaloga 4 (5 točk): brez dodatnih omejitev.

Izhodni podatki: po vsakem izmed S strelov v novo vrstico izpiši število balonov, ki so še celi.

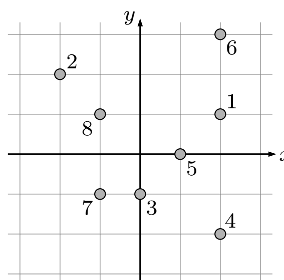
Primer vhoda:

```
8 7
2 1
-2 2
0 -1
2 -2
1 0
2 3
-1 -1
-1 1
4 h
2 h
2 v
-1 h
2 v
1 v
-7 h
```

Pripadajoči izhod:

```
8
7
4
2
2
1
1
```

Naslednja slika prikazuje balone pri tem primeru:



19. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2024

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, mi pa jih bomo preverili s prevajalniki FreePascal 3.0.4, GNUjevima `gcc` in `g++` 10.3.0 (ta verzija podpira C++17, novejše različice standarda C++ pa le delno), prevajalnikom za javo iz JDK 17, s prevajalnikom Mono 6.8 za C#, s prevajalnikom `rustc` 1.65 za rust in z interpreterjem za python 3.8.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2024-3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Pred začetkom tekmovanja lahko poskusiš oddati katero od nalog iz arhiva `https://putka-rtk.acm.si/tasks/s/test-sistema/list/`. Uporabniško ime in geslo za Putko sta enaki kot za računalnike. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava 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 podrobne nastavitve prevajalnikov na ocenjevalnem strežniku glej `https://putka-rtk.acm.si/info/`. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku. 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/4298cad5`

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 25 in vsak je vreden po 4 točke, pri drugi nalogi jih je 10 in vsak je vreden po 10 točk, pri četrti jih je 20 in vsak je vreden po 5 točk; pri posameznem testnem primeru dobi program vse točke, če je izpisal pravilen odgovor, sicer pa 0 točk (izjema je druga naloga, kjer so možne tudi delne točke pri posameznem testnem primeru). Tretja 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-ju:

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

- V C++:

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

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

- V pythonu:

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

- V javi:

```
import java.io.*;
import java.util.Scanner;
public class Poskus
{
  public static void main(String[] args)
                           throws IOException
  {
    Scanner fi = new Scanner(System.in);
    int i = fi.nextInt(); int j = fi.nextInt();
    System.out.println(10 * (i + j));
  }
}
```

- V C#:

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

19. tekmovanje ACM v znanju računalništva za srednješolce

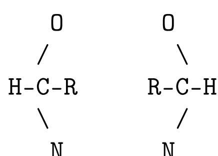
23. marca 2024

NALOGE ZA TRETJO SKUPINO

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

1. Kiralnost

Nekatere molekule imajo zanimivo lastnost, in sicer da sta si molekula in njena zrcalna slika različni (posledično imata drugačne kemijske lastnosti ipd.). Natančneje: zrcalne slike ne moremo spremeniti v originalno sliko z nobenim zaporedjem premikov ali vrtenj. Primer:



```
   O         O
  /           \
H-C-R       R-C-H
  \           /
   N         N
```

Na gornji sliki je prikazana ena taka molekula v obeh različicah. Tej lastnosti rečemo *kiralnost* („ročnost“) molekul, tj. leva molekula je levo-ročna, desna pa desno-ročna (ima desno kiralnost). **Napiši program**, ki ugotovi, koliko je na podani sliki levih oz. desnih molekul. Vse molekule so točno take kot na gornjem primeru (z istimi petimi atomi v enakem razporedu itd.), le da je lahko posamezna molekula tudi zasukana za 90, 180 ali 270 stopinj. Molekule na sliki se ne prekrivajo (natančneje povedano: noben znak na sliki ne pripada več kot eni molekuli), lahko pa se dotikajo. Vsaka molekula je vidna v celoti (torej ne štrlijo čez rob slike) in na sliki ni ničesar drugega razen teh molekul.

Vhodni podatki: v prvi vrstici sta dve s presledkom ločeni naravni števili, w (širina slike) in h (višina slike). Sledi h vrstic, ki podajajo vsebino slike; v vsaki od njih je w znakov. Možni znaki so: `.` (pike, ki predstavljajo prazne dele slike), `/`, `\`, `|`, `-` in črke `H`, `C`, `R`, `O` in `N`.

Omejitve: $1 \leq w \cdot h \leq 10^6$.

Izhodni podatki: izpiši dve števili, ločeni s presledkom, najprej število levih in nato število desnih molekul.

Primer vhoda:

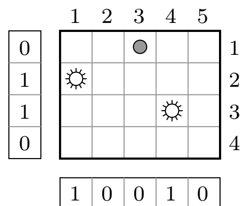
```
19 7
...N...............
....\........H..H..
..R-C-HH.....|..|..
..../..|..../C\/C\.
...O../C\..N.|NO|.O
.....O.|.N...R..R..
.......R...........
```

Pripadajoči izhod:

```
3 1
```

2. Lučke

Iz n božičnih lučk smo sestavili napis na steni. Lučke so postavljene na mreži velikosti $w \times h$, niso pa zapolnjeni vsi položaji na tej mreži. Krmilnik nam omogoča, da prižgemo oz. ugasnemo vsako lučko posebej, kar bomo izrabili, da napis animiramo, pred tem pa moramo vedeti, kje je posamična lučka nameščena — lučke so namreč oštevilčene tako, kot so nanizane na žico. Da ugotovimo, kje so lučke postavljene, smo kupili dva svetlobna senzorja. Enega smo postavili pod napis, enega pa levo od napisa. Z njima lahko zaznavamo, v katerih stolpcih oz. vrsticah je prisotna kakšna prižgana lučka.



Primer mreže velikosti 5×4 s tremi lučkami, od katerih sta dve prižgani, ena pa ugasnjena. Nad mrežo in desno od nje so številke stolpcev oz. vrstic. Ničle in enice levo in spodaj kažejo, kaj zaznavata svetlobna senzorja.

Napiši program, ki določi položaj vseh lučk na mreži, pri čemer mora uporabiti čim manj poizvedb. Vsaka poizvedba je sestavljena iz binarnega niza dolžine n; i-ti znak v nizu pove, ali naj se i-ta lučka prižge, preden se aktivirata senzorja svetlobe. Odgovor na poizvedbo sta dva binarna niza. Prvi je dolžine w in pove, v katerih stolpcih mreže je spodnji senzor zaznal svetlobo, drugi pa je dolžine h in pove, v katerih vrsticah mreže je levi senzor zaznal svetlobo. (Z drugimi besedami, v prvem nizu je enica na x-tem mestu natanko v primeru, ko je v x-tem stolpcu kakšna prižgana lučka, sicer pa je tam ničla. Podobno je v drugem nizu enica na y-tem mestu natanko v primeru, če je v y-ti vrstici kakšna prižgana lučka, sicer pa je tam ničla.) Zagotovljeno je, da nobeni dve lučki nista postavljeni v istem stolpcu ali isti vrstici.

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 tri s presledki ločena cela števila: w (širina mreže), h (višina mreže) in n (število lučk).

2. Nato naredi eno ali več poizvedb. Za vsako poizvedbo izpiši na standardni izhod eno vrstico, ki se začne z besedo `POIZVEDBA`, tej pa sledi presledek in niz poizvedbe, kot je opisan zgoraj (n znakov, samih ničel in enic); nato pa s standardnega vhoda preberi dve vrstici, ki vsebujeta odgovor na tvojo poizvedbo (v prvi od teh dveh vrstic je w znakov dolg niz samih ničel in enic, v drugi pa h znakov dolg niz samih ničel in enic).

3. Na koncu izpiši rezultate in prenehaj z izvajanjem. Za izpis rezultatov naj program najprej izpiše vrstico z besedo `REZULTATI`, nato pa naj izpiše še n vrstic, pri čemer naj i-ta od teh vrstic vsebuje zaporedno številko stolpca in vrstice (ločeni s presledkom), kjer se nahaja i-ta lučka.

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/`).

Tvoj program sme izvesti največ n 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).

(*Nadaljevanje na naslednji strani.*)

Omejitve: $1 \leq n \leq 50$ in $1 \leq w, h \leq 10^4$.

Točkovanje: pri tej nalogi je deset testnih primerov. Število točk, ki jih dobi tvoj program pri posameznem testnem primeru, je odvisno od tega, koliko poizvedb izvede. Če je m najmanjše možno število poizvedb pri tem testnem primeru, dobi tvoj program toliko točk:

- 10 točk, če izvede natanko m poizvedb;
- 6 točk, če izvede več kot m in kvečjemu $2m - 1$ poizvedb;
- 3 točke, če izvede več kot $2m - 1$ in kvečjemu n poizvedb;
- 0 točk sicer.

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.

Primer:

Tvoj program izpiše:	Sistem izpiše:
	`5 4 3`
`POIZVEDBA 100`	
	`00100`
	`1000`
`POIZVEDBA 011`	
	`10010`
	`0110`
`POIZVEDBA 001`	
	`00010`
	`0010`
`REZULTATI`	
`3 1`	
`1 2`	
`4 3`	

3. Matrika

Dana je matrika (za potrebe naše naloge to ni nič drugega kot pravokotna tabela, ki lahko v svojih celicah vsebuje števila) z n vrsticami in m stolpci. Položaj celice v njej označimo s parom (i, j), kjer je i številka vrstice, j pa številka stolpca. Vrednost v celici (i, j) označimo z $a_{i,j}$.

V nekatere celice (i, j) naše matrike so že vpisana paroma različna števila $a_{i,j}$ z območja $1 \leq a_{i,j} \leq nm$. Take celice imenujmo *fiksne*, vrednosti v njih pa *fiksne vrednosti*. Ostale celice matrike so trenutno prazne. Zanima nas, ali lahko te preostale celice zapolnimo s števili od 1 do nm tako, da:

- se vsako število v matriki pojavi natanko enkrat in
- so vsa števila, ki se nahajajo v celicah zgoraj in levo od fiksne celice, manjša ali enaka od števila v tisti fiksni celici. Z drugimi besedami: če je (i, j) fiksna celica in če je $1 \leq k \leq i$ in $1 \leq \ell \leq j$, potem mora veljati $a_{k,l} \leq a_{i,j}$.

Dva primera rešljive in nerešljive matrike (vrednosti fiksnih celic so zapisane z rdečo):

2	1	6	8	11	13
3	**4**	9	10	**15**	18
5	7	**12**	14	16	17

Rešljiva matrika

			9		
	6				

Nerešljiva matrika

Napiši program, ki prebere podatke o matriki in fiksnih vrednostih v njej ter ugotovi, ali je mogoče matriko zapolniti v skladu s temi pravili ali ne.

Vhodni podatki. V prvi vrstici so tri cela števila, ločena s presledki: n (višina matrike), m (širina matrike) in k (število fiksnih celic). Sledi k vrstic, ki opisujejo fiksne celice; vsaka od teh vrstic vsebuje po tri cela števila, ločena s presledki: i (številka vrstice, v kateri leži ta fiksna celica), j (številka stolpca, v katerem leži ta fiksna celica) in $a_{i,j}$ (vrednost v tej fiksni celici).

Omejitve: $1 \leq n \leq 10^6$; $1 \leq m \leq 10^6$; $1 \leq k \leq 10^5$; za vsako fiksno celico velja $1 \leq i \leq n$, $1 \leq j \leq m$ in $1 \leq a_{i,j} \leq n \cdot m$. Vrednosti v fiksnih celicah so si paroma različne (nobeni dve fiksni celici nimata enake vrednosti).

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

- Podnaloga 1 (10 točk): $n \leq 4$ in $m \leq 3$.
- Podnaloga 2 (20 točk): $n \leq 1000$, $m \leq 1000$ in $k \leq 10^4$.
- Podnaloga 3 (40 točk): $k \leq 10^4$.
- Podnaloga 4 (30 točk): brez dodatnih omejitev.

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

Izhodni podatki: izpiši „`DA`" ali „`NE`" (brez narekovajev), odvisno od tega, ali je matriko mogoče zapolniti v skladu z zahtevami naloge ali ne.

Primer vhoda:

```
3 6 3
3 3 12
2 2 4
2 5 15
```

Pripadajoči izhod:

```
DA
```

Še en primer vhoda:

```
3 6 2
2 4 9
3 2 6
```

Pripadajoči izhod:

```
NE
```

Komentar: to sta ista primera kot v besedilu naloge zgoraj.

4. Vodoravne daljice

Pri tej nalogi se bomo ukvarjali z vodoravnimi daljicami. Vodoravno daljico z levim krajiščem (x_L, y) in desnim krajiščem (x_D, y) bomo krajše pisali kot (x_L, y, x_D).

Na daljicah smemo izvajati dve operaciji:

- Daljico (x_L, y, x_D) lahko pri poljubnem x_M z območja $x_L < x_M < x_D$ razrežemo na dve daljici, (x_L, y, x_M) in (x_M, y, x_D). Cena te operacije je $x_D - x_L$.
- Daljico (x_L, y, x_D) lahko premaknemo gor ali dol v navpični smeri, tako da iz nje nastane (x_L, y', x_D). Cena te operacije je $|y - y'|$.

Dana so števila $x_0, \ldots, x_n$ in $y_1, \ldots, y_n$, pri čemer je $x_0 < x_1 < \ldots < x_n$. Na začetku imamo eno samo daljico, $(x_0, 0, x_n)$. Naš cilj je na koncu imeti n daljic, in sicer (x_{i-1}, y_i, x_i) za $i = 1, \ldots, n$. **Napiši program**, ki izračuna skupno ceno najcenejšega zaporedja operacij, ki nas pripelje v to želeno končno stanje.

Vhodni podatki: v prvi vrstici je naravno število n. V drugi vrstici so cela števila $x_0, x_1, \ldots, x_{n-1}, x_n$, ločena s po enim presledkom. V tretji vrstici so cela števila $y_1, y_2, \ldots, y_{n-1}, y_n$, ločena s po enim presledkom.

Omejitve: $1 \leq n \leq 100$; $-10^6 \leq x_0 < x_1 < \cdots < x_n \leq 10^6$; $|y_i| \leq 10^6$ za vse $i = 1, \ldots, n$. Pri prvih 20 % testnih primerov bodo vsi $y_i = 0$. Pri naslednjih 20 % testnih primerov bo $n \leq 10$. Pri naslednjih 20 % testnih primerov bodo vsi $y_i \geq 0$. Pri naslednjih 20 % testnih primerov bodo vsi x_i in y_i po absolutni vrednosti ≤ 100. Pri zadnjih 20 % testnih primerov ni posebnih dodatnih omejitev.

Izhodni podatki: izpiši eno samo celo število, namreč skupno ceno najcenejšega zaporedja operacij, s katerim lahko daljico $(x_0, 0, x_n)$ predelamo v skupino n daljic (x_{i-1}, y_i, x_i) za $i = 1, \ldots, n$.

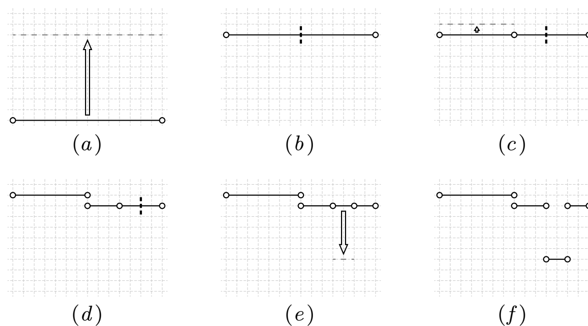
Primer vhoda:

```
4
0 7 10 12 14
9 8 3 8
```

Pripadajoči izhod:

```
39
```

Komentar: naslednja slika kaže najcenejše zaporedje premikov pri tem primeru. — (a) Začetno daljico $(0, 0, 14)$ premaknemo za 8 enot navzgor (cena: 8) in (b) jo pri $x = 7$ prerežemo (cena: 14) na $(0, 8, 7)$ in $(7, 8, 14)$. — (c) Prvo od teh dveh dvignemo še za eno enoto (cena: 1), drugo pa prerežemo pri $x = 10$ (cena: 7) na $(7, 8, 10)$ in $(10, 8, 14)$. — (d) Slednjo prerežemo pri $x = 12$ (cena: 4) na $(10, 8, 12)$ in $(12, 8, 14)$. — (e) Daljico $(10, 8, 12)$ premaknemo za 5 enot navzdol (cena: 5). — (f) Zdaj imamo štiri daljice $(0, 9, 7)$, $(7, 8, 10)$, $(10, 3, 12)$ in $(12, 8, 14)$, kot zahteva naloga, skupna cena operacij pa je bila $8 + 14 + 1 + 7 + 4 + 5 = 39$.



5. Krti

Družina krtov se vseljuje pod solatni vrt in načrtuje postavitev novega doma. Krtina je sestavljena iz n soban, vsak od m članov družine pa bo iz vsake sobane skopal natanko en rov, ki vodi do ene od soban (lahko tudi do iste). Ker so krti slabovidni, ne morejo predvideti, kako se bo rov končal, in lahko rove izkopljejo le tako, da se kasneje po njih lahko premikajo le v eni smeri — v isti smeri, kot so bili rovi izkopani. Namesto vida se krti poslužujejo dobrega voha. Še dolgo po gradnji krtine lahko namreč zaznajo, kateri član družine je izkopal določen rov, torej lahko navodila za premik med sobanami podamo kot zaporedje družinskih članov; ker iz vsake sobane vodi natanko en rov za vsakega člana, le sledimo rovom v zapisanem zaporedju.

Oče krt je ugotovil, da so pozabili načrtovati rov do površja. Ker krti slabo vidijo, mora biti ta rov postavljen tako, da se bo do njega dalo priti z enakimi navodili iz katerekoli sobane — krti namreč ne bodo mogli videti, v kateri sobani se nahajajo ob času kosila.

Napiši program, ki bo iz podatkov o strukturi krtine ugotovil, če izhodni rov lahko postavijo tako, da bodo enaka navodila vodila do sobane, iz katere vodi izhodni rov, ne glede na začetno sobano. Poišči tudi primer takšnih navodil, če obstajajo (ne nujno najkrajših).

Vhodni podatki: v prvi vrstici vhoda sta celi števili n in m, ločeni s presledkom. V naslednjih m vrsticah so opisani rovi vsakega družinskega člana; i-ta od teh vrstic vsebuje n s po enim presledkom ločenih števil $r_{i1}, r_{i2}, \ldots, r_{in}$, pri čemer za vsak j število r_{ij} pove, v katero sobano vodi rov, ki ga je i-ti družinski član izkopal iz j-te sobane. Velja lahko tudi $r_{ij} = j$.

Omejitve: $1 \leq n \leq 100$; $1 \leq m \leq 100$; za vsaka i in j velja $1 \leq r_{ij} \leq n$.

Pri tej nalogi je pet podnalog. Pri posamezni podnalogi dobiš 20 točk, če tvoj program pravilno reši vse testne primere te podnaloge, sicer pa pri njej ne dobiš nobene točke. Pri prvi podnalogi velja še dodatna omejitev: $n \leq 15$. Pri ostalih podnalogah ni dodatnih omejitev.

Izhodni podatki: v prvo vrstico izpiši `DA`, če za dani opis krtine obstaja navodilo, po kakršnem sprašuje naloga, oziroma `NE`, če ne obstaja. Če je odgovor `DA`, izpiši še eno vrstico s primerom takšnega navodila, kjer družinskega člana predstaviš z zaporedno številko, s katero se je ta član pojavil na vhodu. Števila v navodilu naj bodo vsa v eni vrstici in ločena s presledki. Posamezni član se lahko v navodilu pojavlja tudi večkrat. Navodilo naj bo sestavljeno iz največ n^3 elementov. Če je možnih več rešitev, je vseeno, katero od njih izpišeš.

Trije primeri vhoda in možni pripadajoči izhodi:

Vhod 1:	Izhod 1:	Vhod 2:	Izhod 2:	Vhod 3:	Izhod 3:

`3 2`
`3 1 1`
`2 3 3`
	`DA`
	`2 2`
		`2 1`
		`2 1`
			`NE`	`4 2`
				`2 3 4 1`
				`1 2 3 1`
					`DA`
					`2 1 1 1 2 1 1 1 2`

19. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2024

REŠITVE NALOG ZA PRVO SKUPINO

1. Budilka

Omislimo si nekaj globalnih spremenljivk: v zdaj bomo hranili trenutni čas (ki ga ob vsakem klicu podprograma JeCasZaAlarm povečamo za 1, ko pa doseže 1440, jo postavimo nazaj na 0); bujenje je čas, ko mora budilka vsak dan zazvoniti; casPoBujenju šteje, koliko minut je minilo od zadnjega zvonjenja ob bujenju; dremez pa je logična vrednost, ki pove, ali je uporabnik s pritiskom na gumb za dremež zahteval dodatno zvonjenje pet minut po bujenju.

Podprogram Dremez najprej s pomočjo spremenljivke casPoBujenju preveri, ali je bil poklican manj kot pet minut po zadnjem bujenju, in če je bil, mora le postaviti spremenljivko dremez na **true**.

Podprogram NastaviBudilko si zapomni novi čas bujenja in postavi dremez na **false**, saj naloga pravi, da nova nastavitev časa bujenja prekliče trenutno aktivno funkcijo dremeža. Poleg tega ta podprogram tudi postavi casPoBujenju na -1 kot znak, da zvonjenja pri pravkar nastavljenem novem času bujenja še ni bilo (zato je treba tudi klice funkcije Dremez zdaj ignorirati vse do prvega zvonjenja ob novem času).

Podprogram JeCasZaAlarm ima nekaj več dela. Povečati mora števec časa (spremenljivko zdaj) za 1 (pri čemer pazimo, da če naraste na 1440, kolikor je minut v dnevu, ga postavimo nazaj na 0); povečati mora števec časa po zadnjem bujenju (razen če je bil -1, kar pomeni, da zvonjenja pri trenutno nastavljenem času še ni bilo; poleg tega tudi ni treba, da ta števec povečujemo nad 6 — če je minilo od bujenja več kot pet minut, nam je vseeno, koliko časa točno je minilo); nato mora JeCasZaAlarm preveriti, če je novi čas ravno čas bujenja ali pa je zahtevana funkcija dremeža in je zdaj čas pet minut po bujenju; v teh primerih mora budilka zdaj zazvoniti, drugače pa ne. Ko budilka zazvoni ob času bujenja, postavimo tudi casPoBujenju na 0, ko pa zazvoni pet minut po bujenju, postavimo dremez na **false**, saj smo zahtevo po dodatnem zvonjenju s tem izpolnili.

```
int zdaj = −1;              // Trenutni čas (v minutah po polnoči).
int bujenje = −1;           // Čas, za katerega je naročeno bujenje.
int casPoBujenju = −1;      // Koliko minut je minilo od zadnjega bujenja?
bool dremez = false;        // Je vključena funkcija dremeža?

void NastaviBudilko(int t)
{
  bujenje = t;              // Zapomnimo si novi čas bujenja.
  dremez = false;           // Nova nastavitev budilke prekliče funkcijo dremeža.
  casPoBujenju = −1;        // Bujenja ob tem novem času še ni bilo.
}

void Dremez()
{
  // Dremež je mogoče vključiti le v prvih petih minutah po bujenju.
  if (casPoBujenju >= 0 && casPoBujenju < 5)
    dremez = true;          // Vključimo funkcijo dremeža.
}

bool JeCasZaAlarm()
{
  // Začenja se nova minuta.
  zdaj = (zdaj + 1) % 1440;

  // Merimo tudi čas po zadnjem bujenju, vendar le do 6 minut.
  if (casPoBujenju >= 0 && casPoBujenju <= 5) ++casPoBujenju;
```

```
  // Ali je prišel čas bujenja? Takrat zazvonimo in postavimo „casPoBujenju“ na 0.
  if (zdaj == bujenje) { casPoBujenju = 0; return true; }

  // Če je vključena funkcija dremeža, moramo zazvoniti tudi 5 minut po bujenju.
  else if (dremez && casPoBujenju == 5) { dremez = false; return true; }

  // Preostanek časa ni treba zvoniti.
  else return false;
}
```

Še implementacija te rešitve v pythonu:

```
zdaj = −1                  # Trenutni čas (v minutah po polnoči).
bujenje = −1               # Čas, za katerega je naročeno bujenje.
casPoBujenju = −1          # // Koliko minut je minilo od zadnjega bujenja?
dremez = False             # Je vključena funkcija dremeža?

def NastaviBudilko(t: int) −> None:
  global bujenje, dremez, casPoBujenju
  bujenje = t              # Zapomnimo si novi čas bujenja.
  dremez = False           # Nova nastavitev budilke prekliče funkcijo dremeža.
  casPoBujenju = −1        # Bujenja ob tem novem času še ni bilo.

def Dremez() −> None:
  global dremez
  # Dremež je mogoče vključiti le v prvih petih minutah po bujenju.
  if 0 <= casPoBujenju < 5:
    dremez = True          # Vključimo funkcijo dremeža.

def JeCasZaAlarm() −> bool:
  global zdaj, casPoBujenju, dremez

  # Začenja se nova minuta.
  zdaj = (zdaj + 1) % 1440;

  # Merimo tudi čas po zadnjem bujenju, vendar le do 6 minut.
  if 0 <= casPoBujenju <= 5: casPoBujenju += 1

  # Ali je prišel čas bujenja? Takrat zazvonimo in postavimo „casPoBujenju“ na 0.
  if zdaj == bujenje: casPoBujenju = 0; return True

  # Če je vključena funkcija dremeža, moramo zazvoniti tudi 5 minut po bujenju.
  elif dremez and casPoBujenju == 5: dremez = False; return True

  # Preostanek časa ni treba zvoniti.
  else: return False
```

2. Odbojkaške točke

Pri tej nalogi ni treba drugega, kot da pazljivo sledimo navodilom iz besedila naloge. Predpostavimo, da dobimo zaporedje dogodkov kot nize na standardnem vhodu, vsakega v svoji vrstici. Naš podprogram bo bral te nize v zanki, pri tem pa v spremeljivkah hranil podatke o tem, katera ekipa je nazadnje odbila žogo in kolikokrat jo sme še odbiti. Pri dogodku `I` lahko takoj zaključimo, da dobi točko tista ekipa, ki *ni* zadnja odbila žoge. Pri dogodkih `P1` in `P2` takoj dobi točko tista ekipa, ki ji žoga ni padla v polje. Pri odboju pa ločimo dva primera: če je žogo zdaj odbila druga ekipa kot prej, si to novo ekipo zapomnimo in tudi resetirajmo števec odbojev, ki so ji še preostali (načeloma ima ekipa po prvem odboju še dva, razen na začetku igre, pri servisu, ko po tem prvem odboju (servisu) žoge ne sme odbiti še enkrat); če pa je žogo odbila ista ekipa kot prej, moramo števec odbojev, ki so ji še preostali, zmanjšati za 1 — če pa je bil že enak 0, dobi točko druga ekipa (tista, ki ni zadnja odbila žoge).

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

int KdoDobiTocko()      // Vrne številko ekipe (1 ali 2).
```

```
{
   int ekipa = −1;          // Ekipa, ki je zadnja odbila žogo.
   int stOdbojev = 0;    // Kolikokrat sme ta ekipa še odbiti žogo?
   while (true)
   {
      // Preberimo naslednji dogodek.
      string s; cin >> s;

      // Če je šla žoga izven igrišča, dobi točko tista
      // ekipa, ki je ni zadnja odbila.
      if (s[0] == 'I') return 3 − ekipa;

      // Če je žoga padla v polje, dobi točko tista ekipa,
      // v katere polje žoga ni padla.
      else if (s[0] == 'P') return (s[1] == '1') ? 2 : 1;

      // Sicer imamo odboj. Morda je žogo odbila druga ekipa kot prej?
      if (int novaEkipa = s[1] − '0'; novaEkipa != ekipa)

         // Nova ekipa sme odbiti žogo še dvakrat, razen na začetku
         // igre (pri servisu), ko nima nobenega dodatnega odboja.
         stOdbojev = (ekipa < 0) ? 0 : 2,
         ekipa = novaEkipa;

      // Če ima trenutna ekipa preveč odbojev, dobi druga ekipa točko.
      else if (stOdbojev == 0) return 3 − ekipa;

      // Sicer samo zmanjšajmo števec preostalih odbojev.
      else −−stOdbojev;
   }
}
```

Še rešitev v pythonu:

```
def KdoDobiTocko() −> int:     # Vrne številko ekipe (1 ali 2).
   ekipa = −1          # Ekipa, ki je zadnja odbila žogo.
   stOdbojev = 0    # Kolikokrat sme ta ekipa še odbiti žogo?
   while True:

      # Preberimo naslednji dogodek.
      s = input()

      # Če je šla žoga izven igrišča, dobi točko tista
      # ekipa, ki je ni zadnja odbila.
      if s == "I": return 3 − ekipa

      # Če je žoga padla v polje, dobi točko tista ekipa,
      # v katere polje žoga ni padla.
      elif s[0] == 'P': return 2 if s[1] == '1' else 1

      # Sicer imamo odboj. Morda je žogo odbila druga ekipa kot prej?
      novaEkipa = 1 if s[1] == '1' else 2
      if novaEkipa != ekipa:

         # Nova ekipa sme odbiti žogo še dvakrat, razen na začetku
         # igre (pri servisu), ko nima nobenega dodatnega odboja.
         stOdbojev = 0 if ekipa < 0 else 2
         ekipa = novaEkipa

      # Če ima trenutna ekipa preveč odbojev, dobi druga ekipa točko.
      elif stOdbojev == 0: return 3 − ekipa

      # Sicer samo zmanjšajmo števec preostalih odbojev.
      else: stOdbojev −= 1
```

3. Tekstonim

Najprej preberimo telefonsko številko in si jo zapomnimo v spremenljivki. Nato v zanki berimo besede eno po eno; vsako besedo predelajmo v ustrezen niz števk (zamenjajmo črke s števkami, pri čemer si pomagajmo s tabelo, v kateri za vsako od 26 črk angleške abecede piše, v katero števko se preslika), nato pa z vgnezdeno zanko poiščimo vse pojavitve tega podniza v naši telefonski številki. Za vsako najdeno pojavitev potem

izpišimo telefonsko številko, v kateri smo najdeni podniz zamenjali z besedo: najprej torej izpišimo tisto, kar v telefonski številki nastopi pred najdenim podnizom, nato izpišimo trenutno besedo, nato pa še preostanek številke (za najdenim podnizom).

Za iskanje pojavitev podniza v nizu sicer obstajajo različni algoritmi, od preprostejših in manj učinkovitih do bolj zapletenih in učinkovitejših, vendar to ni zares predmet naše naloge. Uporabimo kar tisto, kar nam za iskanje podnizov v nizih ponuja standardna knjižnica našega programskega jezika. Oglejmo si primer takšne rešitve v C++:

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

int main()
{
  static const char stevke[ ] = "22233344455566677778889999";
  // Preberimo telefonsko številko.
  string telSt, beseda; cin >> telSt;
  // Preberimo in obdelajmo besede.
  while (cin >> beseda)
  {
    // Spremenimo znake te besede v ustrezne števke.
    string podniz = beseda;
    for (auto &c : podniz) c = stevke[c − 'a'];
    // Poiščimo prvo pojavitev tega podniza v telefonski številki.
    size_t i = telSt.find(podniz);
    while (i != telSt.npos)
    {
      // Izpišimo telefonsko številko z besedo namesto tega podniza.
      cout << telSt.substr(0, i) << beseda << telSt.substr(i + beseda.length()) << endl;
      // Poiščimo naslednjo pojavitev tega podniza.
      i = telSt.find(podniz, i + 1);
    }
  }
  return 0;
}
```

Še podobna rešitev v pythonu:

```
import sys
stevke = "22233344455566677778889999"
# Preberimo telefonsko številko.
telSt = sys.stdin.readline().strip()
# Preberimo in obdelajmo besede.
for beseda in sys.stdin:
  beseda = beseda.strip() # Odrežimo znak za konec vrstice.
  # Spremenimo znake te besede v ustrezne števke.
  podniz = "".join(stevke[ord(c) − ord('a')] for c in beseda.strip())
  # Poiščimo vse pojavitve tega podniza v telefonski številki.
  i = −1
  while (i := telSt.find(podniz, i + 1)) >= 0:
    # Izpišimo telefonsko številko z besedo namesto tega podniza.
    print(telSt[:i] + beseda + telSt[i + len(beseda):])
```

4. Premešani mozaik

Mozaik lahko urejamo oz. popravljamo sistematično, po vrsticah i od zgoraj navzdol in v vsaki vrstici po stolpcih j od leve proti desni. Če ploščica na trenutnem mestu (i, j) ni prave barve, torej če $b_{ij} \neq a_{ij}$, mora obstajati kakšna ploščica iskane barve a_{ij} v preostanku neurejenega dela mozaika (torej desno od trenutnega mesta v trenutni vrstici

ali pa kjerkoli v kakšni od nižje ležečih vrstic). Recimo, da jo najdemo na (i', j'), tako da je $b_{i'j'} = a_{ij}$. Če še ni v pravem stolpcu (torej če $j' \neq j$), lahko to ploščico z $|j' - j|$ premiki levo ali desno (levo, če je $j' > j$, oz. desno, če je $j' < j$) spravimo v stolpec j; nato pa, če še ni v pravi vrstici (torej če $i' \neq i$), jo z $|i' - i|$ premiki gor premaknimo v pravo vrstico. Zdaj imamo v celici (i, j) ploščico barve a_{ij}, kot smo si tudi želeli.

S tem, ko smo iskano ploščico premikali najprej vodoravno, šele nato pa navpično, smo zagotovili, da oba premika potekata v celoti znotraj še neurejenega dela mozaika, zato si s temi premiki ne bomo nikoli pokvarili že urejenega dela mozaika. Tako bomo korak za korakom sčasoma uredili celoten mozaik. Zapišimo naš postopek še s psevdokodo:

```
for i := 1 to h do for j := 1 to w do if b[i, j] ≠ a_ij:
  i' := i; j' := j;
  while b[i', j] ≠ a_ij: (* Poiščimo kakšno ploščico barve a_ij. *)
    j' := j + 1; if j' > w then j' := 1, i' := i + 1;
  (* Premaknimo jo na mesto (i, j). *)
  while j' > j do zamenjaj ploščici (i', j') in (i', j' − 1);
  while j' < j do zamenjaj ploščici (i', j') in (i', j' + 1);
  while i' > i do zamenjaj ploščici (i', j') in (i' − 1, j');
```

Ta postopek torej izvede pri vsaki ploščici v najslabšem primeru $O(w+h)$ premikov, zato je skupno število premikov reda $O(wh(w+h))$. Bolj neugodno pa je, da pri vsakem (i, j) porabimo v najslabšem primeru $O(wh)$ časa, da poiščemo neko ploščico primerne barve v neurejenem delu mozaika; časovna zahtevnost celotnega postopka je zato $O(w^2h^2)$. To bi se dalo še izboljšati. Za vsako barvo $\beta \in \{1, 2, \ldots, B\}$ bi lahko vzdrževali množico M_β vseh tistih položajev (i', j') v neurejenem delu mozaika, kjer so trenutno ploščice barve b. Seveda moramo vsakič, ko premaknemo neko ploščico barve β, pobrisati iz M_β njen stari položaj in dodati v M_β njen novi položaj. Ko nas potem pri nekem (i, j) zanima, kje bi našli (v neurejenem delu tabele) kakšno ploščico barve a_{ij}, lahko vzamemo poljuben par (i', j') iz množice $M_{a_{ij}}$. Časovna zahtevnost celotne rešitve se tako zmanjša na $O(wh(w + h))$.

Za predstavitev posamezne množice M_β lahko — da se bo dalo dodajati in brisati elemente v $O(1)$ časa — uporabimo slovar oz. razpršeno tabelo, šlo pa bi tudi z dvojno povezanim seznamom (*doubly linked list*), če v neki tabeli za vsako mesto na neurejenem delu mreže hranimo kazalec na pripadajoči element v enem od seznamov M_β.

5. Rokomet

Vhodno zaporedje, recimo $[a_1, \ldots, a_n]$, lahko pregledujemo v zanki; vsak a_i predstavlja skupino a_i strnjenih golov, ki jih je dala ekipa A, za njimi pa pride še en gol, ki ga je dala ekipa B (izjema je a_n, za katerim ni še enega gola ekipe B). Ta števila danih golov lahko seštevamo in tako sproti računamo skupno število golov vsake ekipe. Ob tem lahko tudi opazujemo, kdaj pride do spremembe vodstva: če pride v strnjeni skupini golov ekipe A do spremembe vodstva, se lahko to zgodi le tako, da ekipa A preide v vodstvo, medtem ko je bila prej v vodstvu ekipa B; in od trenutka, ko ekipa A tako preide v vodstvo, bo potem do konca trenutne strnjene skupine golov ekipe A ta ekipa ostala v vodstvu (in to z vse večjo prednostjo pred ekipo B); do spremembe vodstva pride torej lahko največ enkrat v taki strnjeni skupini golov in da to spremembo opazimo, je dovolj, če preverimo rezultat na koncu skupine. Podobno moramo potem seveda preveriti tudi rezultat po golu ekipe B.

Pri ugotavljanju, kdaj je prišlo do spremembe vodstva, moramo paziti še na to, da je lahko ena ekipa nekaj časa v vodstvu, potem druga izenači, potem prva spet povede in to ne šteje za spremembo v vodstvu. Ko je rezultat izenačen, moramo torej hraniti podatek o tem, katera ekipa je bila zadnja v vodstvu, ko rezultat še ni bil izenačen. Spodnji podprogram ima v ta namen spremenljivko `vodilni`, kjer vrednost 1 pomeni ekipo A, vrednost -1 ekipo B, vrednost 0 pa, da še nobena ekipa ni bila v vodstvu (na začetku tekme). Po vsaki strnjeni skupini golov posamezne ekipe pogledamo, kdo je zdaj v vodstvu (če rezultat ni izenačen) in to primerjamo s spremenljivko `vodilni`, da vidimo, če je prišlo do spremembe v vodstvu; potem si novega vodilnega tudi zapomnimo v omenjeni spremenljivki.

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

void AnalizirajTekmo(const vector<int> &goli)
{
   int stSprememb = 0, goliA = 0, goliB = 0, vodilni = 0;
   for (int i = 0; i < goli.size(); ++i) for (int ekipa = 0; ekipa < 2; ++ekipa)
   {
      // Najprej da ekipa A toliko golov, kot piše v vhodnem seznamu.
      if (ekipa == 0) goliA += goli[i];

      // Nato da ekipa B en gol, razen pri zadnjem elementu seznama.
      else if (i < goli.size() - 1) goliB += 1;

      // Če je izid izenačen, do spremembe v vodstvu ni prišlo.
      if (goliA == goliB) continue;

      // Poglejmo, če se je vodilna ekipa zamenjala.
      int noviVodilni = (goliA > goliB) ? 1 : -1;
      if (noviVodilni == vodilni) continue; // Ni sprememb.

      // Prvi gol na začetku tekme ne šteje za spremembo vodstva.
      if (vodilni != 0) ++stSprememb;

      // Zapomnimo si, kdo je zdaj v vodstvu.
      vodilni = noviVodilni;
   }
   // Izpišimo rezultate.
   cout << "Končni rezultat: " << goliA << " : " << goliB << "; "
        << stSprememb << " sprememb vodstva." << endl;
}
```

Zapišimo to rešitev še v pythonu:

```
def AnalizirajTekmo(goli):
   stSprememb = 0; goliA = 0; goliB = 0; vodilni = 0
   for i, stGolov in enumerate(goli):
      for ekipa in range(2):
         # Najprej da ekipa A toliko golov, kot piše v vhodnem seznamu.
         if ekipa == 0: goliA += stGolov

         # Nato da ekipa B en gol, razen pri zadnjem elementu seznama.
         elif i < len(goli) - 1: goliB += 1

         # Če je izid izenačen, do spremembe v vodstvu ni prišlo.
         if goliA == goliB: continue

         # Poglejmo, če se je vodilna ekipa zamenjala.
         noviVodilni = 1 if goliA > goliB else -1
         if noviVodilni == vodilni: continue # Ni sprememb.

         # Prvi gol na začetku tekme ne šteje za spremembo vodstva.
         if vodilni != 0: stSprememb += 1

         # Zapomnimo si, kdo je zdaj v vodstvu.
         vodilni = noviVodilni

   # Izpišimo rezultate.
   print(f"Končni rezultat: {goliA} : {goliB}; {stSprememb} sprememb vodstva.")
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Disleksija

Nalogo lahko rešimo z neke vrste požrešnim pristopom. Vhodne nize pregledujmo po vrsti in vsakega popravimo (zamenjajmo zvezdice s šesticami in deveticami) tako, da bo imel najmanjšo možno vrednost ob upoštevanju omejitve, da mora predstavljati večje število kot prejšnji niz. (Pri tem se lahko tudi izkaže, da niza sploh ni mogoče popraviti

tako, da bi predstavljal večje število kot prejšnji; tedaj zaključimo, da je problem za dani vhodni seznam nerešljiv.) Ker smo iz njega naredili najmanjše možno število, nas bo ta niz kasneje najmanj omejeval, ko bomo obravnavali naslednji niz v zaporedju. Na primeru iz besedila naloge bi tako najprej predelali `7` v `7`, nato `***` v `666`, nato `88*` v `886` in tako dalje.

Prepričajmo se, da ta pristop deluje. Za vhodne nize $(s_1, \ldots, s_k)$ bomo rekli, da je nabor števil $(x_1, \ldots, x_k)$ *dopustna rešitev*, če je $x_1 < x_2 < \ldots < x_k$ in če je mogoče vsak x_i mogoče dobiti iz s_i tako, da zamenjamo zvezdice s šesticami ter deveticami. Iz opisa našega postopka vidimo, da če najde neko rešitev, bo le-ta vsekakor dopustna; prepričati se moramo predvsem, da nikoli neupravičeno ne spregleda dopustne rešitve (in zmotno zatrdi, da rešitev ne obstaja). Z indukcijo po k bomo pokazali: če sploh obstaja kakšna dopustna rešitev, bo naš postopek tudi našel dopustno rešitev, in sicer tako z najmanjšim x_k. Pri $k = 1$ trditev drži, ker prejšnjega niza sploh ni in naš postopek preprosto popravi prvi niz tako, da dobi najmanjše možno število (v praksi to pomeni, da zamenja vse zvezdice s šesticami). Recimo zdaj, da trditev velja pri $k - 1$; prepričajmo se, da velja tudi za k. Recimo torej, da imamo vhodne nize $(s_1, \ldots, s_k)$ in da je problem rešljiv; naj bo $(x_1, \ldots, x_k)$ tista dopustna rešitev, ki ima najmanjši x_k. Potem je $(x_1, \ldots, x_{k-1})$ tudi dopustna rešitev problema $(s_1, \ldots, s_{k-1})$, torej je tudi ta problem rešljiv in zanj po induktivni predpostavki vemo, da bo naš postopek pri njem našel dopustno rešitev z najmanjšo možno vrednostjo zadnjega (torej $(k-1)$-vega) števila; recimo tej rešitvi $(y_1, \ldots, y_{k-1})$. Zanjo torej velja $y_{k-1} \leq x_{k-1}$, kar nam skupaj z $x_{k-1} < x_k$ dá $y_{k-1} < x_k$; zato bo naš postopek, ko bo pri reševanju problema $(s_1, \ldots, s_k)$ prišel do zadnjega niza, videl, da je pred tem predelal predzadnji niz s_{k-1} v število y_{k-1}, zato lahko, če drugega ne, predela zadnji niz v x_k (ker je to število večje od y_{k-1} in ker ga je mogoče dobiti iz s_k). In ker naš postopek predela vsak niz v najmanjše možno število, ki je še večje od prejšnjega, bo predelal s_k ravno v x_k: manjšega števila od tega ni (ker bi drugače obstajala dopustna rešitev, v kateri bi bilo k-to število manjše od x_k, to pa bi bilo v protislovju s tem, kako smo x_k sploh definirali). Naš postopek torej tudi po k nizih vrne dopustno rešitev, v kateri je k-to število najmanjše možno, prav to pa smo hoteli dokazati. □

Nismo pa rekli še ničesar o tem, kako sploh predelati določen niz v najmanjše možno število ob upoštevanju tega, da mora biti to število večje od prejšnjega. Ker naloga pravi, da so nizi lahko dolgi (do m znakov), bomo tudi števila imeli predstavljena kot nize; recimo, da je prejšnje število v nizu p, trenutni niz pa je s (in lahko poleg števk vsebuje tudi zvezdice). Na začetku naloge, ko niza p nimamo in nas ta niz torej nič ne omejuje, dobimo najmanjše primerno število iz s tako, da v njem vse zvezdice spremenimo v šestice. Razmislimo zdaj o primeru, ko obstaja prejšnji niz p. (1) Za začetek opazimo, da če je s daljši od p, bo število, ki ga bomo dobili iz njega, v vsakem primeru večje od p, ne glede na to, kako zamenjamo zvezdice s števkami; najmanjše možno število dobimo torej tako, da vse zvezdice zamenjamo s šesticami, ne pa z deveticami. (2) Podobno, če je s krajši od p, bo število, ki ga bomo dobili iz njega, v vsakem primeru manjše od p in lahko takoj zaključimo, da je problem nerešljiv.

(3) Ostane še možnost, da sta s in p enako dolga. Primerjajmo istoležne znake obeh nizov od leve proti desni, dokler ne opazimo prvega neujemanja. (3.1) Če nastopi to zaradi tega, ker ima s na trenutnem mestu manjšo števko kot p, lahko takoj zaključimo, da bo število, ki ga bomo dobili iz s, manjše od p, in lahko obupamo. (3.2) Podobno, če je neujemanje zato, ker ima s na trenutnem mestu večjo števko kot p, bo število iz njega gotovo večje od p in lahko vse zvezdice zamenjamo s šesticami.

(3.3) Drugače pa je neujemanje zato, ker ima s na trenutnem mestu zvezdico, p pa števko. (3.3.1) Če ima p tu števko 9, moramo tudi zvezdico v s spremeniti v 9, sicer bi iz s nastalo število, manjše od p; spremenimo torej zvezdico v devetico in nadaljujmo, kot da tu ni bilo neujemanja. (3.3.2) Če ima p tu števko 7 ali 8, moramo zvezdico v s spremeniti v 9, sicer bi iz s nastalo število, manjše od p; po tej spremembi pa ima s na tem mestu (in to je prvo neujemanje) višjo števko kot p, zato bo predstavljal večje število kot p ne glede na to, kaj se bo zgodilo kasneje v nizu; vse preostale zvezdice lahko zato spremenimo v šestice, da bo nastalo čim manjše število. (3.3.3) Če ima p tu eno od števk od 0 do 5, bo iz s nastalo večje število kot p, tudi če spremenimo zvezdico v šestico (kaj šele, če jo spremenimo v devetico); zato lahko spremenimo trenutno zvezdico

v šestico, vse ostale (desno od trenutne) pa tudi.

(3.3.4) Ostane še možnost, da ima p tukaj šestico. Če zvezdico v s spremenimo v šestico, se bosta tukaj niza ujemala in ni vnaprej očitno, ali bo mogoče v preostanku s-ja spremeniti zvezdice tako, da bo nastalo število, večje od p. Največje možno število bi dobili, če vse te preostale zvezdice spremenimo v devetice; preverimo torej, ali bi bilo to število (recimo mu x) večje od p. To ne pomeni, da bomo res pripravili celotno število x kot nov niz (in ga potem primerjali s p); to bi bilo potratno. Namesto tega bomo šli le po istoležnih znakih s-ja naprej od trenutnega mesta in jih primerjali z istoležnimi znaki p-ja, pri čemer pa bomo morebitne zvezdice v s-ju v mislih zamenjali z deveticami. Čim opazimo neujemanje (ali pa pridemo do konca nizov), se ustavimo. (3.3.4.1) Če je $x \leq p$, to pomeni, da trenutne zvezdice v s ne smemo spremeniti v šestico, torej jo moramo v devetico; s tem pa je že zagotovljeno, da bo število iz s večje od p, zato lahko potem vse preostale zvezdice v s spremenimo v šestice. (3.3.4.2) Če pa bi bilo $x > p$, to pomeni, da lahko spremenimo trenutno zvezdico s-ja v šestico; zdaj se p in s na trenutnem mestu ujemata in lahko nadaljujemo s primerjanjem istoležnih znakov. (Pomembna podrobnost: če je bilo na območju, ki smo ga že pregledali, ko smo primerjali x in p, v nizu s kaj zvezdic, so morale biti očitno tam v p-ju devetice, kajti drugače bi že tam opazili neujemanje med x (ki ima devetice, kjer ima s zvezdice) in p (če bi ta tam imel neko števko, manjšo od 9). Zato na tem območju ne bo nikoli več prišlo do primera (3.3.4). Območja, ki jih pregledamo za potrebe primerjave x in p pri primerih (3.3.4), se torej med seboj ne prekrivajo in vsa skupaj so lahko dolga le m znakov.)

Razmislimo še o časovni zahtevnosti te rešitve. Pri posameznem s moramo v najslabšem primeru dvakrat pregledati niza s in p, enkrat neposredno in enkrat v okviru primerjav med x in p pri možnosti (3.3.4). Pri nizih dolžine m torej porabimo $O(m)$ časa za predelavo posameznega niza oz. $O(nm)$ za celoten vhodni seznam n takšnih nizov. Oglejmo si še implementacijo te rešitve v C++:

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

bool ZamenjajZvezdice(const string &p, string &s)
{
  const int d = s.length();

  // Če je trenutni niz krajši od prejšnjega, potem bo manjši od njega
  // ne glede na to, v kaj spremenimo zvezdice.
  if (d < p.length()) return false;

  // Če je trenutni niz daljši od prejšnjega, bo gotovo tudi večji,
  // četudi spremenimo vse zvezdice v šestice.
  bool vecji = (d > p.length()); // Ali smo že poskrbeli, da je s večji od p?

  // Če sta niza enako dolga, ju primerjajmo od leve proti desni in
  // spremenimo zvezdice v trenutnem nizu tako, da bo nastal najmanjši
  // možni niz, ki je še večji od prejšnjega (če je to sploh mogoče).
  int i; // indeks znakov, ki ju primerjamo
  for (i = 0; i < d && ! vecji; ++i)
    if (s[i] == '*') {
      // Če ima p tu devetico, moramo tudi s-jevo zvezdico spremeniti v devetico.
      if (p[i] == '9') s[i] = '9';

      // Če ima p tu 7 ali 8, moramo v s narediti 9; če ima p tu 0..5, lahko v s naredimo 6.
      // Po tem bo s zagotovo večji od p.
      else if (p[i] > '6') { s[i] = '9'; vecji = true; }
      else if (p[i] < '6') { s[i] = '6'; vecji = true; }

      else {
        // Tu ima p šestico; ali lahko s postane večji od p, če tudi v s naredimo šestico?
        // Primerjajmo s p-jem niz, ki nastane, če v s vse preostale zvezdice
        for (int j = i + 1; j < d; ++j)                    // spremenimo v devetke.
          if (char c = (s[j] == '*') ? '9' : s[j]; c != p[j]) {
            vecji = (c > p[j]); break; }
```

```
            // Če na ta način lahko dobimo niz, večji od p, potem iz trenutne
            // zvezdice v s naredimo šestico in nadaljujmo s primerjanjem istoležnih znakov;
            // sicer pa iz trenutne zvedice naredimo devetko, s pa je s tem zagotovo že večji od p.
            s[i] = (vecji) ? '6' : '9'; vecji = ! vecji; } }
        // Če ima s tu manjšo števko kot p, je problem nerešljiv.
        else if (s[i] < p[i]) return false;
        // Če ima s tu večjo števko kot p, je gotovo večji od p.
        else if (s[i] > p[i]) vecji = true;
    // Če smo prišli do konca nizov s in p, ne da bi uspeli zagotoviti,
    // da je s večji od p, potem je problem nerešljiv.
    if (! vecji) return false;
    // Sicer je s že zaradi števk na indeksih od 0 do i − 1 večji od p;
    // morebitne zvezdice v preostanku s-ja lahko postanejo šestice.
    for ( ; i < d; ++i) if (s[i] == '*') s[i] = '6';
    return true;
}

int main()
{
    string p, s;         // Prejšnji in trenutni niz.
    while (cin >> s) // Prebirajmo vhodne nize.
    {
        // Zamenjajmo zvezdice v „s“ s števkami; če to ni mogoče, končajmo izvajanje.
        if (! ZamenjajZvezdice(p, s)) { cout << "Problem je nerešljiv." << endl; return 1; }
        // Izpišimo popravljeni niz in si ga zapomnimo v p.
        cout << s << endl; p = s;
    }
    return 0;
}
```

2. Preurejanje

Nalogo lahko rešimo na več načinov. Ena možnost je, da se zgledujemo po znanem postopku urejanja z mehurčki (*bubble sort*). Razdelimo v mislih naš seznam na štiri četrtine, recimo $[A, B, C, D]$. Za urejanje z operacijo, ki jo dopušča naša naloga, smemo uporabiti polovico indeksov naenkrat, torej dve četrtini. Uredimo najprej četrtini A in B; s tem pridejo največji elementi iz prve polovice seznama v B. Nato uredimo četrtini B in C; po tem so največji elementi iz prvih treh četrtin seznama v C. Nato uredimo četrtini C in D; zdaj so največji elementi celotnega seznama v D (in urejeni naraščajoče), torej prav tam, kjer morajo biti. Zdaj je torej zadnja četrtina seznama dobila svojo pravo končno podobo. V nadaljevanju lahko na podoben način uredimo prve tri četrtine (ki zdaj vsebujejo najmanjših $3n/4$ elementov prvotnega vhodnega seznama). Najprej uredimo četrtini A in B, da pridejo večji elementi iz prve polovice v B; nato uredimo B in C, da pridejo največji elementi iz prvih treh četrtin v C; in to so prav tisti elementi, ki morajo na koncu biti v C. Tudi C ima torej že svojo končno podobo. Ostala nam je le še prva polovica seznama, ki jo lahko uredimo z enim korakom. Tako smo izvedli vsega skupaj šest urejanj (AB, BC, CD, AB, BC, AB).

Še ena možnost je, da se zgledujemo po urejanju z zlivanjem (*merge sort*). Uredimo najprej prvo polovico seznama posebej in nato drugo polovico posebej. V A so zdaj najmanjši elementi iz prve polovice, v C pa najmanjši iz druge polovice. V naslednjem koraku uredimo A in C, pa bodo v A prišli najmanjši elementi iz obeh polovic, torej iz celotnega seznama; tako je zdaj v A točno tisto, kar mora na koncu biti tam. Podobno vidimo, da imamo (zaradi prvih dveh urejanj) v B največje elemente iz prve polovice, v D pa največje iz druge; uredimo torej zdaj B in D, pa pridejo v D največji elementi iz celega seznama (ki točno tam tudi morajo biti). Neurejena nam je ostala le še srednja polovica seznama (četrtini B in C), ki jo lahko uredimo v še enem koraku. Tako smo izvedli vsega skupaj pet urejanj (AB, BC, AC, BD, BC).

Še boljšo rešitev pa dobimo, če se ne držimo strogo urejanja po četrtinah. Spomnimo se, da pri tej nalogi vidimo vsebino celotne tabele, le spreminjati je ne smemo drugače kot z urejanjem polovice elementov naenkrat. V prvem koraku vključimo v urejanje

celotno A in pa tistih $n/4$ indeksov iz preostalih treh četrtin, kjer so najmanjši elementi teh treh četrtin tabele. Po tem urejanju je v A najmanjša četrtina elementov celotne tabele, to pa je točno tisto, kar mora v A biti tudi na koncu urejanja. Ostalo nam je le še urejanje zadnjih treh četrtin tabele. V drugem koraku vključimo v urejanje celotno B in pa tistih $n/4$ indeksov iz zadnjih dveh četrtin, kjer so njuni najmanjši elementi. Po tem urejanju pride v B najmanjših $n/4$ elementov izmed tistih, ki niso v A, tako da ima tudi B zdaj že svojo končno podobo. Neurejena je ostala le še zadnja polovica tabele, ki jo lahko uredimo s še enim korakom. Tako smo tabelo uredili s samo tremi urejanji.

Prepričajmo se, da je ta rešitev najboljša možna. Da tabele ni mogoče vedno urediti z enim samim urejanjem, je jasno, kajti eno urejanje pusti vsaj $n/2$ elementov pri miru, nekatere tabele pa so premešane tako, da ni niti en element na pravem mestu (npr. tabela, ki je urejena padajoče namesto naraščajoče). Ali je nemara mogoče urediti vsako tabelo z dvema urejanjema? Če imamo tabelo, v kateri noben element ni na pravem mestu, potem mora vsak element sodelovati v vsaj enem urejanju (sicer se ne bo premaknil), to pa je mogoče le, če se urejanji ne prekrivata — če sodeluje polovica elementov v enem, polovica pa v drugem urejanju. Toda predstavljajmo si tabelo $x = (2, 3, \ldots, n, 1)$. Za vsak $i = 1, \ldots, n-1$ velja, da morata indeksa i in $i+1$ sodelovati oba v istem urejanju, kajti le tako ima lahko element z vrednostjo $i+1$ kakršno koli možnost, da pride z indeksa i (kjer se nahaja v začetnem stanju tabele x) na indeks $i+1$ (kjer bo moral biti, ko bo tabela urejena). Tako torej vidimo, da bi morala sodelovati v istem urejanju indeksa 1 in 2, pa indeksa 2 in 3, pa indeksa 3 in 4 itd.; vsi indeksi bi morali torej sodelovati v istem urejanju, kar pa je nemogoče, saj lahko urejamo le $n/2$ elementov naenkrat. Nemogoče je torej, da bi vsako tabelo lahko uredili z le dvema urejanjema; naša rešitev s tremi je v splošnem najboljša možna.

3. Boggle

Naloga je zelo primerna za reševanje z rekurzijo. V zanki preglejmo vse znake mreže; na mestih, kjer najdemo prvo črko iskane besede, poženemo rekurzijo, da preverimo, ali bi se dalo od tam najti še preostanek besede. Posamezni rekurzivni klic mora pregledati vseh osem sosedov prejšnjega znaka na mreži, da vidi, če se tam morda nahaja naslednja črka iskane besede; če se, izvedemo od tam vgnezden rekurzivni klic za preostanek besede. Pazimo pa na to, da iste črke mreže ne smemo uporabiti po večkrat v posamezni besedi; zato na začetku rekurzivnega klica čez pravkar uporabljeno črko mreže napišimo neveljavni znak `#`, pred vrnitvijo iz klica pa vpišimo črko nazaj, da povrnemo mrežo v prvotno stanje.

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

// Poišče v mreži m niz s od s[i + 1] naprej, pri čemer začne v
// sosedih polja (xp, yp), v katerem se nahaja znak [i].
bool Poisci(vector<string> &m, const string &s, int xp, int yp, int i)
{
  if (i + 1 >= s.length()) return true; // Morda smo že na koncu niza.
  m[yp][xp] = '#'; // Da ne bomo iste črke uporabili še kdaj.
  const int h = m.size(), w = m[0].length();

  // Preglejmo vse sosede polja (xp, yp).
  for (int dy = -1; dy <= 1; ++dy) for (int dx = -1; dx <= 1; ++dx)
  {
    if (dx == 0 && dy == 0) continue;
    int x = xp + dx, y = yp + dy;

    // Pazimo, da ne pademo čez rob mreže.
    if (y < 0 || y >= m.size() || x < 0 || x >= m[y].size()) continue;

    // Ali je na polju (x, y) naslednji znak niza s?
    if (m[y][x] != s[i + 1]) continue;

    // Če je, poiščimo preostanek niza z rekurzivnim klicem.
    if (Poisci(m, s, x, y, i + 1)) { m[yp][xp] = s[i]; return true; }
  }
```

```
   // Če pridemo do sem, niza s nismo našli.
   m[yp][xp] = s[i]; return false;
}

bool Poisci(vector<string> &m, const string &s)
{
   for (int y = 0; y < m.size(); ++y) for (int x = 0; x < m[y].length(); ++x)
      // Če se na polju (x, y) nahaja prva črka niza s,
      // poženimo od tam rekurzijo, da poiščemo preostanek niza.
      if (m[y][x] == s[0] && Poisci(m, s, x, y, 0)) return true;
   return false;
}
```

Glavni podprogram mora iti le v zanki po vseh besedah seznama, za vsako preveriti, če se pojavlja v mreži, in take besede šteti:

```
int KolikoNizov(vector<string> mreza, const vector<string> &besede)
{
   int stNajdenih = 0;
   for (const string &beseda : besede)
      if (Poisci(mreza, beseda)) ++stNajdenih;
   return stNajdenih;
}
```

To rešitev bi se dalo na razne načine še izboljšati; na primer, lahko bi si vnaprej za vsako črko abecede pripravili seznam, kje v mreži se pojavlja, da nam ne bi bilo treba pri vsaki besedi iti čez celo mrežo, da najdemo pojavitve prve črke te besede. Besede bi lahko zložili v drevo po črkah (*trie*) in se pri rekurziji spuščali po tem drevesu; tako bi v enem zamahu poiskali vse besede in prihranili nekaj časa, če se več besed ujema v prvih nekaj črkah.

4. Prepisovanje

Problema, ki ju rešujeta Janko in Metka, imata sicer na prvi pogled zelo različno zgodbico, v resnici pa sta si zelo podobna. Metka zbira sličice in skuša zbrati vse; Janko osvetljuje ceste in skuša osvetliti vse. Metka lahko z nakupom paketa dobi nekaj sličic; Janko lahko postavi luč v neko križišče in tako osvetli nekaj cest. Metka hoče kupiti čim manj paketov, Janko pa postaviti čim manj luči. Še najbolj opazna razlika je ta, da je pri Metki lahko posamezna sličica prisotna v poljubno veliko paketih, pri Janku pa vsaka cesta povezuje natanko dve križišči; v tem pogledu je torej Metkin problem splošnejši od Jankovega — prav to pa pomeni, da lahko Janko brez težav „prevede“ svoj problem na Metkinega.

Janko lahko torej definira primerek Metkinega problema s C sličicami in K paketi. Vsaki cesti v Jankovem problemu tako ustreza neka sličica v Metkinem, vsakemu križišču v Jankovem pa neki paket v Metkinem. Če cesta c_i v Jankovem problemu povezuje križišči u_i in v_i, naj bo sličica c_i v Metkinem problemu vključena v paketa u_i in v_i (in v noben drug paket). Janko naj nato reši Metkin problem z njenim algoritmom, potem pa za vsak paket, ki ga je treba kupiti v rešitvi Metkinega problema, postavi luč na križišče, ki ustreza temu paketu v Jankovem problemu. (Pravzaprav, ker naloga pravi, da mora izračunati le najmanjše število postavljenih luči, je dovolj že, če samo vrne število paketov, kupljenih v rešitvi Metkinega problema.)

Prepričajmo se, da tako dobimo veljavno rešitev Jankovega problema, torej da so vse ceste res osvetljene. Cesti c_i ustreza v Metkinem problemu sličica c_i; in v rešitvi Metkinega problema je med kupljenimi paketi gotovo neki tak paket — recimo mu p —, ki vsebuje sličico c_i, sicer tisto ne bi bila veljavna rešitev Metkinega problema; Janko je torej na križišče p postavil luč; in ker je sličica c_i pripadala paketu p, to pomeni, da je v Jankovem problemu križišče p eno od tistih dveh, ki ju povezuje cesta c_i; ker je na p luč, je cesta c_i osvetljena. Ta razmislek velja za poljubno cesto, torej so res vse ceste osvetljene.

Prepričajmo se še, da je dobljena rešitev Jankovega problema res tista z najmanj lučmi. Luči ima toliko, kolikor je bilo v rešitvi Metkinega problema kupljenih paketov.

Recimo, da bi obstajala neka druga rešitev Jankovega problema z manj lučmi. Za vsako križišče p, kjer v tej drugi rešitvi stoji luč, bi se dalo v Metkinem problemu kupiti paket p; in ker so v Jankovem problemu vse ceste osvetljene, so s tem v Metkinem problemu kupljene vse sličice; tako smo dobili veljavno rešitev Metkinega problema, ki ima toliko paketov, kot ima luči naša domnevna druga rešitev Jankovega problema, to pa je manj kot prvotna rešitev Jankovega problema, ki je imela toliko luči, kolikor je imela paketov prvotna rešitev Metkinega problema (tista, ki jo je vrnil Metkin algoritem); toda za Metkin algoritem vemo, da vrne rešitev z najmanj kupljenimi paketi, torej smo v protislovju. V resnici torej ni mogoče, da bi se dalo Jankov problem rešiti še z manj lučmi, tako da je bila naša rešitev Jankovega problema res pravilna.

Janko torej reši svoj problem v $t(C, K)$ časa, k temu pa moramo načeloma prišteti še nekaj časa, potrebnega za predelavo podatkov Jankovega problema v Metkinega. Lahko si predstavljamo, da najprej inicializiramo K praznih seznamov (za vsak paket oz. križišče po enega), nato pa gremo v zanki po vseh cestah in vsako cesto c_i dodamo v seznama u_i in v_i; tako dobimo sezname, ki predstavljajo vsebino paketov pri Metkinem problemu. Takšna predelava nam torej vzame $O(C + K)$ časa.

5. Laserji

Ker je balonov in strelov veliko (vsaj pri večjih testnih primerih), si ne moremo privoščiti, da bi šli pri vsakem strelu po vseh balonih in preverjali, katere je zadel. Pomagamo pa si lahko z dejstvom, da so streli samo vodoravni in navpični. Če pride navpičen strel $x = s_j$, bi bilo koristno imeti seznam balonov na tej x-koordinati; to so ravno tisti baloni, ki jih bo ta strel zadel. Podobno je za obravnavo vodoravnega strela $y = s_j$ koristno imeti seznam balonov na tej y-koordinati. Imejmo torej za vsako x- in y-koordinato (vsako tako, na kateri je prisoten vsaj en balon) po en seznam, v katerem bodo številke tistih balonov, ki ležijo na tej x- oz. y-koordinati. Da bomo do seznama za določen x oz. y lažje prišli, pa zložimo te sezname v slovar (npr. razred unordered_map iz C++ove standardne knjižnice), kjer bo ključ koordinata, pripadajoča vrednost pa seznam balonov na njej.

Zdaj imamo torej za vsak strel pri roki seznam balonov, ki jih ta strel načeloma zadene; paziti pa moramo na to, da je nekatere od teh balonov mogoče zadel že nek zgodnejši strel po drugi koordinati. Zato imejmo še neko tabelo oz. vektor, v katerem označujemo, kateri baloni še niso počeni. Ko pride nov strel, gremo po balonih v ustreznem seznamu in tiste, ki še niso počeni, zdaj označimo kot počene (in zmanjšamo števec ne-počenih balonov).

Paziti moramo še na eno podrobnost: lahko pride tudi več strelov v isti smeri in po isti koordinati. Potratno bi bilo vsakič znova pregledovati seznam balonov na tej koordinati, saj so že po prvem strelu vsi gotovo počeni. Zato po vsakem strelu pobrišimo seznam balonov, ki smo ga pravkar pregledali.

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

int main()
{
   // Preberimo balone in jih zložimo v sezname.
   int n, s; cin >> n >> s;
   unordered_map<int, vector<int>> poX, poY;
   for (int i = 0; i < n; ++i) {
      int x, y; cin >> x >> y;
      poX[x].emplace_back(i); poY[y].emplace_back(i); }

   // Obdelajmo strele in izpišimo rezultate.
   vector<bool> pocen(n, false);
   while (s-- > 0)
   {
      int koord; char smer; cin >> koord >> smer;
      // Sprehodimo se po seznamu balonov, ki jih ta strel zadene,
      // in poglejmo, kateri so zdaj počili.
```

```
    auto &v = (smer == 'v' ? poX : poY)[koord];
    for (int i : v) if (! pocen[i]) pocen[i] = true, --n;

    // Seznam pobrišimo, da ga ne bomo še enkrat pregledovali,
    // če pride kasneje še kdaj enak strel.
    v.clear();

    cout << n << endl; // Izpišimo število ne-počenih balonov.
  }
  return 0;
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Kiralnost

Pojdimo v zanki po vhodni mreži in iščimo pojavitve znaka `C`. Vemo, da vsaka taka pojavitev predstavlja po eno molekulo in da je tisti `C` povezan z `R`-jem v eni od štirih možnih smeri: levo, desno, gor ali dol. Takšno smer lahko opišemo s parom $(\triangle_x, \triangle_y)$, ki pove, kako se v tej smeri spreminjata x- in y-koordinata; eno od števil $\triangle_x$ in $\triangle_y$ je enako 0, drugo pa ± 1. Da najdemo pravo smer, preizkusimo v zanki vse štiri možne smeri in poglejmo, če v tisti smeri vidimo `R` dva koraka od trenutnega `C`-ja in če je med njima povezava (torej znak `-`, če je $\triangle_y = 0$, oz. `|`, če je $\triangle_x = 0$);[1] to slednje je pomembno, ker je sicer mogoče tudi, da tisti `R` pripada neki drugi molekuli in je povezan z nekim drugim `C`-jem. Če smo našli `C` na položaju (x, y), mora biti pripadajoči `R` na položaju $(x + 2\triangle_x, y + 2\triangle_y)$, povezava med njima pa na $(x + \triangle_x, y + \triangle_y)$,

Ko na ta način najdemo pravo smer (s tem smo pravzaprav ugotovili, kako je molekula zasukana), se lahko v mislih postavimo v `C`, pogledamo proti `R` in se vprašamo, kateri atom je na naši levi: če je to `O`, je molekula levo-ročna, če pa je `N`, je desno-ročna. Da pridemo do tega atoma, moramo narediti iz `C` en korak v smeri $(\triangle_x, \triangle_y)$ in nato še dva koraka v levo, pravokotno na to smer, torej v smeri $(\triangle_y, -\triangle_x)$. (Tu smo predpostavili, da y-koordinate naraščajo smeri navzdol po mreži.) Pogledati moramo torej, kateri atom je položaju $(x + \triangle_x + 2\triangle_y, y + \triangle_y - 2\triangle_x)$. Odvisno od tega, ali tam vidimo `O` oz. `N`, povečamo števec levih oz. desnih molekul; ko pregledamo celotno mrežo, oba števca izpišemo.

Omenimo še to, da nam `C`-jev ni treba iskati v prvih ali zadnjih dveh stolpcih ali vrsticah, kajti če bi bil `C` tam, bi taka molekula štrlela čez rob mreže, za kar pa naloga zagotavlja, da se ne bo zgodilo. Zato nam tudi ni treba skrbeti, da bi, ko v okolici `C`-ja iščemo `R` in `O` ali `N`, kdaj dobili neveljavne koordinate, ki bi ležale zunaj mreže.

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

int main()
{
  // Preberimo vhodne podatke.
  int w, h; cin >> w >> h;
  vector<string> A(h); for (auto &s : A) cin >> s;

  // Preglejmo mrežo in preštejmo molekule.
  const int DX[] = {1, 0, -1, 0}, DY[] = {0, 1, 0, -1};
  int levih = 0, desnih = 0;
  for (int y = 2; y < h - 2; ++y) for (int x = 2; x < w - 2; ++x)
    // Pri vsakem C poglejmo, v kateri smeri je njegov pripadajoči R.
```

[1] Pri tem ni treba zares paziti, katerega od znakov `-` in `|` vidimo med `C` in `R`. V naših molekulah so okrog `C`-ja na vseh štirih poljih, ki imajo skupno stranico s `C`-jem, povezave, ki povezujejo ta `C` z drugimi štirimi atomi iste molekule. Če torej najdemo `C` in `R` dva koraka narazen, je med njima zagotovo povezava, ki pripada tistemu `C`-ju, zato pa gotovo ne more biti pravokotna na smer med `C` in `R` — lahko bodisi povezuje ta `C` s tem `R`-jem bodisi je to ena od poševnih povezav, ki povezujeta `C` z `N`-jem in `O`-jem. Dovolj je torej, če preverimo, ali je med `C` in `R` ne-poševna povezava, ni pa treba preveriti, ali je vodoravna ali navpična.

```
    if (A[y][x] == 'C') for (int d = 0; d < 4; ++d) {
      int dx = DX[d], dy = DY[d];
      // Ali sta v tej smeri R in povezava med C in R?
      if (A[y + dy][x + dx] != (dy == 0 ? '-' : '|')) continue;
      if (A[y + 2 * dy][x + 2 * dx] != 'R') continue;
      // Poglejmo, kateri atom je levo od povezave med C in R.
      if (A[y + dy − 2 * dx][x + dx + 2 * dy] == 'O') ++levih; else ++desnih; }
  // Izpišimo rezultate.
  cout << levih << " " << desnih << endl; return 0;
}
```

2. Lučke

Recimo, da izvedemo q poizvedb; niz n bitov, ki smo ga oddali pri p-ti poizvedbi, naj bo recimo $b_{p1}b_{p2}\ldots b_{pn}$. Vrednost $b_{p\ell}$ je torej enaka 1, če je bila ℓ-ta lučka pri p-ti poizvedbi prižgana, ali 0, če je bila ugasnjena.

Na te podatke pa lahko pogledamo tudi drugače: niz q bitov $B_\ell := b_{1\ell}b_{2\ell}\ldots b_{q\ell}$ nam za ℓ-to lučko pove, pri katerih poizvedbah je bila prižgana in pri katerih ugasnjena. Če imata dve lučki, recimo k in ℓ, popolnoma enaka niza ($B_k = B_\ell$), to pomeni, da sta bili pri vsaki poizvedbi ali obe prižgani ali obe ugasnjeni. Če bi tidve lučki namesto na koordinatah (x_k, y_k) in (x_ℓ, y_ℓ) stali na koordinatah (x_k, y_ℓ) in (x_ℓ, y_k), bi bili odgovori na vse poizvedbe še vedno enaki kot prej: pri posamezni poizvedbi sta x_k in x_ℓ bodisi obe prisotni v odgovoru bodisi nobena, prav tako pa tudi y_k in y_ℓ. Tako torej ne bomo mogli ugotoviti, ali je x_k v paru z y_k ali z y_ℓ.

Potreben pogoj za to, da lahko določimo koordinate vseh lučk, je torej ta, da so njihovi nizi $B_1, \ldots, B_n$ vsi različni.

Temu moramo v večini primerov dodati še en pogoj: nobena lučka ne sme imeti za B_ℓ niza samih ničel, kajti takšna bi bila pri vseh poizvedbah ugasnjena in mi nikoli ne bi izvedeli njenih koordinat. Edina izjema je primer, ko je $w = h = n$. Naloga namreč zagotavlja, da je vsaka lučka v svojem stolpcu in tudi vsaka v svoji vrstici; če je torej lučk prav toliko kot vrstic, potem je lahko ena lučka ves čas ugasnjena in bomo na koncu vedeli, da stoji v tisti vrstici, v kateri ne stoji nobena druga lučka; enako pa je seveda tudi pri stolpcih.

Vseh možnih nizov q bitov je 2^q; mi pa zahtevamo, da mora obstajati vsaj n takih nizov, pri čemer (razen če je $w = h = n$) ne velja tisti iz samih ničel. Pišimo $d = 1$, če je $w = h = n$, sicer pa $d = 0$; potem torej zahtevamo, da mora obstajati vsaj $n + 1 - d$ različnih nizov. Dobili smo pogoj $2^q \geq n + 1 - d$. Najmanjši celoštevilski q, ki mu ustreza, je $\lceil \log_2(n + 1 - d) \rceil$ (ali, kar je enakovredno, $1 + \lfloor \log_2(n - d) \rfloor$); to je vrednost, ki jo ima besedilo naloge v mislih, ko v razdelku o točkovanju govori o najmanjšem možnem številu poizvedb m. (Razlika zaradi tega, ker je $d = 1$ namesto 0, se torej pozna le v primeru, če je n potenca števila 2; tako lahko na primer pri $w = h = n = 8$ rešimo nalogo že s tremi poizvedbami, medtem ko drugače pri $n = 8$ potrebujemo že štiri poizvedbe.)

Primeren nabor nizov $B_1, \ldots, B_n$ je potem ta, da za B_ℓ vzamemo kar dvojiški zapis števila $\ell - d$ (po potrebi z nekaj vodilnimi ničlami na levi, tako da bo dolg ravno m bitov) — z drugimi besedami, večinoma uporabimo dvojiški zapis števil od 1 do n, razen če je $w = h = n$, ko lahko uporabimo dvojiški zapis števil od 0 do $n - 1$. Ali še drugače: pri p-ti poizvedbi so prižgane natanko tiste lučke ℓ, pri katerih je bit p v dvojiškem zapisu števila $\ell - d$ prižgan.

Odgovore na poizvedbe lahko potem uporabimo takole: imejmo tabelo, v kateri za vsak stolpec računamo številko lučke v njem; na začetku so povsod v njej ničle; po p-ti poizvedbi pa za tiste stolpce, v katerih senzor takrat zaznava prižgano lučko, vemo, da ima številka lučke v tistem stolpcu prižgan bit b, torej tudi na tistih mestih naše tabele prižgimo bit b. Enako naredimo seveda tudi za vrstice. Po vseh poizvedbah iz teh dveh tabel preprosto odčitamo številko lučke za vsako vrstico ali stolpec.

```
#include <iostream>
#include <string>
```

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

int main()
{
  int w, h, n; cin >> w >> h >> n;
  // Poseben primer: če je n = w = h, lahko rešimo nalogo za prvih n − 1 lučk
  // in za n-to potem vemo, da je na edinih še neuporabljenih koordinatah.
  int d = (n == w && n == h) ? 1 : 0;
  // V naslednjih vektorjih za vsako možno koordinato počasi nastaja
  // številka lučke na tej koordinati.
  vector<int> xKatera(w, 0), yKatera(h, 0);
  for (int b = 0; (n − d) >> b; ++b)
  {
    // V naslednji poizvedbi prižgimo tiste lučke,
    // katerih številka ima bit b prižgan.
    cout << "POIZVEDBA ";
    for (int i = 1; i <= n; ++i) cout << (((i − d) >> b) & 1);
    cout << endl << flush;
    // Preberimo odgovor.
    string s; cin >> s;
    for (int x = 0; x < w; ++x) if (s[x] == '1') xKatera[x] |= 1 << b;
    cin >> s;
    for (int y = 0; y < h; ++y) if (s[y] == '1') yKatera[y] |= 1 << b;
  }
  // Zdaj poznamo koordinate vseh lučk.
  vector<int> xLucka(n + 1, −1), yLucka(n + 1, −1);
  for (int x = 0; x < w; ++x) xLucka[xKatera[x] + d] = x;
  for (int y = 0; y < h; ++y) yLucka[yKatera[y] + d] = y;
  // Izpišimo rezultate.
  cout << "REZULTATI" << endl;
  for (int i = 1; i <= n; ++i) cout << xLucka[i] + 1 << " " << yLucka[i] + 1 << endl;
  return 0;
}
```

3. Matrika

Če je (i, j) fiksna celica in če velja $k \leq i$ in $\ell \leq j$, bomo rekli, da fiksna celica (i, j) *pokriva* celico (k, ℓ). Te celice tvorijo pravokotnik z ogliščema $(1, 1)$ in (i, ℓ).

Če neko celico pokriva več fiksnih celic, bo moralo biti število v njej manjše od vrednosti vseh tistih fiksnih celic; to pa je enakovredno omejitvi, naj bo manjše od najmanjše izmed tistih vrednosti. Koristno je torej obravnavati fiksne vrednosti od manjših proti večjim, kajti manjše nas bolj omejujejo. Uredimo torej fiksne celice naraščajoče po vrednosti in jih v tem vrstnem redu oštevilčimo; t-ta od njih naj ima položaj (i_t, j_t) in vrednost a_t. Pravokotniku, ki ga pokriva, recimo $R_t = \{(k, \ell) : 1 \leq k \leq i_t, 1 \leq \ell \leq j_t\}$. Zdaj si lahko predstavljamo naslednji požrešni postopek za vpisovanje števil v matriko:

1 začnimo s popolnoma prazno matriko;
2 **for** $t := 1$ **to** k:
3   če (i_t, j_t) ni več prazna, je problem nerešljiv (in končajmo);
4   vpiši v (i_t, j_t) število a_t;
5   v vsako še prazno celico iz R_t vpiši najmanjše število,
      ki ga še nima nobena celica matrike;
6   če smo pri tem že uporabili kakšno število, večje od a_t,
      je problem nerešljiv (in končajmo);
7 v vsako celico, ki je še prazna, vpiši najmanjše število,
    ki ga še nima nobena celica matrike;

Prepričajmo se, da ta postopek deluje. Z indukcijo po t bomo dokazali naslednje: (a) če je mogoče zapolniti celice iz $R_1 \cup \ldots \cup R_t$ s števili (različnimi z območja od 1 do nm) tako, da so v vsakem R_u (za $u = 1, \ldots, t$) vsa števila $\leq a_u$ in da je a_u na (i_u, j_u), potem

bo naš postopek po t iteracijah te celice res zapolnil v skladu s temi omejitvami, vse ostale celice pa bodo takrat še prazne; (b) če pa teh celic ni mogoče zapolniti v skladu s temi omejitvami, se bo postopek nekje v prvih t iteracijah prekinil. (c) Poleg tega, če pridemo do konca t-te iteracije, bo veljalo naslednje: če bo v neki celici, ki ni ena od prvih t fiksnih celic, vpisano število r, bodo vsa števila od 1 do $r-1$ tudi vpisana v neke druge (morda fiksne) celice.

Pri $t = 0$ trditev drži, saj je mreža takrat še povsem prazna in omenjena trditev pri $t = 0$ tudi ne zatrjuje ničesar drugega kot to.

Recimo zdaj, da trditev drži pri $t - 1$; prepričajmo se, da drži tudi pri t. Na začetku t-te iteracije glavne zanke je torej (ker trditev drži pri $t - 1$) območje $R_1 \cup \ldots \cup R_{t-1}$ primerno zapolnjeno s števili, preostanek mreže pa je prazen. Če zdaj (v t-ti iteraciji zanke) vrstica 3 opazi, da (i_t, j_t) ni prazna, to pomeni, da to celico pokriva ena od prvih $t - 1$ fiksnih vrednosti, recimo u-ta (za neki u z območja $1 \leq u < t$); zato bo morala biti vrednost v tej celici na koncu $\leq a_u$; toda obenem bo morala biti, ker je to ravno t-ta fiksna celica, vrednost v njej na koncu enaka a_t, to pa je (ker je $u < t$ in imamo fiksne vrednosti urejene naraščajoče) večje od a_u; zato je problem nerešljiv in vrstica 3 upravičeno prekine izvajanje.

Sicer se t-ta iteracija zanke nadaljuje in v vrsticah 4 in 5 zapolni s števili vse celice iz R_t, ki niso bile zapolnjene že od prej. Če se izvajanje v vrstici 6 ne prekine, lahko po njej zaključimo, da so vse vrednosti v R_t manjše ali enake a_t: tiste, ki so bile zapolnjene že v prejšnjih iteracijah, recimo v u-ti iteraciji za $u < t$, so bile $\leq a_u$ (po induktivni predpostavki) in so zato $< a_t$ (ker je $u < t$); tiste pa, ki smo jih vpisali zdaj v vrsticah 4–5, so $\leq a_t$ zato, ker bi drugače vrstica 6 prekinila izvajanje. Mreža je torej zdaj (na koncu t-te iteracije) zapolnjena tako, kot zahteva (a) za trditev pri t; preverimo še (c): ali je mogoče, da je v neki ne-fiksni celici število r, pri čemer pa nekega manjšega števila, recimo $q < r$, nima še nobena celica? Pred trenutno iteracijo to ni veljalo (ker je veljala naša trditev za $t - 1$); moralo se je torej zgoditi v trenutni iteraciji; r smo torej vpisali v vrstici 5; toda ker tam vsakič uporabimo najmanjše število, ki ga nima še nobena celica, je moral biti takrat, ko smo se v vrstici 5 odločili uporabiti r, tudi q že nekam vpisan; tako smo v protislovju; tudi (c) torej drži.

Preveriti moramo le še, da se ne more zgoditi, da bi vrstica 6 prekinila postopek neupravičeno. Z drugimi besedami, pokazati moramo, da če vrstica 6 prekine postopek, je problem za prvih t fiksnih vrednosti res nerešljiv — to bo pokazalo, da pri t velja tudi točka (b) naše trditve. Če je vrstica 6 prekinila postopek, to pomeni, da smo neki ne-fiksni celici dodelili neko številko $b > a_t$; ker za našo rešitev po vrstici 5 velja (c), imamo očitno tudi celice z vsemi številkami od 1 do $b - 1$; torej je vseh celic v $R_1 \cup \ldots \cup R_t$ vsaj b, kar je več kot a_t; torej je nemogoče, da bi vse te celice dobile številke, manjše ali enake a_t; toda vsako od teh celic pokriva ena od prvih t fiksnih celic, kar torej zahteva, da je vrednost v vsaki celici manjša ali enaka od neke a_u za $u \leq t$, s tem pa tudi manjša ali enaka od a_t; celic je torej več, kot je na voljo (dovolj majhnih) številk, zato je problem res nerešljiv in je vrstica 6 upravičeno prekinila naš postopek.

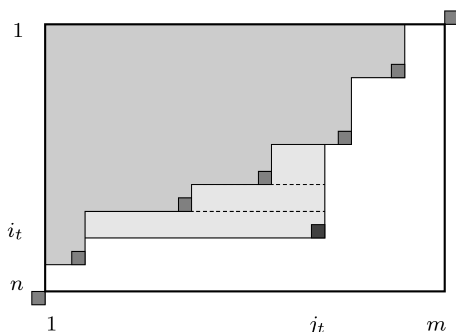
Zdaj torej vidimo, da naša trditev res velja po vsaki iteraciji zanke, torej tudi po k-ti; ko se zanka konča, smo vpisali primerna števila na vse celice, ki jih pokriva kakšna fiksna celica. Preostalih celic (ki jih bo zapolnila vrstica 7) ne pokriva nobena fiksna celica, zato je vseeno, kakšne vrednosti dobijo, samo da so $\leq nm$. Ker ima mreža skupno nm celic in smemo uporabljati števila od 1 do nm, je neuporabljenih števil prav toliko kot neuporabljenih celic, zato bo lahko vrstica 7 zapolnila preostale celice s še neuporabljenimi števili, ne da bi bila kdaj prisiljena uporabiti števila nad nm. Na koncu postopka je zato celotna mreža primerno izpolnjena.

Imamo torej požrešni algoritem, s katerim je mogoče preveriti, če je problem rešljiv, vendar pa je za velike mreže, s kakršnimi imamo opravka pri tej nalogi, prepočasen. Recimo, da smo v t-ti iteraciji naše glavne zanke; v vrstici 3 moramo znati hitro preveriti, ali leži (i_t, j_t) na območju, ki ga pokrivajo dosedanje fiksne celice, torej v uniji $R_1 \cup \ldots \cup R_{t-1}$; v vrstici 6 pa moramo znati hitro preveriti, ali bi bila vrstica 5, če bi jo bili izvedli (kar pa je ne bomo, saj pri velikih mrežah nimamo ne časa ne pomnilnika za to), prisiljena uporabiti kakšno število nad a_t. Videli smo že, da zaradi lastnosti (b) velja, da če je vrstica 5 uporabila kakšno tako število, potem je uporabila že tudi vsa manjša, torej je celic v $R_1 \cup \ldots \cup R_t$ več kot a_t. Po drugi strani, če je celic v tem območju več kot

a_t, je tudi neizogibno, da bo vrstica 5 prisiljena uporabiti kakšno število nad a_t, saj števil do vključno a_t preprosto ni dovolj. Tako torej vidimo, da vrstica 6 prekine izvajanje natanko v primeru, ko je $|R_1 \cup \ldots \cup R_t| > a_t$. Naš postopek lahko zdaj zapišemo takole:

funkcija JeRešljiva:
1 $R :=$ prazna množica;
2 **for** $t := 1$ **to** k:
3 **if** $(i_t, j_t) \in R$ **then return false**;
4–5 $R := R \cup R_t$;
6 **if** $|R| > a_t$ **then return false**;
7 **return true**;

Območje R je unija več pravokotnikov z zgornjim levim kotom $(1, 1)$, njihovi spodnji desni koti pa so (i_u, j_u) za $u = 1, \ldots, t$. Če se v nekem stolpcu j razteza območje R recimo i vrstic globoko, mora biti to zaradi nekega pravokotnika, ki ima višino $i_u = i$ in širino vsaj $j_u \geq j$; tak pravokotnik pa je prisoten tudi v vseh stolpcih levo od j, torej se območje tudi tam razteza vsaj i vrstic globoko. Iz tega sledi, da ko se premikamo proti desni, se globina območja R lahko le zmanjšuje ali pa ostaja enaka, nikoli pa se ne more povečati; spodnji in desni rob območja R ima stopničasto obliko, v vogalih teh stopnic pa so fiksne točke, kot kaže naslednja slika.



Primer dodajanja fiksne celice (i_t, j_t), ki jo predstavlja črni kvadratek. Temno sivo stopničasto območje je R pred dodajanjem nove fiksne celice; še temnejši kvadratki na njem so fiksne celice na robu stopnišča, ki jih bomo morali hraniti v našem zaporedju. Svetlo sivo območje je na novo pokrito po dodajanju, torej $R_t - R$; črtkane črte kažejo, kako ga lahko razrežemo na vodoravne trakove, da izračunamo njegovo površino.

Območje R lahko torej predstavimo z zaporedjem fiksnih celic v vogalih stopnic; urejene naj bodo od leve proti desni, tako da njihove številke stolpcev j_t naraščajo, številke vrstic i_t pa padajo. Koristno je na začetku in na koncu zaporedja kot stražarja vzdrževati še celici $(n + 1, 0)$ in $(0, m + 1)$, ki ležita na zunanji strani mreže ob njenem spodnjem levem in zgornjem desnem vogalu ter predstavljata začetek in konec stopnišča.

Razmislimo, kaj je treba v tem zaporedju spremeniti, ko hočemo v R dodati novo fiksno celico (i_t, j_t). Zaporedje je urejeno padajoče po i; poglejmo, kam v to zaporedje pade i_t; recimo, da med u in v, tako da je $i_u \geq i_t > i_v$. Fiksna celica v in vse kasnejše imajo premajhen i, da bi lahko pokrivale novo celico t; pač pa imajo dovolj velik i celica u in vse zgodnejše v zaporedju; med njimi pa ima največji j ravno celica u; če bo torej sploh katera od obstoječih fiksnih celic pokrivala novo celico t, je to celica u. Če u pokriva celico t, lahko zaključimo, da je problem nerešljiv (kar ustreza vrstici 3 naše psevdokode zgoraj).

Sicer pa vidimo, da imajo v in kasnejše celice v zaporedju dovolj majhen i, da jih morda lahko pokrije nova celica t; da jih res pokrije, morajo imeti tudi dovolj majhen j, in ker j po zaporedju narašča, pride v poštev v in še prvih nekaj naslednjih celic v zaporedju, dokler ne pridemo do take, katere j je večji od j_t. Celice, ki jih je t pokrila, pobrišemo iz zaporedja, nato pa v zaporedje vstavimo t (tik pred prvo tako, ki ima $j > j_t$ in ki zato ni več pokrita).

Med pregledovanjem zaporedja in brisanjem celic, ki jih je t zdaj pokrila, lahko tudi računamo površino na novo pokritega območja in tako pridemo do novega števila pokritih celic, ki ga potrebujemo v vrstici 6 naše psevdokode. Na novo pokrito območje lahko v mislih razrežemo na vodoravne pravokotne „trakove“; če sta u in v dve zaporedni fiksni točki na dosedanjem robu, dobimo trak, ki obsega stolpce od j_u+1 do j_t ter vrstice od $i_v + 1$ do $\min\{i_u, i_t\}$. Po en tak trak dobimo za vsako v, ki jo je t pokrila, in še za prvo naslednjo (nepokrito) celico našega zaporedja.

Vidimo, da bomo morali znati v zaporedju hitro iskati po številki vrstice, se premikati naprej po njem, brisati in vrivati elemente; primerna podatkovna struktura za to je

kakšna uravnotežena vrsta drevesa, npr. rdeče-črno drevo, kjer vsaka od naštetih operacij vzame po $O(\log k)$ časa. (V C++ lahko uporabimo razred map iz standardne knjižnice.) V našem primeru bomo morali za vsako fiksno celico izvesti eno iskanje v drevesu, po eno dodajanje in zato tudi največ eno brisanje; skupaj ima ta postopek časovno zahtevnost $O(k \log k)$. Prav toliko časa porabimo pred tem tudi za urejanje fiksnih točk po vrednosti.

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

int main()
{
  // Preberimo vhodne podatke.
  struct Fiksna { long long x, y, a; };
  long long w, h, k; scanf("%lld %lld %lld", &h, &w, &k);
  vector<Fiksna> fiksne(k);
  for (auto &f : fiksne) scanf("%lld %lld %lld", &f.y, &f.x, &f.a);

  // Slovar „rob“ vzdržuje fiksne celice na robu stopnišča,
  // urejene padajoče po y (in s tem naraščajoče po x).
  // Na začetku dodajmo kot stražarja spodnji levi in zgornji desni kot mreže.
  map<long long, Fiksna, greater<long long>> rob;
  rob[h + 1] = {0, h + 1, 0}; rob[0] = {w + 1, 0, 0};
  long long povrsina = 0; // število celic, ki jih pokrivajo doslej obdelane fiksne celice

  // Preglejmo fiksne celice po naraščajoči vrednosti.
  sort(fiksne.begin(), fiksne.end(), [] (const auto &u, const auto &v) { return u.a < v.a; });
  for (auto &f : fiksne)
  {
    auto it = rob.upper_bound(f.y); // it = prva na robu, ki ima y < f.y
    auto prej = it; --prej;          // prej = zadnja, ki ima y ≥ f.y
    long long xLevo = prej->second.x, ySpodaj = f.y;

    // Ali leži f na območju, ki ga pokriva prej->second?
    if (f.x <= xLevo) { printf("NE\n"); return 0; }

    // Pobrišimo z roba tiste, ki jih pokrije f.
    while (true)
    {
      // V vrsticah g.y < y ≤ ySpodaj so zdaj na novo pokrite
      // celice v stolpcih xLevo < x ≤ f.x.
      auto &g = it->second;
      povrsina += (f.x - xLevo) * (ySpodaj - g.y);

      // Če f ne pokriva g-ja, smo z brisanjem končali.
      if (g.x > f.x) break;

      // Sicer g pobrišimo in se premaknimo po robu naprej,
      // zapomnimo pa si koordinati g-ja, ki bosta prišli prav pri
      // računanju površine v naslednji iteraciji.
      xLevo = it->second.x; ySpodaj = g.y;
      it = rob.erase(it);
    }
    // Ali je pokritih več celic, kot je na voljo števil zanje?
    if (povrsina > f.a) { printf("NE\n"); return 0; }

    // Dodajmo f na rob stopnišča.
    rob[f.y] = f;
  }
  // Če smo prišli do konca, je problem rešljiv.
  printf("DA\n"); return 0;
}
```

4. Vodoravne daljice

Če neko daljico prerežemo pri nekem x, dobimo dve daljici s krajiščem v tem x in tudi po morebitnih nadaljnjih rezih bomo imeli še vedno po dve daljici s krajiščem x. V poštev za x pridejo torej le vrednosti x_i iz vhodne datoteke, saj imajo vse daljice, ki jih moramo dobiti na koncu našega zaporedja operacij, krajišča v teh x_i in nikjer drugje. To pa pomeni, da je vsaka daljica, s katero imamo med našim delom opravka, oblike (x_i, y, x_j) za $0 \leq i < j \leq n$ (in da lahko tako daljico prerežemo le pri $x = x_k$ za $i < k < j$).

Podobno tudi, če neko daljico premaknemo v navpični smeri, recimo na višino y', je tak premik smiseln le, če je y' enak enemu od y_i, torej če bo na tej y-koordinati ena od daljic, ki jih moramo dobiti na koncu našega zaporedja operacij. O tem se lahko prepričamo s protislovjem. Pa recimo, da je pri nekem testnem primeru najcenejše zaporedje operacij takšno, ki vsebuje neki premik na $y' \notin Y := \{y_1, \ldots, y_n\}$. Oglejmo si zadnji tak premik v tem zaporedju operacij. Daljica, ki smo jo tako premaknili, očitno še ni na svojem končnem položaju, ker na koncu ne bo nobene daljice na višini y'; na njej bomo torej gotovo izvedli še kakšno operacijo.

(1) Če bomo to daljico kasneje še enkrat premaknili v navpični smeri, bo ta kasnejši premik gotovo na neko $y'' \in Y$ (kajti pravkar opravljeni premik na y' je bil po predpostavki zadnji premik na neko višino, ki ni iz Y); toda potem bi lahko že pravkar opravljeni premik izvedli do y'' namesto do y'; cena tega bi bila bodisi enaka kot pri dveh ločenih premikih (če sta bila oba gor ali oba dol) bodisi celo manjša (če je bil eden od teh dveh premikov gor, eden pa dol); torej premik na $y' \notin Y$ ni bil potreben.

(2) Druga možnost pa je, da bomo pravkar premaknjeno daljico, ki je zdaj na višini y', najprej enkrat ali večkrat prerezali. Toda tudi tako nastali koščki daljice še niso na svojem končnem položaju (ker na koncu ne bo nobene daljice na višini y') in bo treba vsakega od njih sčasoma premakniti na neko drugo y-koordinato. Ker je bil po predpostavki naš pravkar opravljeni premik na y' zadnji premik na neko višino zunaj Y, bodo ti kasnejši premiki posameznih koščkov vsi šli na višine iz Y. Če so vsi ti premiki v smeri navzgor ali vsi v smeri navzdol, bi bilo ceneje, če bi premaknili (v tisto smer) celotno daljico, preden smo jo prerezali; toda mi smo predpostavili, da delamo z optimalnim zaporedjem.

Torej bodo šli nekateri od teh premikov navzgor, nekateri pa navzdol; naša trenutna višina y' torej leži med dvema višinama iz Y. Recimo, da v mislih elemente množice Y uredimo in jih v tem vrstnem redu oštevilčimo: $y_{(1)} < y_{(2)} < \cdots < y_{(m)}$, kjer je $m = |Y|$ (to je morda manjše od n, ker niso nujno vse y-koordinate končnih daljic različne). Rekli smo, da y' leži med dvema višinama iz Y; na primer na $y_{(i-1)} < y' < y_{(i)}$. Vse koščke naše trenutne daljice, ki jih bomo premaknili navzgor (oz. navzdol), bomo torej premaknili vsaj do $y_{(i)}$ (oz. do $y_{(i-1)}$). Recimo, da jih g premaknemo gor, d pa dol; cena teh premikov (navzgor do $y_{(i)}$ oz. navzdol do $y_{(i-1)}$) je torej $C_0 := (y_{(i)} - y') \cdot g + (y' - y_{(i-1)}) \cdot d$.

(2.1) Recimo, da je $g > d$. Če bi trenutno daljico dvignili z y' na $y_{(i)}$, jo šele tam razrezali na koščke in potem premikali te koščke naprej, bi bila cena skupaj $C_1 := (y_{(i)} - y') + (y_{(i)} - y_{(i-1)}) \cdot d$; prvi člen je cena premika celotne daljice (koščkov, ki smo jih prej premikali gor, zdaj ni več treba premikati gor do $y_{(i)}$, ker so že tam), drugi člen pa je cena premika tistih koščkov, ki jih je treba premakniti navzdol (in je ta premik zdaj malo daljši kot prej). Razlika med novo in staro ceno je $C_1 - C_0 = (y_{(i)} - y')(1 - g + d)$; prvi faktor je > 0, drugi pa je (zardi $g > d$) ≥ 0, zato je $C_1 - C_0 \leq 0$. Nova cena torej ni nič višja od stare, znebili pa smo se premika na y', ker lahko novi dodatni premik na $y_{(i)}$ združimo s prejšnjim premikom (na y') v en sam premik, ki ni nič dražji od dveh ločenih premikov.

(2.2) Če je $g < d$, je razmislek analogen kot v prejšnjem odstavku, le da razmišljamo o možnosti, da trenutno daljico najprej spustimo na $y_{(i-1)}$ in jo šele tam prerežemo.

(2.3) Ostane še možnost, da je $g = d$. Daljico, ki smo jo ravnokar premaknili na y', bi bili lahko v resnici premaknili za malo manj, recimo za ε manj, v isti smeri kot prej; torej na $y' - \varepsilon$, če je bil to premik navzgor, oz. na $y' + \varepsilon$, če je šlo za premik navzdol. Vrednost $\varepsilon > 0$ izberimo dovolj majhno, da bi bila nova višina še vedno med $y_{(i-1)}$ in $y_{(i)}$. Na tej novi višini potem daljico razrežimo in premikajmo naprej vsak košček posebej. Kakšna je cena tega scenarija v primerjavi s prvotnim? Premik daljice pred rezanjem se je pocenil za ε, premiki posameznih koščkov pa so se pocenili ali podražili za

ε; pocenijo se tisti, katerih premik je v nasprotni smeri od pravkar opravljenega premika daljice, ostali pa se podražijo; ker je $g = d$, je obojih enako in podražitve in pocenitve se med seboj izničijo. Ostane le prihranek ε pri ceni premika daljice pred rezanjem. Novi scenarij je torej cenejši od prvotnega, kar pa je protislovje, ker smo predpostavili, da je bil že prvotni najcenejši.

Tako torej vidimo, da če imamo v najcenejšem zaporedju operacij neki premik na $y' \notin Y$, lahko rešitev izboljšamo ali vsaj ne poslabšamo, če se tega premika znebimo. □

V nadaljevanju se torej lahko omejimo na take navpične premike, pri katerih je nova y-koordinata enaka eni od $y_1, \ldots, y_n$. Možne y-koordinate daljice so torej le $y_1, \ldots, y_n$ in še $y_0 := 0$ (kar je višina začetne daljice pred prvim premikom); za krajišča pa smo že prej videli, da so možna le $x_0, x_1, \ldots, x_n$.

Po tem razmisleku postane naloga zelo primerna za reševanje z dinamičnim programiranjem. Naj bo $f(i, j, h)$ cena najcenejšega zaporedja operacij, s katero predelamo daljico (x_i, y_h, x_{j+1}) v skupino daljic (x_t, y_{t+1}, x_{t+1}) za $t = i, \ldots, j$. Rezultat, po katerem sprašuje naloga, je potem $f(0, n-1, 0)$. Funkcijo f lahko računamo z rekurzivnim razmislekom: daljico lahko najprej premaknemo na neko novo višino $y_{h'}$, najcenejša rešitev od tam do konca pa je $f(i, j, h')$; lahko pa daljico najprej prerežemo pri nekem x_k in potem na najcenejši način rešimo dva ločena podproblema, $f(i, k-1, h)$ in $f(k, j, h)$. Tako dobimo:

$$f(i, j, h) = \min\{\ \min\{|y_{h'} - y_h| + f(i, j, h') : 1 \le h' \le n\},\\ (x_{j+1} - x_i) + \min\{f(i, k-1, h) + f(k, j, h) : i < k \le j\}\ \}.$$

Za potrebe računanja funkcije ta rekurzivna zveza še ni najbolj prikladna, ker se vrednosti $f(i, j, \cdot)$ pojavljajo tako na levi kot na desni strani te enačbe. S tem pravzaprav naši funkciji dovolimo, da večkrat zaporedoma premakne daljico, ne da bi jo vmes rezala, od česar pa seveda ne more biti nobene koristi. Vpeljimo še pomožno funkcijo $g(i, j, h)$, ki jo definirajmo enako kot f, toda z dodatno omejitvijo, da mora biti prva operacija rez in ne premik. Zdaj dobimo:

$$g(i, j, h) = (x_{j+1} - x_i) + \min\{f(i, k-1, h) + f(k, j, h) : i < k \le j\} \text{ in}$$
$$f(i, j, h) = \min\{|y_{h'} - y_h| + g(i, j, h') : 1 \le h' \le n\}.$$

Tu se torej g sklicuje le na vrednosti f za krajša podzaporedja končnih daljic (od i do $k-1$ ali od k do j namesto od i do j), f pa se sklicuje le na g in ne več neposredno na f. V minimumu, s katerim računamo f, je prisotna tudi možnost $h' = h$, ko daljice sploh ne premaknemo, ampak v njej takoj izvedemo naslednji rez.

Vrednosti funkcij f in g računajmo sistematično po naraščajoči $j - i$, torej dolžini opazovanega intervala; že izračunane vrednosti shranjujmo v neko tabelo oz. vektor, da jih bomo imeli kasneje pri roki, ko jih bomo potrebovali. Časovna zahtevnost te rešitve je $O(n^4)$: izračunati moramo $O(n^3)$ vrednosti funkcij f oz. g, pri vsaki pa moramo izračunati minimum po $O(n)$ možnostih (glede izbire h' oz. k).

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

int main()
{
  // Preberimo vhodne podatke.
  int n; cin >> n;
  vector<int> xi(n + 1), yi(n + 1);
  for (int i = 0; i <= n; ++i) cin >> xi[i];
  yi[0] = 0; for (int i = 1; i <= n; ++i) cin >> yi[i];
  // f(i, j, h) bo rešitev podproblema, kjer začnemo z daljico (x[i], y[h], x[j + 1])
  // in bi radi na koncu dobili daljice (x[t], y[t + 1], x[t + 1]) za i ≤ t ≤ j.
  const int inf = 2'000'000 * 2 * n; // več od vsake f(i, j, h)
  vector<int> f_(n * n * (n + 1), inf);
  auto f = [n, &f_] (int i, int j, int h) -> int& {
```

```
      return f_[(i * n + j) * (n + 1) + h]; };
   // Ko je i == j, lahko daljico le premaknemo na pravo višino.
   for (int i = 0; i < n; ++i) for (int h = 0; h <= n; ++h)
      f(i, i, h) = abs(yi[h] − yi[i + 1]);
   // Daljši primeri.
   vector<int> g(n + 1);
   for (int d = 1; d < n; ++d) for (int i = 0; i + d < n; ++i)
   {
      int j = i + d; // Rešujemo podproblem, kjer začnemo z daljico (x[i], y[h], x[j + 1]).
      // V g[h] zapišimo najboljšo rešitev, če moramo daljico najprej prerezati.
      for (int h = 0; h <= n; ++h) {
         auto &G = g[h]; G = inf;
         for (int k = i + 1; k <= j; ++k)
            // Lahko bi najprej prerezali pri x[k].
            G = min(G, f(i, k − 1, h) + f(k, j, h) + xi[j + 1] − xi[i]); }
      // V f(i, j, h) zapišimo najboljšo rešitev, če smemo daljico najprej premakniti.
      for (int h = 0; h <= n; ++h) {
         auto &F = f(i, j, h);
         for (int hh = 0; hh <= n; ++hh)
            // Lahko bi daljico najprej premaknili na y[hh].
            F = min(F, g[hh] + abs(yi[hh] − yi[h])); }
   }
   // Izpišimo rezultat. Tretji parameter je h = 0, ker začnemo na višini y[0] = 0.
   cout << f(0, n − 1, 0) << endl; return 0;
}
```

Še opomba glede gornje implementacije: ko računamo vrednosti f ali g kot minimum več možnosti, smo vrednost na začetku inicializirali na $4n \cdot 10^6$, ker je to gotovo več od vsake prave vrednosti funkcij f ali g. O tem se lahko prepričamo takole: posamezni podproblem lahko rešimo s kvečjemu $n-1$ rezi in nato kvečjemu n premiki, posamezni rez ali premik pa ima ceno največ $2 \cdot 10^6$, ker so koordinate v naših vhodnih podatkih po absolutni vrednosti $\leq 10^6$. Zato je podproblem gotovo mogoče rešiti za ceno, manjšo od $2n \cdot 2 \cdot 10^6$.

5. Krti

Za začetek vpeljimo nekaj oznak. Množico soban v krtini imenujmo $Q = \{1, 2, \ldots, n\}$, množico krtov pa $\Sigma = \{1, 2, \ldots, m\}$. Vsako zaporedje 0 ali več krtov si lahko predstavljamo kot navodilo za premik po krtini; če je s neko tako zaporedje, naj bo $\delta(q, s)$ številka sobane, v kateri končamo, če začnemo v q in sledimo navodilu s.

Recimo, da obstaja navodilo s, po kakršnem sprašuje naloga; če začnemo v vsaki od n možnih soban in sledimo temu navodilu, dobimo n različnih poti po krtini, ki pa se (če navodilo res ustreza zahtevam naloge) vse končajo v isti sobani: $\delta(q, s)$ je enaka za vse $q \in Q$. Za vsaki dve poti torej velja, da se sicer začneta v različnih sobanah, prej ali slej pa se v nekem trenutku, po nekem številu korakov (seveda obe po enakem številu korakov!), obe znajdeta v isti sobani in se odtlej ne ločita več.

Primerno navodilo lahko sestavimo postopoma: na začetku imamo n poti, ki so vsaka v svoji sobani; nato pa na vsakem koraku izberimo dve od teh poti in razmislimo, kako moramo podaljšati navodilo s še nekaj novimi koraki, da se bosta tidve poti znašli v isti sobani; ta podaljšek navodila uporabimo potem tudi na vseh ostalih poteh; zdaj sta torej vsaj dve od naših poti v isti sobani in se odtlej ne bosta več ločili, zato lahko eno zavržemo. To ponavljamo, dokler nam ne ostane ena sama pot; takrat vemo, da bi bile tudi vse ostale poti, ki smo jih medtem zavrgli, zdaj v isti sobani kot tale edina preostala pot, torej je navodilo, ki se nam je doslej nabralo, prav takšno, kakršno zahteva naloga. Zapišimo ta postopek s psevdokodo:

1 $s :=$ prazno zaporedje (navodilo z 0 koraki); $A := \{1, 2, \ldots, n\}$;
2 **while** $|A| > 1$:
  (* *Na tem mestu velja* $A = \{\delta(q, s) : q \in Q\}$. *)
3 naj bosta u in v poljubni dve sobani iz A;

4 naj bo t poljubno navodilo (zaporedje krtov), za katero je $\delta(u,t) = \delta(v,t)$;
5 dodaj t na konec navodila s;
6 $A := \{\delta(q,t) : q \in A\}$;
7 izpiši s;

V glavni zanki torej vzdržujemo množico soban A, v katerih se trenutno nahaja naših n poti (torej poti, ki se začnejo v vseh možnih začetnih sobanah iz Q in nato sledijo navodilu s). V vsaki iteraciji zanke podaljšamo navodilo s tako, da se vsaj dve od teh poti združita v isti sobani: poti, ki sta bili pred tem podaljšanjem v sobanah u in v, se po podaljšanju znajdeta obe v isti sobani $\delta(u,t)$ oz. $\delta(v,t)$. Enako podaljšanje nato uporabimo na vseh sobanah iz A, da vidimo, kam s tem podaljšanjem pridejo ostale poti (vrstica 6); tako dobimo novo stanje množice A, ki ima vsaj eno sobano manj kot prej, tako da se bo zanka ustavila po največ $n-1$ iteracijah.

Vprašanje je še, kako v vrstici 4 za dani u in v poiskati primerno navodilo t, za katero bo $\delta(u,t) = \delta(v,t)$. Lahko si predstavljamo usmerjen graf, v katerem so točke vse možne množice ene ali dveh soban, torej $V = \{\{p,q\} : p,q \in Q\}$; in iz vsake take točke $\{p,q\}$ naj gre za vsakega krta $a \in \Sigma$ povezava z oznako a v točko $\{r_{ap}, r_{aq}\}$. Vsaka pot v tem grafu ustreza nekemu navodilu (zaporedju krtov) in obratno. V našem primeru iščemo torej pot, ki nas v tem grafu pripelje od $\{u,v\}$ do poljubne točke oblike $\{w,w\}$; če odčitamo oznake na povezavah, ki tvorijo to pot, dobimo ravno navodilo t, kakršno iščemo.

Ker bomo morali to početi po večkrat za različne u in v (namreč v vsaki iteraciji glavne zanke našega postopka po enkrat), je koristno, če pregledamo graf le enkrat namesto večkrat. To lahko naredimo z iskanjem v širino nazaj po grafu (nasproti smeri povezav), pri čemer začnemo v vseh točkah oblike $\{w,w\}$. Tako bomo sčasoma za vsako $\{p,q\} \in V$ dobili njeno oddaljenost (recimo d_{pq}) od njej najbližje točke oblike $\{w,w\}$; zapomnimo pa si tudi, po katerem krtu je šel prvi korak na tej poti (recimo K_{pq}). Če se iskanje v širino ustavi, ne da bi doseglo vse točke grafa, to pomeni, da iz neke $\{p,q\}$ ni dosegljiva nobena točka oblike $\{w,w\}$, torej ne obstaja nobeno navodilo, ki bi nas iz p in q obeh pripeljalo v isto točko w; takrat lahko takoj zaključimo, da je problem nerešljiv.

Če pa iskanje v širino uspešno obišče vse točke grafa, si bomo lahko z vrednostmi K_{pq} pomagali, da bomo v vrstici 4 naše glavne zanke sestavili primerno navodilo t, ki nas bo iz soban u in v pripeljalo v neko sobano w (in to celo po najkrajši poti). Vrstico 4 lahko torej podrobneje zapišemo takole:

$t :=$ prazno zaporedje;
while $u \neq v$:
  $a := K_{uv}$; dodaj a na konec t;
  $u := r_{au}$; $v := r_{av}$;

V vrstici 3 smo prvotno napisali, da sobani $u, v \in A$ izberemo poljubno; ne škodi pa, če izberemo tisti dve, ki imata najmanjšo vrednost d_{uv} — tako bomo navodilo s v trenutni iteraciji glavne zanke kar najmanj podaljšali.

Kako dolgo navodilo s na ta način dobimo? Glavna zanka izvede največ $n-1$ iteracij in v vsaki podaljša navodilo za d_{uv} korakov, pri čemer je d_{uv} dolžina neke najkrajše poti v grafu z $|V| = n(n+1)/2$ točkami. Taka najkrajša pot je gotovo krajša od $|V|$ korakov (saj bi se sicer točke na poti začele ponavljati, torej bi tvorile cikel in bi lahko pot skrajšali, če bi ta cikel izrezali). Tako torej vidimo, da bo navodilo s na koncu dolgo kvečjemu $(n-1)n(n+1)/2 \leq n^3/2$ korakov. Naša naloga pravi, da sme biti navodilo dolgo kvečjemu 10^6 korakov, število soban n pa je največ 100; ni nam torej treba skrbeti, da bi to omejitev presegli.

Razmislimo še o časovni zahtevnosti te rešitve. Pri iskanju v širino smo pregledali graf z $O(n^2)$ točkami, iz vsake od njih pa je šlo m izhodnih povezav; to nam torej vzame $O(n^2 m)$ časa. Poleg tega smo v vrstici 6 naše glavne zanke morali za vsako točko iz A izračunati, kam pridemo iz nje, če sledimo navodilu t; ker je t dolžine $O(n^2)$, v A je $O(n)$ točk in ker to vrstico izvedemo $O(n)$-krat, nam to vzame vsega skupaj $O(n^4)$ časa. Časovna zahtevnost celotne rešitve je torej $O(n^3(n+m))$.

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

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

int main()
{
  // Preberimo vhodne podatke.
  int n, m; cin >> n >> m;
  vector<vector<int>> delta(m, vector<int>(n, -1));
  for (int a = 0; a < m; ++a) for (auto &x : delta[a]) { cin >> x; --x; }

  // Za vsako sobo si pripravimo seznam vhodnih povezav.
  // vhodne[u][a] = seznam sob, iz katerih gre povezava krta a v sobo u
  vector<vector<vector<int>>> vhodne(n, vector<vector<int>>(m));
  for (int v = 0; v < n; ++v) for (int a = 0; a < m; ++a)
    vhodne[delta[a][v]][a].emplace_back(v);

  // Za vsak par sob poiščimo najkrajšo pot, po kateri iz obeh pridemo v isto sobo.
  // To bomo naredili z iskanjem v širino po grafu, v katerem vsaka točka
  // predstavlja množico ene ali dveh sob.
  vector<int> d(n * n, -1), nasl(n * n, -1); queue<int> vrsta;
  for (int u = 0; u < n; ++u) {
    int stanje = u * n + u; vrsta.push(stanje); d[stanje] = 0; }
  while (! vrsta.empty()) {
    int stanje = vrsta.front(); vrsta.pop();
    int u = stanje % n, v = stanje / n;
    for (int a = 0; a < m; ++a) for (int uu : vhodne[u][a]) for (int vv : vhodne[v][a]) {
      int stanje2 = min(uu, vv) * n + max(uu, vv);
      if (d[stanje2] >= 0) continue;
      nasl[stanje2] = a; d[stanje2] = d[stanje] + 1; vrsta.push(stanje2); } }

  // Če se iz kakšnih dveh sob ne da priti v isto sobo, je problem nerešljiv.
  for (int u = 1; u < n; ++u) for (int v = 0; v < u; ++v)
    if (d[v * n + u] < 0) { cout << "NE" << endl; return 0; }

  // Sicer sestavimo primerno pot. Začnimo v vseh sobah.
  printf("DA\n");
  vector<int> sobe(n), dosegljiva(n, -1);
  for (int u = 0; u < n; ++u) sobe[u] = u;
  for (int i = 0; sobe.size() > 1; ++i)
  {
    // Dopolnimo pot s takšnim zaporedjem korakov, ki nas iz
    // vsaj dveh trenutno dosegljivih sob pripelje v isto sobo.
    for (int u = sobe[0], v = sobe[1]; u != v; ) {
      int a = nasl[min(u, v) * n + max(u, v)]; cout << a + 1 << " ";
      auto &da = delta[a]; for (auto &w : sobe) w = da[w];
      u = da[u]; v = da[v]; }

    // Pobrišimo duplikate.
    for (int j = 0; j < sobe.size(); )
      if (int w = sobe[j]; dosegljiva[w] != i) dosegljiva[w] = i, ++j;
      else { sobe[j] = sobe.back(); sobe.pop_back(); }
  }

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

Naloge so sestavili: kiralnost — Bor Brudar; boggle — Urban Duh; disleksija, preurejanje, prepisovanje — Tomaž Hočevar; budilka — Vid Kocijan; matrika — Matija Likar; tekstonim — Mark Martinec; odbojkaške točke — Jakob Schrader; rokomet, laserji — Jure Slak; lučke, krti — Patrik Žnidaršič; premešani mozaik, vodoravne daljice — Janez Brank.

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