21. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2026

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.");
   }
}
```

21. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2026

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://ucilnica.acm.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla vas bodo čakala na mizi v učilnici. Pri oddaji preko računalnika odpreš dotično nalogo v spletni učilnici in rešitev natipkaš oz. prilepiš v polje za programsko kodo. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v kakšnem drugem urejevalniku na računalniku (npr. Visual Studio Code) in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

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

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

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

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

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

`https://www.1ka.si/a/dfb2527a`

1. Pangramski stavki

Pangramski stavek je stavek, v katerem so uporabljene vse črke abecede (za potrebe naše naloge je to angleška abeceda s 26 črkami od `a` do `z`).

Napiši program (ali del programa), ki prebere več stavkov in med njimi poišče najkrajši pangramski stavek, to je pangramski stavek z najmanj črkami. Program naj izpiše, kateri izmed vhodnih stavkov je najkrajši pangramski stavek (izpiši zaporedno številko stavka od 1 naprej), oz. izpiše, da ni pangramskega stavka. Če obstaja več enako dolgih najkrajših pangramskih stavkov, je vseeno, katerega od njih najdeš.

Tvoj program lahko podatke bere s standardnega vhoda ali pa iz datoteke `vhod.txt` (karkoli ti je lažje). V prvi vrstici vhodnih podatkov je število stavkov, ki sledijo, nato pa so navedeni stavki. Vsak je v svoji vrstici in je dolg največ 200 znakov. V vhodnih stavkih nastopajo velike in male črke angleške abecede, poleg njih pa tudi presledki, ločila in drugi ne-črkovni znaki, ki ne vplivajo na to, ali je stavek pangramski ali ne. Za potrebe preverjanja, ali je stavek pangramski, je vseeno, ali se neka črka v njem pojavi kot velika ali kot mala.

Primer vhodnih podatkov:

```
3
abcdefghijklmnopqrstuvwxzz
The quick brown fox jumps over the lazy dog.
Pack my box with five dozen liquor jugs.
```

Pripadajoči izhod:

```
3
```

2. WC-kabine

Imaš mikrokontroler, ki nadzoruje luči v več WC-kabinah. Vsaka WC-kabina ima detektor gibanja in detektor, ki zaznava, ali so vrata kabine odprta. Radi bi napisali program, ki bo skrbel, da bo luč v posamezni kabini prižgana, ko je človek v njej.

Problem je, da se ljudje na WCju ne gibljejo vedno dovolj, da bi senzor zaznaval gibanje, ne želimo pa, da se jim luč ugasne, dokler so vrata zaprta. Prav tako ne želimo, da luč gori, če so vrata odprta (ker bi lahko senzor takrat po nesreči ujel gibanje izven kabine).

Napiši dva **podprograma** (oz. funkciji), ki ju bo sistem poklical ob določenih dogodkih, onadva pa morata poskrbeti za prižiganje in ugašanje luči v skladu z navodili iz prejšnjega odstavka:

- Gibanje(k, zacelo) — v kabini k se je gibanje začelo (če je zacelo == 1) oz. končalo (če je zacelo == 0).
- Vrata(k, odprla) — vrata kabine k so se odprla (če je odprla == 1) ali zaprla (če je odprla == 0).

Stanje luči spreminjaš tako, da pokličeš funkcijo Luc(k, prizgi), ki luč v kabini k prižge (če je prizgi == 1) oz. ugasne (če je prizgi == 0).

Za število kabin lahko predpostaviš, da je podano v neki globalni spremenljivki oz. konstanti n. Tvoja rešitev lahko definira tudi svoje dodatne globalne spremenljivke, ki jih lahko tudi po svoje inicializiraš. Številka kabine k je pri vseh funkcijah celo število od 1 do n. Predpostaviš lahko, da so na začetku izvajanja vsa vrata zaprta, luči ugasnjene in da noben detektor ne zaznava gibanja.

3. Zaporedje števk

Dano je zaporedje števk $a_1, \ldots, a_n$ (n je vsaj 2), vsaka od njih pa ima eno izmed vrednosti $\{0, 1, \ldots, 9\}$. Med vsaki dve zaporedni števki moramo vriniti enega od operatorjev $+$ ali $\cdot$, to pa bi radi storili tako, da bo vrednost dobljenega izraza na koncu čim manjša. **Opiši postopek** (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki to naredi, torej za dano zaporedje števk ugotovi, kako je treba mednje vriniti operatorje. Če je možnih več enako dobrih rešitev, je vseeno, katero od njih tvoj postopek poišče. Tvoj postopek naj bo učinkovit, da bo deloval hitro tudi za velike n.

Upoštevaj, da pri določanju vrednosti izraza operator $\cdot$ veže močneje kot $+$ in da lahko vrivaš le tadva operatorja, ne pa na primer oklepajev ali česa podobnega. Dobro tudi **utemelji**, zakaj je tvoj postopek res pravilen.

Primer: recimo, da dobimo zaporedje števk $[4, 1, 2]$. Z vrivanjem operatorjev lahko dobimo izraze $4 + 1 + 2$, $4 + 1 \cdot 2$, $4 \cdot 1 + 2$ in $4 \cdot 1 \cdot 2$. Najmanjša možna vrednost, ki jo lahko dosežemo, je torej 6, ki jo dobimo pri $4 + 1 \cdot 2$ ali pri $4 \cdot 1 + 2$.

4. Prelet gorovja

Lan je strasten letalec in si je zadal nov izziv: čim nižje želi na konstantni višini leteti čez gorske verige v okolici Bleda. Tu te prosi za pomoč, saj mu računanje ne gre najbolje od rok, sploh ko se vključi še tretjo dimenzijo.

Lan bo vedno letel od juga proti severu (na zemljevidu od spodaj navzgor) in po začetku preleta ne bo spreminjal višine.

Napiši program, ki bo s pomočjo višinskega zemljevida izrisal, na kateri višini mora Lan leteti, da ne bo posnel kakšnega gorskega vrha.

Vhodni podatki: vhod je tabela števil $a_{i,j}$ velikosti $h \times w$, ki jo lahko dobiš v sebi najljubši obliki (npr. na standardnem vhodu s številoma h in w v prvi vrstici ter mrežo števil v sledečih h vrsticah).

Omejitve vhodnih podatkov: $0 \leq a_{i,j} \leq 200$; $1 \leq h \leq 10\,000$; $1 \leq w \leq 200$.

Izhodni podatki: tvoj program naj izriše, kje Lan lahko leti in kje ne. V i-ti vrstici (če jih štejemo od najnižje navzgor) j-tega stolpca naj bo pika („.“), če lahko leti na višini i po j-tem stolpcu vhodnega zemljevida, in lojtra oz. hash („#“), če bi se tam zaletel v kakšno goro. Izpiši najmanjše možno število vrstic, pri katerem bodo v vrhnji same pike (in kjer najnižja vrstica predstavlja višino 1).

Primer vhoda:

```
3 10
2 3 2 5 7 3 8 3 2 2
1 2 3 6 5 6 8 2 1 4
1 4 1 8 4 5 6 4 2 5
```

Pripadajoči izhod:

```
..........
...#..#...
...##.#...
...####...
...####..#
.#.#####.#
.#######.#
##########
##########
```

5. Merilec elektrike

Elektropodjetje od uporabnikov brez naprednih merilcev porabe energije zahteva, da stanje svojega števca vsak mesec ročno vnesejo v spletni obrazec.

Vsak mesec uporabnik vnese število kilovatnih ur na svojem števcu, sistem pa mora izračunati, koliko energije je ta uporabnik porabil ta mesec. Žal imajo števci zgolj pet števk. Ko pridejo do 100 000 kilovatnih ur, se resetirajo nazaj na 0 in štejejo od tam naprej.

Od n uporabnikov smo dobili meritve števca na koncu prejšnjega in na koncu trenutnega meseca. Nekdo je meritve prejšnjega meseca uredil naraščajoče po velikosti, prav tako pa tudi meritve trenutnega meseca. Tako zdaj ne vemo več, katera od meritev prejšnjega meseca in katera od meritev trenutnega meseca se nanašata na istega uporabnika. Zato ne moremo več za vsakega uporabnika izračunati, koliko elektrike je porabil, radi pa bi vsaj ugotovili (najvišjo) spodnjo mejo, za katero je mogoče z gotovostjo trditi, da je vsak uporabnik porabil vsaj toliko elektrike.

Opiši postopek (ali napiši program ali podprogram oz. funkcijo), ki kot vhodne podatke dobi število uporabnikov n ter dva seznama n meritev (enega s konca prejšnjega meseca in enega s konca trenutnega meseca), ki sta vsak zase urejena naraščajoče. Tvoj postopek naj izračuna ali izpiše, najmanj koliko elektrike je zagotovo porabil vsak uporabnik.

Predpostaviš lahko, da je $n \geq 1$, zaželeno pa je, da je tvoj postopek učinkovit, da bo deloval hitro tudi za velike n.

Primer: če imamo $n = 4$ uporabnike in smo s konca prejšnjega meseca dobili meritve $[10\,000, 20\,000, 89\,000, 90\,000]$, s konca trenutnega meseca pa $[5000, 9000, 36\,000, 45\,678]$, lahko zaključimo, da je vsak uporabnik zagotovo porabil vsaj 15 000 kilovatnih ur elektrike.

21. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2026

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-2026-2/` 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/dfb2527a`

1. Palačinke

Dana je skladovnica n palačink različnih velikosti, katerih premeri so predstavljeni kot seznam števil $(a_1, \ldots, a_n)$, ki si sledijo od vrha skladovnice proti dnu. Radi bi jih uredili naraščajoče po velikosti (torej tako, da bo na koncu veljalo $a_1 \leq a_2 \leq \cdots \leq a_n$), pri čemer smemo seznam spreminjati le tako, da si na vsakem koraku izberemo neko število k in obrnemo zgornjih k palačink na seznamu:

$$(a_1, a_2, \ldots, a_{k-1}, a_k, a_{k+1}, \ldots, a_n) \to (a_k, a_{k-1}, \ldots, a_2, a_1, a_{k+1}, \ldots, a_n).$$

Napiši program (ali podprogram oz. funkcijo), ki za dano začetno stanje skladovnice izpiše zaporedje teh operacij, s katerim skladovnica postane urejena naraščajoče (za vsako operacijo je torej treba izpisati, koliko palačink se pri njej obrne). Ni treba iskati najkrajšega možnega zaporedja, ne sme pa biti daljše od $2n$ korakov. Podrobnosti tega, v kakšni obliki tvoj (pod)program dobi vhodne podatke in izpiše rezultate, si izberi sam in jih v svoji rešitvi tudi opiši.

Primer: če imamo $n = 4$ palačinke in kot vhod dobimo zaporedje $(5, 2, 3, 5)$, ga lahko uredimo na primer v dveh korakih tako, da v prvem koraku obrnemo zgornje tri palačinke, v drugem koraku pa zgornji dve.

2. ChordPro v2

ChordPro je format, v katerem lahko pišemo besedila in akorde za pesmarice. Pesem v tem formatu je videti kot vsaka druga, le da ima na želenih mestih vstavljene akorde (ki kitaristu povedo, kaj naj igra). Akord je niz znakov znotraj oglatih oklepajev (ti se drugače v besedilu pesmi ne pojavljajo), ki je vstavljen neposredno pred del besedila, na katerega se nanaša.

Pesmi imajo ponavadi enako melodijo za večino kitic, besedilo pa se spreminja. Kitaristi začetniki imajo zato pogosto težave, saj bi morali načeloma slediti besedilu v naslednjih kiticah, akordi pa so običajno napisani samo nad prvo kitico.

Napiši program, ki bo akorde iz prve kitice dodal še ostalim kiticam. Da tvojemu programu ne bo treba šteti zlogov (kar je vse prej kot enostavno), so mesta, kamor je treba vriniti akorde, že označena s praznimi oglatimi oklepaji „`[]`“.

Vhodni podatki: v prvi vrstici dobi tvoj program število n (velja $1 \leq n \leq 200$), sledi pa n vrstic z besedilom pesmi. Posamezna vrstica je dolga največ 200 znakov. Vsaki dve zaporedni kitici sta ločeni z natanko eno prazno vrstico. Vse kitice imajo enako število akordov, lahko pa imajo različno število verzov in tudi akordi so lahko različno razporejeni po verzih. Akordi so zapisani le v prvi kitici, v ostalih kiticah pa nastopajo le prazni pari oglatih oklepajev „`[]`“.

Izhodni podatki: v n vrstic izhoda izpiši pesem z dodanimi akordi v ne-prve kitice. Akordi naj bodo tudi na izhodu označeni z oglatimi oklepaji `[` in `]`.

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

Primer vhoda:

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

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

[]Ko jim sanje []zadisijo []bele muce []brz zaspijo.
[]In za njimi tudi črne []in se zlate []in srebrne[].
```

(*Nadaljevanje na naslednji strani.*)

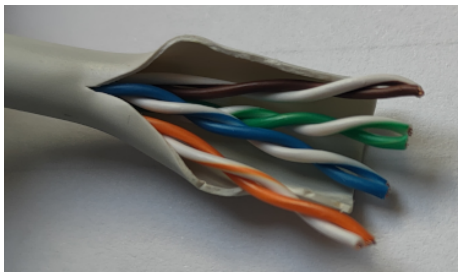
Pripadajoči izhod:

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

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

[Am]Ko jim sanje [E]zadisijo [Am]bele muce [C]brz zaspijo.
[Dm]In za njimi tudi črne [Esus4]in se zlate [E]in srebrne[Am].
```

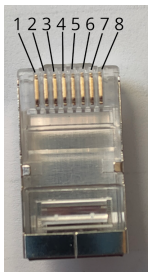
3. Parica

Parica je v osnovi dvojna izolirana prepletena žica, ki je nastala za potrebe telefonskih komunikacij. Dandanes se z njo najpogosteje srečamo, ko želimo položiti kakšen ethernetni kabel.

V standardnem kablu UTP je osem žic v štirih parih, kjer se žici v paru ovijata druga okoli druge. Vsak par ima eno od štirih barv (rjavo, zeleno, modro, oranžno). Ena od žic ima izolacijo samo te barve, druga pa je bela s tobarvno črto (glej prvo sliko, kjer se to lepo vidi pri belo-oranžni žici).



Slika 1



Slika 2

Slika 3

Te žice se nato poveže v konektor RJ-45 (navadni ethernetni konektor; glej drugo sliko), ki ima osem kontaktov. V glavnem se uporablja dve različni shemi (vrstna reda žic v konektorju), prikazani v spodnji tabeli:

Kontakt	T568A	T568B
1.	belo-zelena	belo-oranžna
2.	zelena	oranžna
3.	belo-oranžna	belo-zelena
4.	modra	modra
5.	belo-modra	belo-modra
6.	oranžna	zelena
7.	belo-rjava	belo-rjava
8.	rjava	rjava

Na tretji sliki s povezanim konektorjem so žice na konektor povezane po shemi T568B.

Po standardu ANSI/TIA-568 so veljavna naslednja povezovanja konektorjev na obeh koncih kabla:

1. Na obeh straneh sta konektorja povezana po shemi T568A.
2. Na obeh straneh sta konektorja povezana po shemi T568B.
3. Na eni strani je konektor povezan po shemi T568A, na drugi pa po shemi T568B.

Ko je Janezek zadnjič brskal po podstrešju pri svojem dedku, pa je našel razne čudno povezane kable, ki se včasih tudi ne držijo široko uveljavljenih standardov.

Napiši program, ki prebere, kako sta povezana oba konektorja na koncih kabla, in sporoči, ali bo kabel deloval ali ne.

Da bo kabel deloval, sta pomembna dva pogoja:

1. To, kateri kontakt na prvi strani je povezan s katerim kontaktom na drugi strani, se ujema z enim od treh zgoraj omenjenih standardnih povezovanj. (Kontakt i na prvi strani je povezan s kontaktom j na drugi strani, če sta oba povezana na isto žico.) Tako sta v prvem tipu standardnega povezovanja povezana prva kontakta na obeh straneh (saj sta oba povezana na belo-zeleno žico).

2. Na vsaki strani kabla je vsak par žic, ki se ovijata ena okoli druge, povezan na tak par kontaktov, na katerega sta tudi v shemah T568A in T568B povezani neki taki žici, ki se ovijata ena okoli druge. Na primer, eden od takih parov kontaktov sta 7. in 8. kontakt (ki si v shemah T568A in T568B delita rjavi par žic — sedmi kontakt je povezan na belo-rjavo, osmi pa na rjavo žico). To pomeni, da si morata (za delujoč kabel) 7. in 8. kontakt deliti en par žic (ne pa nujno ravno rjavi par).

Vhodni podatki: na vhodu dobi tvoj program dve vrstici, ki opisujeta vsaka en konec kabla; v vsaki od teh dveh vrstic je po osem oznak za barve, ki povedo, v kakšnem vrstnem redu so žice povezane na kontakte konektorja. Vsaka barva je označena s prvo črko svojega imena (`r`, `z`, `m`, `o`), belo-barvne žice pa z `b` in črko barve. Predpostaviš lahko, da se v vsaki vrstici vsaka barva pojavi natanko enkrat.

Izhodni podatki: izpiši „`Bo deloval`“, če bo kabel deloval, in „`Ne bo deloval`“, če kabel ne bo deloval (ker krši enega od zgornjih dveh pogojev).

Pet primerov vhoda in izhoda:

Vhod 1:

```
bz z bo m bm o br r
bo o bz m bm z br r
```

Izhod 1:

```
Bo deloval
```

Vhod 2:

```
bz z bo m bm o br r
bz z bo m bm o br r
```

Izhod 2:

```
Bo deloval
```

Vhod 3:

```
bz z bo m bm o br r
bz z bo m br o bm r
```

Izhod 3:

```
Ne bo deloval
```

Vhod 4:

```
br z bo m bm o bz r
br z bo m bm o bz r
```

Izhod 4:

```
Ne bo deloval
```

Vhod 5:

```
bm m bo z bz o r br
bo o bm z bz m r br
```

Izhod 5:

```
Bo deloval
```

Komentar. V prvem primeru je prvi konektor povezan s T568A, drugi pa s T568B. V drugem primeru sta oba povezana s T568A.

V tretjem primeru je na eni strani sedmi kontakt povezan s petim na drugi (kar ni pravilno, saj mora biti sedmi kontakt na eni strani vedno povezan z sedmim kontaktom na drugi strani).

V četrtem primeru so kontakti sicer pravilno povezani (i-ti na eni strani z i-tim na drugi strani), sta pa prvi in drugi kontakt (ki bi morala imeti ovijajoč se par žic) povezana z žicama različnih barv, ki se torej ne ovijata in zato ne bosta mogli zmanjševati šuma.

V petem primeru so kontakti pravilno povezani, čeprav konektorja nista niti po shemi T568A niti po T568B.

4. Diagram

V Ljubljani bodo vzpostavili podzemno železnico v smeri od zahoda proti vzhodu. Linije so že izbrane in opisane s pari črk, kjer velika in mala črka označujeta začetek in konec iste linije, med njimi pa je lahko tudi različno število presledkov. Primer opisa: `"XB A  aDd bC  cx"`. Linije so načrtovane tako, da se med seboj ne sekajo, ampak so lepo „vgnezdene“ ena v drugi ali pa si sledijo ena za drugo. **Napiši program** (ali podprogram oz. funkcijo), ki prebere niz z opisom linij (ali ga dobi kot parameter) in izpiše oz. izriše čim bolj kompakten (čim nižji) zemljevid linij v sledeči obliki:

```
/------------------\
|/------------\     |
|| /------\/\ |/---\|
XB A      aDd bC   cx
```

V spodnji vrstici izpisa mora biti torej ravno vhodni niz z opisom linij, nad njim pa mora biti vsak par črk, ki označujeta začetek in konec iste linije, povezan s črto, sestavljeno iz znakov `|`, `/`, `-` in `\`. Te črte se ne smejo prekrivati in naj ne bodo višje, kot je neobhodno potrebno.

Podrobnosti tega, v kakšni obliki tvoj (pod)program dobi vhodni niz z opisom linij in kam izpiše oz. izriše zemljevid, si izberi sam in jih v svoji rešitvi tudi opiši. Predpostaviš lahko, da je vhodni niz dolg največ 200 znakov, v njem pa nastopajo le črke angleške abecede in presledki. Nobeni dve liniji ne uporabljata iste črke.

5. Alkoholni test

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 dve sekundi in sme porabiti največ 256 MB pomnilnika.

Naloga ima 10 testnih primerov, vsak je vreden po dve točki. 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.

Imamo sliko spojin iz atomov C, O in H. Vsak C ima štiri vezi, vsak O dve in vsak H eno, pri čemer so vse vezi enojne. Posamezna spojina ima kvečjemu en C. Nekatere spojine vsebujejo zaporedje C–O–H; za potrebe naše naloge bomo te (in samo te) spojine šteli za alkohole. **Napiši progam**, ki prešteje alkohole na sliki.

Vhodni podatki: prva vrstica vsebuje celi števili n in m (velja $1 \leq n \leq 1000$ in $1 \leq m \leq 1000$). Sledi n vrstic s po m znaki, ki skupaj predstavljajo sliko. Na sliki so atomi označeni s črkami `C`, `O` in `H`, vezi so predstavljene z znaki „`-`“ in „`|`“, pike pa označujejo prazen prostor. Znak „`-`“ se vedno na levi in desni dotika dveh atomov in označuje vez med njima, znak „`|`“ pa se dotika atomov nad in pod njim in prav tako označuje vez med njima. Spojine na sliki se ne prekrivajo (noben znak ne pripada več kot eni spojini) in vsaka spojina je vidna v celoti.

Izhodni podatki: tvoj program naj izpiše eno samo celo število, in sicer število alkoholov na vhodni sliki.

Primer vhoda:

```
6 21
H-H....O-H......H....
..O-O-H|....H...|....
..|..H-C-H..|.H-C-O-H
H-C-H.H|H-O-C-H.|.O-H
..|...|O-H..|...H.|..
..H...H.....H.....H..
```

Pripadajoči izhod:

```
3
```

Komentar: na gornji sliki je sedem spojin, od tega so trije alkoholi (pri enem od teh je njegov atom C udeležen celo v dveh verigah C–O–H hkrati).

Pozor: tvoj program mora seveda delovati za poljuben vhod, ki je skladen z zgoraj opisanimi omejitvami, ne le npr. za tu prikazani primer.

21. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2026

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše. Programi naj berejo vhodne podatke s standardnega vhoda in izpisujejo svoje rezultate na standardni izhod. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih podatkov. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih podatkov, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih podatkov.

Tvoji programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java, python ali rust.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2026-3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi svoje rešitve nalog. Pred začetkom tekmovanja lahko poskusiš oddati katero od nalog iz arhiva `https://putka-rtk.acm.si/tasks/s/test-sistema/list/`. Uporabniško ime in geslo za Putko sta enaki kot za računalnike. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava pod „Pogovor o nalogi“ v okvirju „Osnovne informacije“ desno od besedila posamezne naloge).

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih. Za vsak testni primer se bo izpisalo, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal predolgo ali pa porabil preveč pomnilnika (točne omejitve so navedene na ocenjevalnem sistemu pri besedilu vsake naloge), ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika (za podrobnosti o tem, katere prevajalnike uporablja ocenjevalni strežnik in s kakšnimi nastavitvami, glej stran `https://putka-rtk.acm.si/info/`). Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku in na ocenjevalnem strežniku), prenosnih računalnikov, prenosnih telefonov itd.

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

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

`https://www.1ka.si/a/dfb2527a`

Ocenjevanje

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

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

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

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

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

Primer naloge (ne šteje k tekmovanju)

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

Primer vhoda:

```
123 456
```

Ustrezen izhod:

```
5790
```

Primeri rešitev:

- V pascalu:

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

- V C++:

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

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

- V javi:

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

- V C-ju:

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

- V pythonu:

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

- V C#:

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

- V rustu:

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

21. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2026

NALOGE ZA TRETJO SKUPINO

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

1. Giganti

Giganti so se uprli Zevsu in ostalim bogovom, a ker bogovi prebivajo na Olimpu, so se morali za spopad z njimi najprej povzpeti nanj. Legenda pravi, da so za vzpon uporabili dve gori z višinama a in b metrov. V eni potezi so Giganti izmed dveh gora izbrali eno, ki se je magično podvojila, in so to njeno kopijo postavili na drugo goro, ki je s tem postala višja. (Tako na primer iz gora $\{3, 8\}$ lahko nastaneta $\{3, 11\}$ ali pa $\{8, 11\}$.) Ta postopek so ponavljali, dokler nista bili gori veliki natanko c in d metrov, torej ena c, druga pa d metrov (pri tem je vseeno, katera je katera), in so tako lahko napadli bogove.

Napiši program, ki za več četveric (a, b, c, d) ugotovi minimalno število operacij, da lahko Giganti napadejo bogove (ali pa ugotovi, da napad na bogove ni mogoč).

Vhodni podatki: prva vrstica vsebuje število poizvedb q. Sledi q vrstic, ki opisujejo posamezne poizvedbe; vsaka od teh vrstic vsebuje po štiri cela števila a, b, c, d, ločena s po enim presledkom.

Izhodni podatki: izpiši q vrstic, za vsako poizvedbo po eno. V i-ti vrstici izpiši eno samo celo število — odgovor na i-to poizvedbo, torej najmanjše število operacij, da lahko Giganti napadejo bogove, če so začeli z gorama višine a in b, na koncu pa hočejo imeti gori višine c in d. Če je napad na bogove nemogoč, izpiši -1.

Omejitve vhodnih podatkov: število poizvedb q je vsaj 1 in kvečjemu 100 000. Pri vsaki poizvedbi so števila a, b, c, d večja ali enaka 1 ter manjša ali enaka 10^{18}.

Naloga je razdeljena na štiri podnaloge, ki se ločijo po dodatnih omejitvah:

- 1. podnaloga (25 točk): $q \leq 1000$ in $\max\{c, d\} \leq 20 \cdot \min\{a, b\}$
- 2. podnaloga (25 točk): $q \leq 1000$ in $c \leq 10\,000$ in $d \leq 10\,000$ in napad na bogove je vedno mogoč.
- 3. podnaloga (25 točk): $q \leq 10$ in $c \leq 10^6$ in $d \leq 10^6$
- 4. podnaloga (25 točk): brez dodatnih omejitev.

Tvoj program dobi vse točke za posamezno podnalogo, če pravilno reši vse njene testne primere, sicer pa pri tisti podnalogi ne dobi nobenih točk.

Primer vhoda:

```
6
3 10 16 19
7 10 10 7
2 1 99 100
3 5 7 13
35 37 2864 8849
1 2 13 21
```

Pripadajoči izhod:

```
3
0
98
-1
20
5
```

2. Multiprocesiranje

Podan imamo seznam n nalog, ki jih moramo izvesti na p procesorjih. Vsako nalogo bomo dodelili enemu od procesorjev (pri tem lahko posamezni procesor dobi tudi več nalog). Naloge so oštevilčene od 1 do n po naraščajoči pomembnosti, procesorji pa so oštevilčeni od 1 do p po naraščajoči moči. Vsaka naloga ima določen interval procesorjev, na katerih se lahko izvaja: naloga i se lahko izvaja le na procesorjih $a_i, a_i + 1, \ldots, b_i - 1, b_i$. Poleg tega imamo dodatno zahtevo, da se morajo pomembnejše naloge izvajati na močnejših procesorjih; z drugimi besedami, če nalogo i izvajamo na procesorju r_i, nalogo j pa na procesorju r_j in če velja $i < j$, potem mora veljati tudi $r_i \leq r_j$.

Napiši program, ki izračuna, koliko je veljavnih razporeditev nalog na procesorje. Pravzaprav, ker je lahko število takšnih razporeditev zelo veliko, nas zanima le ostanek po deljenju tega števila z $10^9 + 7$.

Vhodni podatki: v prvi vrstici sta celi števili n (število nalog) in p (število procesorjev), ločeni s presledkom. Sledi n vrstic, od katerih i-ta vsebuje celi števili a_i in b_i, ločeni s presledkom; tidve števili povesta, da se lahko naloga i izvaja na procesorjih od vključno a_i do vključno b_i (na ostalih pa ne).

Omejitve: $1 \leq n \leq 10\,000$; $1 \leq p$; $n \cdot p \leq 10^7$; za vsak i velja $1 \leq a_i \leq b_i \leq p$.

Pri prvih 50 % testnih primerov bo veljalo tudi $n \leq 100$ in $p \leq 100$.

Izhodni podatki: izpiši eno samo celo število, namreč ostanek po deljenju števila veljavnih razporedov nalog na procesorje z $10^9 + 7$.

Primer vhoda:	Pripadajoči izhod:
3 6 4 4 5 6 2 6	3

Še en primer vhoda:	Pripadajoči izhod:
6 10 2 5 1 3 1 4 6 8 8 9 1 10	105

Komentar: pri prvem primeru se prva naloga lahko izvaja samo na četrtem procesorju. Če se druga naloga izvaja na petem procesorju, se lahko tretja na petem ali šestem. Če pa se druga naloga izvaja na šestem procesorju, se mora tretja prav tako na šestem. Tako dobimo tri možne razporede.

3. Tabela ničel

Dana je pravokotna tabela, široka w stolpcev in visoka h vrstic. V vsaki celici tabele je lahko bodisi vrednost 0 bodisi vrednost 1. Na začetku so v vseh celicah ničle, nato pa smemo tabelo spreminjati le z naslednjima dvema operacijama:

- lahko si izberemo neko vrstico in postavimo vse vrednosti v njej na 1;
- ali pa si izberemo neki stolpec in postavimo vse vrednosti v njem na 0.

Napiši program, ki ugotovi, kako je mogoče na ta način priti do nekega danega končnega stanja tabele.

Vhodni podatki. V prvi vrstici sta celi števili w (širina tabele) in h (višina tabele), ločeni s presledkom, temu pa sledi h vrstic, ki podajajo vsebino tabele; vsaka od teh vrstic vsebuje po w znakov, vsak od teh znakov pa je bodisi `0` bodisi `1` (med njimi ni presledkov).

Omejitve: $1 \leq w \leq 10^6$, $1 \leq h \leq 10^6$, $w \cdot h \leq 9 \cdot 10^6$. Vhodne tabele bodo vedno take, da rešitev obstaja.

Pri prvih 15 % testnih primerov si bodo vse vrstice vhodne tabele med seboj enake.

Pri drugih 15 % testnih primerov bo problem rešljiv tako, da najprej delaš le operacije na vrsticah, nato pa le na stolpcih.

Pri naslednjih 30 % testnih primerov bo veljalo tudi $w \leq 100$ in $h \leq 100$.

Izhodni podatki: v prvo vrstico izpiši število potez, recimo p, nato pa izpiši poteze, vsako v svojo vrstico. Posamezno potezo opiši z dvema številoma, recimo a_i in b_i, pri čemer a_i pove, ali ta poteza postavlja neko vrstico na enice ($a_i = 1$) ali neki stolpec na ničle ($a_i = 0$), število b_i pa pove, za katero vrstico ali stolpec gre (vrstice so oštevilčene od 1 do h, stolpci pa od 1 do w). Število potez p mora biti $\leq 2 \cdot 10^6$. Če je možnih več rešitev, je vseeno, katero od njih izpišeš.

Primer vhoda:

```
5 3
00000
10101
10111
```

Eden od možnih pripadajočih izhodov:

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

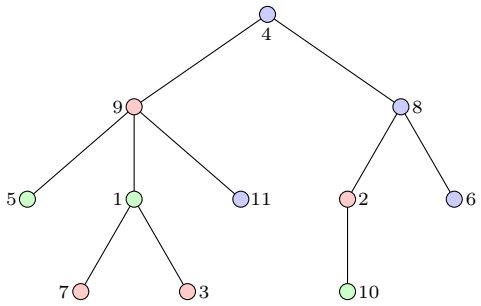
4. Lepe podmnožice

Dano je drevo z n vozlišči, oštevilčenimi od 1 do n. Vsako vozlišče je pobarvano z eno od b možnih barv (barve so predstavljene kar z naravnimi števili od 1 do b). Rekli bomo, da je podmnožica vozlišč $A \subseteq \{1, 2, \ldots, n\}$ *lepa*, če:

- so vsa vozlišča iz A enake barve in
- obstaja neka taka pot po drevesu, ki nobenega vozlišča ne obišče več kot enkrat in ki obišče vsa vozlišča iz A (sme pa obiskati tudi kakšna vozlišča zunaj A).

Napiši program, ki za vsako barvo poišče velikost največje lepe podmnožice, ki vsebuje vozlišča tiste barve.

Primer drevesa z $n = 11$ vozlišči in $b = 3$ barvami:



Vhodni podatki: v prvi vrstici sta celi števili n (število vozlišč) in b (število barv), ločeni s presledkom. Nadaljevanje je odvisno od tega, kolikšen je n:

- Če je $n \leq 200\,000$ (kar velja pri prvih 80 % testnih primerov), sledi prvi vrstici (tisti z n-jem in b-jem) še n vrstic, od katerih i-ta vsebuje celi števili p_i in c_i, ki povesta, da je vozlišče i barve c_i in ima starša p_i. Če je i koren drevesa in starša torej nima, bo $p_i = 0$. (Na gornji sliki je na primer vozlišče 9 starš vozlišč 1, 5 in 11; koren drevesa pa je vozlišče 4.)
- Če pa je $n > 200\,000$ (kar velja le pri zadnjih 20 % testnih primerov), sledi prvi vrstici le še ena vrstica, ta pa vsebuje sedem števil, ločenih s po enim presledkom: A, B, M, K, A', B' in M'. To so cela števila od 1 do 10^9. Ta števila nam določajo dve zaporedji, eno po formulah $r_0 = 0$ in $r_{i+1} = (A \cdot r_i + B) \bmod M$, drugo pa po formulah $r'_0 = 0$ in $r'_{i+1} = (A' \cdot r'_i + B') \bmod M'$. Takšni vhodni podatki predstavljajo drevo z naslednjimi vrednostmi: koren je vozlišče 1 (torej $p_1 = 0$); za vsak $i = 2, \ldots, n$ ima vozlišče i starša $p_i = i - 1 - (r_i \bmod \min(K, i-1))$; za vsak $i = 1, \ldots, n$ ima vozlišče i barvo $c_i = 1 + (r'_i \bmod b)$. (Namen teh formul je, da ti lahko podamo veliko in približno naključno drevo, ne da bi ti bilo treba brati ogromno vhodnih podatkov.)

Omejitve: $1 \leq b \leq n \leq 5\,000\,000$; pri vsakem i bo $1 \leq p_i \leq n$ (razen pri korenu, kjer je $p_i = 0$) in $1 \leq c_i \leq b$. Podatki o starših p_i so taki, da povezave med vozlišči in njihovimi starši res tvorijo drevo.

Pri prvih 25 % testnih primerov bo veljalo tudi $n \leq 20$. Pri naslednjih 25 % testnih primerov bo $n \leq 1000$. Pri naslednjih 30 % testnih primerov bo $n \leq 200\,000$.

Izhodni podatki: označimo z ℓ_i velikost največje take lepe podmnožice, ki vsebuje vozlišča barve i. Zahtevani izpis tvojega programa je odvisen od tega, kolikšen je n:

- Če je $n \leq 200\,000$, izpiši b vrstic, v i-to od teh (za vsak $i = 1, \ldots, b$) pa izpiši le število ℓ_i (velikost največje take lepe podmnožice, ki vsebuje vozlišča barve i).
- Če pa je $n > 200\,000$, izpiši eno samo vrstico, vanjo pa izpiši vsoto $\ell_1 + \ell_2 + \ldots + \ell_b$. (Namen te zahteve je, da pri velikih testnih primerih tvojemu programu ni treba izpisovati ogromno izhodnih podatkov.)

(*Nadaljevanje na naslednji strani.*)

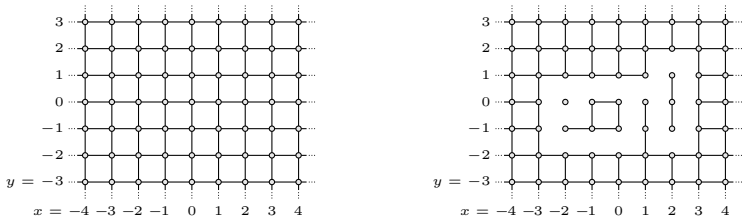
Primer vhoda:	Pripadajoči izhod:
11 3 9 2 8 1 1 1 0 3 9 2 8 3 1 1 4 3 4 1 2 2 9 3	3 2 4

Še en primer vhoda:	Pripadajoči izhod:
200001 123 7 17 11 2 19 3 13	120010

Komentar: primer na levi ustreza gornji sliki. Za barvo 1 obstaja na primer lepa podmnožica $\{2, 7, 9\}$, ki jo lahko obiščemo s potjo $2 \to 8 \to 4 \to 9 \to 1 \to 7$.

5. Otoki

Dana je neskončna karirasta mreža, v kateri je za vsaki celi števili x in y prisotna točka (x, y), ki je povezana s štirimi drugimi točkami, $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$ in $(x, y - 1)$. Nato iz te mreže odstranimo n povezav; za vsak $i = 1, 2, \ldots, n$ pobrišemo povezavo med točkama $(x_{i,1}, y_{i,1})$ in $(x_{i,2}, y_{i,2})$. Zaradi tega se lahko zgodi, da mreža zdaj ni več povezana, ampak je sestavljena iz več ločenih delov. **Napiši program**, ki ugotovi, na koliko delov je mreža razpadla.



Primer neskončne mreže na začetku (*levo*) in po brisanju 21 povezav (*desno*).

Vhodni podatki. V prvi vrstici je naravno število n. Sledi še n vrstic; v i-ti izmed teh vrstic so štiri cela števila $(x_{i,1}, y_{i,1}, x_{i,2}, y_{i,2})$, ločena s po enim presledkom, ki opisujejo i-to pobrisano povezavo. Zagotovljeno je, da sta točki $(x_{i,1}, y_{i,1})$ in $(x_{i,2}, y_{i,2})$ v začetni mreži povezani. Vse pobrisane povezave so različne (torej nobena ni navedena več kot enkrat).

Omejitve:

- $1 \leq n \leq 200\,000$
- $-10^{18} \leq x_{i,1}, y_{i,1}, x_{i,2}, y_{i,2} \leq 10^{18}$ za vsak $i = 1, \ldots, n$ ter
- $|x_{i,1} - x_{i,2}| + |y_{i,1} - y_{i,2}| = 1$ za vsak $i = 1, \ldots, n$.

Ta naloga je razdeljena na podnaloge, ki se razlikujejo po dodatnih omejitvah:

- 1. podnaloga (30 točk): vsi $x_{i,j}$ in $y_{i,j}$ so z območja od 0 do 1000
- 2. podnaloga (25 točk): $n \leq 1000$
- 3. podnaloga (20 točk): $n \leq 10\,000$
- 4. podnaloga (25 točk): ni dodatnih omejitev.

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

Izhodni podatki: izpiši število delov, na katere je mreža razpadla po brisanju n povezav iz vhodnih podatkov.

Primer vhoda:

```
21
2 1 2 2
2 1 1 1
2 1 3 1
2 0 3 0
2 -1 3 -1
2 -1 2 -2
2 -1 1 -1
2 0 1 0
1 0 1 1
0 0 0 1
0 0 1 0
0 -1 1 -1
0 -1 0 -2
-1 -1 -1 -2
-2 -1 -2 -2
-2 -1 -3 -1
-2 -1 -2 0
-2 0 -3 0
-2 0 -2 1
-2 0 -1 0
-1 0 -1 1
```

Pripadajoči izhod:

```
4
```

Komentar: to je primer z gornje slike. Mreža je razpadla na štiri dele (od katerih je eden neskončen, eden obsega le eno točko, eden tri točke in eden pet točk).

21. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2026

REŠITVE NALOG ZA PRVO SKUPINO

1. Pangramski stavki

Stavke bomo brali in obdelovali v zanki, pri tem pa vzdrževali dve spremenljivki z indeksom najkrajšega doslej najdenega pangramskega stavka in številom črk v njej (v spodnjem programu sta to najkrajsi in najDolzina).

Pri vsakem stavku gremo s še eno zanko po njegovih znakih, da preštejemo, koliko črk je v njem (ta podatek potrebujemo, ker nas pri tej nalogi zanima pangramski stavek z najmanj črkami) in koliko od teh črk je različnih (če jih ni 26, to pomeni, da stavek sploh ni pangramski). Da preštejemo, koliko različnih črk je v stavku, bomo v tabeli zeVidena za vsako črko abecede hranili podatek o tem, ali smo jo že videli ali ne; če pri neki črki opazimo, da je še nismo videli, povečajmo števec različnih črk (stRazlicnih) in si v tabeli zapomnimo, da smo to črko zdaj videli.

Ko pridemo do konca stavka, lahko torej preverimo, ali je bil pangramski in ali je krajši od najkrajšega doslej znanega pangramskega stavka; če je tako, si zapomnimo njegov indeks in dolžino. Na koncu izpišemo indeks najkrajšega pangramskega stavka oz. izpišemo, da ni bilo nobenega pangramskega stavka (če ima spremenljivka najkrajsi še vedno vrednost 0, na katero smo jo inicializirali na začetku programa).

```
int main()
{
  int n; cin >> n >> ws; // Preberimo število stavkov in znak za konec vrstice.
  int najkrajsi = 0, najDolzina;
  for (int i = 1; i <= n; ++i)
  {
    // Preštejmo, koliko črk je v naslednjem stavku in koliko od njih je različnih.
    int stCrk = 0, stRazlicnih = 0;
    bool zeVidena[26] = {}; // Tabela pove, katere črke smo že videli.
    while (true)
    {
      // Preberimo naslednji znak stavka; ustavimo se po koncu vrstice.
      int c = cin.get();
      if (! cin || c == '\n') break;

      // Črke pretvorimo v indekse 0..25.
      if ('A' <= c && c <= 'Z') c -= 'A';
      else if ('a' <= c && c <= 'z') c -= 'a';
      else continue; // Nečrkovne znake preskočimo.
      ++stCrk; // Povečajmo števec črk v stavku.

      // Če to črko vidimo prvič, povečajmo tudi števec različnih črk.
      if (! zeVidena[c]) zeVidena[c] = true, ++stRazlicnih;
    }
    // Če je to najkrajši pangramski stavek doslej, si ga zapomnimo.
    if (stRazlicnih == 26 && (najkrajsi == 0 || stCrk < najDolzina))
      najkrajsi = i, najDolzina = stCrk;
  }
  if (najkrajsi == 0) cout << "Noben stavek ni bil pangramski." << endl;
  else cout << najkrajsi << endl;
  return 0;
}
```

Gornja rešitev bere vhodne podatke znak po znak, ker pa vnaprej vemo, da stavki niso zelo dolgi, bi jih lahko brali tudi po vrsticah (v C++ bi uporabili na primer funkcijo getline iz standardne knjižnice).

Namesto tabele, kot je zeVidena v gornji rešitvi, bi lahko imeli tudi neko celoštevilsko spremenljivko, ki bi jo uporabljali kot bitno karto in v njej s prižiganjem posameznih bitov označevali, katere črke smo že videli.

Še ena možnost pa je, da črke stavka dodamo v kakšno podatkovno strukturo za predstavitev množice; ker posamezni element v množici ne more nastopiti več kot enkrat, nam bo število elementov v množici na koncu povedalo, koliko različnih črk je bilo v stavku. Ta pristop (ki je sicer za neki konstanten faktor manj učinkovit) uporablja naslednja rešitev v pythonu:

```
n = int(input()) # Preberimo število stavkov.
najkrajsi = 0; najDolzina = 0
for i in range(n):
  stavek = input() # Preberimo naslednji stavek.
  # Poglejmo, ali ima 26 različnih črk.
  stRazlicnih = len(set(c for c in stavek.lower() if c.isalpha()))
  if stRazlicnih < 26: continue # Če jih je manj kot 26, ni pangramski.
  # Preštejmo črke v stavku.
  stCrk = sum(1 for c in stavek if c.isalpha())
  # Če je to najkrajši pangramski stavek doslej, si ga zapomnimo.
  if najkrajsi == 0 or stCrk < najDolzina:
    najkrajsi = i + 1; najDolzina = stCrk
print("Noben stavek ni bil pangramski." if najkrajsi == 0 else najkrajsi)
```

2. WC-kabine

Razmislimo, kaj mora naš program narediti ob posameznih dogodkih, ki jih omenja besedilo naloge. (1) Naloga pravi, da nočemo, da luč gori, ko so vrata odprta; ko se torej vrata odprejo, moramo luč ugasniti, če je bila prižgana. (2) Kaj pa, ko se vrata kabine zaprejo? Če takrat zaznavamo gibanje, to pomeni, da je uporabnik v kabini in moramo luč prižgati; če pa gibanja takrat ne zaznavamo, luči ne bomo prižgali, kajti če bi jo, bi to pomenilo, da bo luč gorela vedno, ko so vrata zaprta, četudi ni nikogar v kabini. (3) Pač pa pazimo na to, da če začnemo zaznavati gibanje takrat, ko so vrata zaprta in luč ugasnjena, je to znak, da je nekdo v kabini in moramo luč prižgati.

Za točki (1) in (2) lahko poskrbimo v podprogramu Vrata, za (3) pa v Gibanje. Opazimo še, da za točko (2) potrebujemo podatek o tem, ali trenutno zaznavamo gibanje ali ne; ta podatek lahko vzdržujemo v neki globalni spremenljivki, ki jo ob vsaki spremembi popravi podprogram Gibanje. Podobno pa za točko (3) potrebujemo podatek o tem ali so vrata trenutno odprta ali zaprta, kar bo v neki globalni spremenljivki vzdrževal podprogram Vrata.

Pazimo še na to, da imamo opravka z več kabinami, zato bomo podatke o stanju vrat in zaznavanju gibanja za posamezne kabine hranili v neki tabeli ali vektorju.

```
const int n = ...;
vector<bool> odprta(n, false), gibanje(n, false);
extern void Luc(int k, int prizgi);

void Gibanje(int k, int zacelo)
{
  // Zapomnimo si novo stanje gibanja.
  gibanje[k] = zacelo;
  // Če so vrata zaprta in začnemo zaznavati gibanje, prižgimo luč.
  if (zacelo && ! odprta[k]) Luc(k, 1);
}

void Vrata(int k, int odprla)
{
  // Zapomnimo si novo stanje vrat.
  odprta[k] = odprla;
  // Če se vrata zaprejo in gibanja ne zaznavamo, naj luč ostane ugasnjena.
  if (! odprla && ! gibanje[k]) return;
```

```
  // Sicer prižgimo ali ugasnimo luč (odvisno od tega, ali so se
  Luc(k, 1 − odprla);                  // vrata zaprla ali odprla).
}
```

Pri gornji rešitvi se lahko zgodi, da poskuša prižgati luč takrat, ko je že prižgana, ali jo ugasniti takrat, ko je že ugasnjena. Če se hočemo takšnim nepotrebnim klicem podprograma Luc izogniti, pa moramo stanje luči vzdrževati v še eni globalni spremenljivki, da lahko pred klicem Luc preverimo, ali je ta klic sploh potreben. Oglejmo si primer takšne rešitve v pythonu; stanje luči, vrat in detektorjev gibanja bomo hranili v slovarjih, kjer kot ključ uporabljamo številko kabine:

```
n = ...
def Luc(k: int, prizgi: int): ...
lucGori = {}; odprta = {}; gibanje = {}

def Luc_(k: int, prizgi: int):
  # Če je luč že v želenem stanju, ne storimo ničesar.
  if prizgi == lucGori.get(k, 0): return

  # Sicer spremenimo stanje luči.
  lucGori[k] = prizgi; Luc(k, prizgi)

def Gibanje(k: int, zacelo: int):
  # Zapomnimo si novo stanje gibanja.
  gibanje[k] = zacelo

  # Če so vrata zaprta in začnemo zaznavati gibanje, prižgimo luč.
  if zacelo and not odprta.get(k, 0): Luc_(k, 1)

def Vrata(k: int, odprla: int):
  # Zapomnimo si novo stanje vrat.
  odprta[k] = odprla

  # Če se vrata zaprejo in gibanja ne zaznavamo, naj luč ostane ugasnjena.
  if not odprla and not gibanje.get(k, 0): return

  # Sicer prižgimo ali ugasnimo luč (odvisno od tega, ali so se
  Luc_(k, 1 − odprla)                  # vrata zaprla ali odprla).
```

3. Zaporedje števk

Če je kakšna od števk a_i enaka 0, je najbolje, če mednje povsod postavimo $\cdot$ in tako dobimo izraz z vrednostjo 0. Izraza z vrednostjo, manjšo od 0, gotovo ni mogoče sestaviti, kajti vse naše števke so večje ali enake 0 in če seštevamo ali množimo dve števili, večji ali enaki 0, je tudi rezultat vedno večji ali enak 0.

Če so vse števke enake 1, je tudi najbolje, če povsod postavimo $\cdot$ in dobimo izraz z vrednostjo 1. Izraza z vrednostjo 0 tu gotovo ni mogoče dobiti, kajti s seštevanjem in množenjem števil, večjih od 0, dobimo vedno tudi rezultat, večji od 0.

Ostane še primer, ko so vse števke večje ali enake 1 in vsaj kakšna je strogo večja od 1. Ker veže $\cdot$ močneje kot $+$, si lahko izraz, ki ga bomo po vrivanju operatorjev $+$ in $\cdot$ dobili, predstavljamo kot vsoto nekaj zmnožkov.

Recimo, da ima neki zmnožek vrednost 1, torej je zmnožek samih enic; in recimo, da ima eden od sosednjih zmnožkov (prejšnji ali naslednji) vrednost p (ki je ≥ 1); če bi zdaj znak $+$ med tema zmnožkoma spremenili v $\cdot$, bi se oba skupaj združila v en sam zmnožek, ki bi k vrednosti izraza prispeval $1 \cdot p = p$, medtem ko sta prej oba zmnožka skupaj prispevala $1 + p$. Tako bi se vrednost izraza zmanjšala, torej izraz pred to spremembo ni imel najmanjše možne vrednosti. Vidimo torej, da v optimalni rešitvi ne bo nobenega zmnožka z vrednostjo 1; vsak zmnožek bo torej vseboval vsaj eno števko, večjo od 1.

Recimo zdaj, da neki zmnožek vsebuje dve ali več števk, večjih od 1; naj bo a_i prva od njih in naj bo p produkt preostalih. Tako a_i kot p sta torej večja ali enaka 2. Trenutno ta zmnožek prispeva k celotnemu izrazu vrednost $a_i \cdot p$, če pa bi znak $\cdot$ za števko a_i spremenili v $+$, bi naš zmnožek razpadel na dva dela, ki bi k celotnemu izrazu prispevala vrednost $a_i + p$. Ta vrednost pa ni nič večja od stare; velja namreč:

$$a_i \cdot p = \tfrac{1}{2}(a_i \cdot p + a_i \cdot p) \geq \tfrac{1}{2}(a_i \cdot 2 + 2 \cdot p) = a_i + p,$$

pri čemer relacija $\geq$ velja zato, ker sta a_i in p oba ≥ 2. Vidimo torej, da v optimalni rešitvi ni nobene koristi od tega, da bi kak zmnožek vseboval dve ali več števk, večjih od 1, kajti rešitev lahko izboljšamo ali je vsaj ne poslabšamo, če tak zmnožek razbijemo na več delov, ki vsebujejo vsak po eno števko, večjo od 1.

Optimalno rešitev (za primer, ko so vse števke ≥ 1 in vsaj ena je > 1) lahko torej dobimo tako, da za vsako števko, večjo od 1, razen za zadnjo, postavimo operator $+$, vsepovsod drugod pa postavimo operator $\cdot$.

Oglejmo si še implementacijo tu opisanega postopka v jeziku C++. Spodnji podprogram pričakuje kot parameter niz števk in vrne daljši niz, v katerem je med števke vrinil tudi znake `'+'` in `'*'`; tako na primer PostaviOperatorje(`"3141121"`) vrne `"3+1*4+1*1*2*1"`.

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

string PostaviOperatorje(const string &s)
{
  int n = (int) s.length();
  // V izhodni niz „t“ skopirajmo števke iz vhodnega niza „s“,
  // med njimi pa pustimo prostor za operatorje.
  string t(2 * n - 1, ' ');
  bool nicla = false; int zadnjaVelika = -1;
  for (int i = 0; i < n; ++i) {
    t[2 * i] = s[i];
    // Zapomnimo si tudi, če je kakšna števka 0.
    if (s[i] == '0') nicla = true;
    // Zapomnimo si indeks zadnje števke, večje od 1.
    else if (s[i] > '1') zadnjaVelika = i; }
  // Če je med števkami ničla ali pa so same enice, vrinimo povsod '*'.
  if (nicla || zadnjaVelika < 0)
    for (int i = 0; i < n - 1; ++i) t[2 * i + 1] = '*';
  else
    // Sicer pa vrinimo '+' za vsako števko, večjo od 1, razen za zadnjo,
    // povsod drugod pa vrinimo '*'.
    for (int i = 0; i < n - 1; ++i)
      t[2 * i + 1] = (s[i] == '1' || i == zadnjaVelika) ? '*' : '+';
  return t;
}
```

In še enaka rešitev v pythonu:

```
def PostaviOperatorje(s: str) -> str:
  # V izhodni niz „t“ skopirajmo števke iz vhodnega niza „s“,
  # med njimi pa pustimo prostor za operatorje.
  n = len(s); t = [' '] * (2 * n - 1)
  nicla = False; zadnjaVelika = -1
  for i in range(n):
    t[2 * i] = s[i]
    # Zapomnimo si tudi, če je kakšna števka 0.
    if s[i] == '0': nicla = True
    # Zapomnimo si indeks zadnje števke, večje od 1.
    elif s[i] > '1': zadnjaVelika = i
  # Če je med števkami ničla ali pa so same enice, vrinimo povsod '*'.
  if nicla or zadnjaVelika < 0:
    for i in range(n - 1): t[2 * i + 1] = '*'
  else:
    # Sicer pa vrinimo '+' za vsako števko, večjo od 1, razen za zadnjo,
    # povsod drugod pa vrinimo '*'.
    for i in range(n - 1):
      t[2 * i + 1] = '*' if s[i] == '1' or i == zadnjaVelika else '+'
  return "".join(t)
```

4. Prelet gorovja

Za to, pri kateri višini lahko letalec preleti neki stolpec, je pomembna le najvišja višina v tistem stolpcu. Za začetek lahko torej z dvema gnezdenima zankama (po vrsticah in po stolpcih) preberemo zemljevid in si v neki tabeli ali vektorju za vsak stolpec zapomnimo največjo višino v njem.

Nato pojdimo še z eno zanko po tej tabeli in si zapomnimo največjo višino celotnega zemljevida; to je koristno, ker nam pove, koliko vrstic moramo izpisati: če je največja višina v, moramo izpisati $v+1$ vrstic, da bo prva (najvišja) izmed njih lahko sestavljena iz samih pik, kot zahteva besedilo naloge.

Po tem smo pripravljeni na izpis višinskega profila. Spet potrebujemo dve gnezdeni zanki, zunanjo po vrsticah in notranjo po stolpcih. Če je v vhodnem zemljevidu največja višina v stolpcu x enaka v, moramo pri izpisu v stolpcu x imeti na dnu stolpca v znakov „#“, vsepovsod nad njimi pa znake „.“. Ko izpisujemo neko vrstico izpisa, lahko na ta način pri vsakem stolpcu preverimo, ali mora imeti v tej vrstici znak „#“ ali „.“. Oglejmo si implementacijo te rešitve v C++:

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

int main()
{
  // Preberimo zemljevid in si zapomnimo največjo višino vsakega stolpca.
  int w, h; cin >> h >> w;
  vector<int> visina(w, 0);
  for (int y = 0; y < h; ++y)
    for (int x = 0; x < w; ++x) {
      int v; cin >> v;
      if (v > visina[x]) visina[x] = v; }
  // Izračunajmo najvišjo višino na celem zemljevidu.
  int najVisina = 0; for (int v : visina) if (v > najVisina) najVisina = v;
  // Izpišimo oz. izrišimo višinski profil.
  for (int y = najVisina; y >= 0; --y) {
    for (int x = 0; x < w; ++x) cout.put(y >= visina[x] ? '.' : '#');
    cout << endl; }
  return 0;
}
```

In še enaka rešitev v pythonu:

```
# Preberimo zemljevid in si zapomnimo največjo višino vsakega stolpca.
h, w = map(int, input().split())
visina = [0] * w
for y in range(h):
  for x, v in enumerate(map(int, input().split())):
    visina[x] = max(visina[x], v)
# Izpišimo oz. izrišimo višinski profil.
for y in range(max(visina), -1, -1):
  print("".join('.' if y >= visina[x] else '#' for x in range(w)))
```

5. Merilec elektrike

Recimo, da imamo s konca prejšnjega meseca meritve $a_1, \ldots, a_n$, s konca trenutnega pa meritve $b_1, \ldots, b_n$. Naloga pravi, da sta oba seznama urejena naraščajoče, zato ne vemo, katera meritev prejšnjega meseca in katera meritev trenutnega meseca pripadata istemu uporabniku. Recimo, da je neki uporabnik konec prejšnjega meseca oddal meritev a_i, konec tega meseca pa b_j; če je $b_j \geq a_i$, lahko zaključimo, da je njegova poraba vsaj $b_j - a_i$ (lahko pa bi bila tudi za neki večkratnik 100 000 večja, ker je števec pač le petmestni); če pa je $b_j < a_i$, lahko zaključimo, da se je števec tega uporabnika ta mesec očitno resetiral na 0 in je bila njegova poraba vsaj $100\,000 + b_j - a_i$ (lahko pa bi bila tudi za neki večkratnik 100 000 večja).

Naivna rešitev bi torej bila, da bi pri vsakem a_i preizkusili vse možne b_j, izračunali pri vsaki spodnjo mejo porabe (v spodnji psevdokodi je to p) po razmisleku iz prejšnjega odstavka ter vrnili najmanjšo od tako dobljenih meja (r v spodnji psevdokodi):

$r := 10^5$;
for $i := 1$ **to** n **do for** $j := 1$ **to** n:
  $p := b_j - a_i$; **if** $p < 0$ **then** $p := p + 10^5$;
  if $p < r$ **then** $r := p$;
return r;

Ker moramo pregledati n^2 parov (i, j), ima ta postopek časovno zahtevnost $O(n^2)$, kar je neugodno, saj naloga želi učinkovito rešitev, ki bi delovala hitro tudi za velike n.

Pri posameznem a_i nas v resnici ne zanima poraba pri vseh možnih b_j, pač pa le pri tistem, ki bo dal najnižjo mejo porabe (za ta a_i); to pa je najmanjši tak b_j, ki je večji ali enak a_i. (Če pa takega b_j sploh ni, ker so vsi manjši od a_i, lahko zaključimo, da se je števec uporabnika i ta mesec zagotovo resetiral in bomo najmanjšo oceno porabe zanj dobili pri b_1.) Spomnimo se, da sta obe zaporedji meritev urejeni naraščajoče; če smo za neki a_i že našli primeren b_j in si nato enako vprašanje zastavimo pri a_{i+1}, vidimo, da bo zanj pravi bodisi isti b_j bodisi (če je ta manjši od a_{i+1}) neki kasnejši $b_{j'}$ za $j' > j$, ni pa nam treba gledati manjših meritev $b_{j'}$ za $j' < j$, kajti tiste so bile manjše od a_i in so zato manjše tudi od a_{i+1} (ki je $\geq a_i$). Tako torej, ko v zunanji zanki povečujemo i, v notranji zanki ni treba iti z j vsakič od začetka, ampak lahko nadaljujemo tam, kjer smo ostali po prejšnji iteraciji zunanje zanke. Paziti moramo le še na to, da ko pride j do konca seznama meritev in je tudi zadnji element, b_n, premajhen (manjši od a_i), moramo odtlej vseskozi uporabljati b_1.

$r := 10^5$; $j := 1$;
for $i := 1$ **to** n:
  while $j \leq n$ **and** $b_j < a_i$ **do** $j := j + 1$;
  if $j > n$ **then** $p := b_1 - a_i + 10^5$ **else** $p := b_j - a_i$;
  if $p < r$ **then** $r := p$;
return r;

Tako gre zunanja zanka enkrat po vseh a_i, notranja pa tudi enkrat po vseh b_j in imamo rešitev z linearno časovno zahtevnostjo, $O(n)$.

REŠITVE NALOG ZA DRUGO SKUPINO

1. Palačinke

Za urejanje palačink lahko uporabimo neke vrste urejanje z izbiranjem. Poglejmo, kje na skladovnici je največja palačinka; recimo, da je to a_k. Obrnimo zgornjih k palačink, pa pride največja na vrh. V naslednjem koraku obrnimo vseh n palačink in največja pride na dno, prav tam pa bo tudi morala biti, ko bo skladovnica urejena. Zato jo bomo odslej pustili tam na dnu skladovnice in se v prihodnje ukvarjali le še z ostalimi $n-1$ palačinkami. Zdaj spet poiščemo največjo med njimi, jo z enim obračanjem spravimo na vrh skladovnice in s še enim na $(n-1)$-vo mesto, torej tja, kjer bo morala biti, ko bo skladovnica urejena. Tako nadaljujemo, dokler ne pridejo vse palačinke na svoja mesta; Natančneje povedano, ko spravimo $n-1$ palačink na pravo mesto, je edina preostala palačinka (to je najmanjša od vseh) že tudi na pravem mestu in se lahko ustavimo. Za urejanje celotne skladovnice bomo tako porabili $2 \cdot (n-1)$ korakov.

Oglejmo si implementacijo te rešitve v C++; naš podprogram Uredi dobi seznam palačink v vektorju S, operacije obračanja pa izpisuje na standardni izhod:

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

void Obrni(vector<int> &S, int k)
{
  cout << k << endl; // Izpišimo trenutno operacijo.
  // Primerno popravimo stanje skladovnice v vektorju S.
  for (int i = 0; i < --k; ++i) swap(S[i], S[k]);
}

void Uredi(vector<int> &S)
{
  for (int n = (int) S.size(); n > 1; --n)
  {
    // Urejamo le še zgornjih n palačink, S[0], ..., S[n - 1],
    // v preostanku seznama S pa so palačinke že na pravih mestih.
    // Poiščimo največjo izmed zgornjih n palačink.
    int k = 0;
    for (int i = 1; i < n; ++i)
      if (S[i] > S[k]) k = i;
    Obrni(S, k + 1); // Premaknimo največjo palačinko na vrh.
    Obrni(S, n);     // Premaknimo jo na pravo mesto.
  }
}
```

Imamo torej glavno zanko z $n-1$ iteracijami, v vsaki od teh iteracij pa porabimo $O(n)$ časa za iskanje največje palačinke in še $O(n)$, da popravimo vsebino vektorja S, ki predstavlja našo skladovnico. Tako ima ta rešitev časovno zahtevnost $O(n^2)$. Za tekmovanje v drugi skupini je to dovolj dobro, kot zanimivost pa vseeno razmislimo, kako jo lahko izboljšamo. Namesto v seznamu (tabeli, vektorju ipd.) bi bilo palačinke dobro hraniti v kakšni uravnoteženi drevesasti strukturi (npr. treap, rdeče-črno drevo, lomljeno drevo ipd.). V vsakem vozlišču drevesa vzdržujmo tudi podatke o tem, koliko vozlišč (in s tem palačink) je v njegovem poddrevesu in kako velika je največja od njih. Globina takšnega drevesa bo $O(\log n)$, zato bomo lahko v $O(\log n)$ časa ugotovili, kje v drevesu je največja palačinka in koliko jih je nad njo. Da pa bomo hitro izvedli tudi operacijo obračanja zgornjih nekaj palačink, je treba v vsakem vozlišču drevesa hraniti tudi zastavico (oz. logično vrednost), ki pove, ali je vrstni red palačink v njegovem poddrevesu ravno obrnjen (glede na vrstni red, ki velja v njegovem staršu); tako nam na primer pri obračanju k palačink ne bo treba spremeniti k vozlišč drevesa, ampak bomo morali le obrniti zastavice pri $O(\log n)$ vozliščih (podrobnosti tega, kaj točno bo treba pri tem narediti, so sicer precej odvisne od tega, katero vrsto dreves uporabimo). Tako

bomo imeli v vsaki iteraciji zunanje zanke le $O(\log n)$ dela, časovna zahtevnost celotne rešitve pa bo $O(n \log n)$.

2. ChordPro v2

Vhodne podatke bomo brali po vrsticah in šli pri vsaki vrstici v zanki po njenih znakih. (Ni sicer nujno brati po vrsticah, je pa lažje in za naš namen dovolj dobro, saj besedilo naloge zagotavlja, da vrstice niso daljše od dvesto znakov.) Znake besedila, torej tiste zunaj oglatih oklepajev, sproti tudi izpisujemo. Ko pa pridemo do oglatega oklepaja, poiščimo prvi naslednji oglati zaklepaj. Če smo v prvi kitici, je podniz v oglatih oklepajih akord, ki ga moramo ne le izpisati, ampak si ga tudi zapomniti, da ga bomo na primernem mestu vrinili v kasnejše kitice. V ta namen potrebujemo torej nekakšen seznam nizov; spodnja rešitev jih hrani v vektorju akordi. Če pa nismo v prvi kitici, bo treba akord izpisati iz omenjenega seznama, saj imamo v vhodnem nizu takrat le prazen par oglatih oklepajev „[]“.

Vidimo torej, da bomo morali vzdrževati podatek o tem, pri katerem akordu smo (da bomo vedeli, kateri element seznama izpisati), in o tem, ali smo v prvi kitici ali ne (da vemo, ali naj akorde shranjujemo ali ne). Imejmo torej še dve spremenljivki: prva pove, ali smo v prvi kitici (na začetku smo, čim pa naletimo na prazno vrstico, vemo, da nismo več v prvi kitici), stAkorda pa šteje akorde znotraj trenutne kitice (povečamo jo za 1 pri vsakem oglatem oklepaju, na koncu kitice pa jo postavimo nazaj na 0).

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

int main()
{
  // Preberimo n in znak za konec vrstice.
  int n; cin >> n;
  string s; getline(cin, s);

  // Obdelajmo preostalih n vhodnih vrstic.
  vector<string> akordi; // Seznam akordov iz prve kitice.
  bool prva = true;      // Ali smo trenutno v prvi kitici?
  int stAkorda = 0;      // Pri katerem akordu trenutne kitici smo?

  while (n-- > 0)
  {
    // Preberimo naslednjo vrstico.
    getline(cin, s);

    // Če je vrstica prazna, se začenja nova kitica, torej vsekakor
    // nismo več v prvi kitici, akorde pa štejmo spet od 0 naprej.
    if (s.empty()) prva = false, stAkorda = 0;

    // Pojdimo po znakih trenutne vrstice.
    for (int i = 0, d = (int) s.length(); i < d; ++i)
    {
      // Če nismo pri akordu, trenutni znak le izpišimo.
      if (s[i] != '[') { cout.put(s[i]); continue; }

      // Poiščimo zaklepaj na koncu akorda.
      int j = i; while (s[++i] != ']') ;

      // Če smo v prvi kitici, si pravkar prebrani akord zapomnimo.
      if (prva) akordi.emplace_back(&s[j], i - j + 1);

      // Trenutni akord tudi izpišimo.
      cout << akordi[stAkorda++];
    }
    cout << endl;
  }
  return 0;
}
```

3. Parica

Preveriti moramo dva pogoja, ki ju omenja besedilo naloge; razmislimo o vsakem pogoju posebej.

(1) Prvi pogoj pravi, da se mora to, kateri kontakt na eni strani je povezan s katerim na drugi strani, ujemati z enim od treh standardnih povezovanj, ki jih omenja besedilo naloge.

(1.1) Glede tistih treh standardnih povezovanj pa vidimo, da ima kabel pri prvih dveh povezovanjih na obeh koncih enak konektor in torej za vsak i velja, da je i-ti kontakt enega konektorja povezan z i-tim kontaktom drugega.

(1.2) Pri tretjem standardnem povezovanju pa ima kabel na enem koncu konektor T568A, na drugem pa T568B; če primerjamo vrstni red žic v teh dveh konektorjih, vidimo, da je i-ti kontakt prvega povezan z i-tim kontaktom drugega za $i \in \{4, 5, 7, 8\}$; poleg tega sta povezana prvi kontakt enega konektorja in tretji kontakt drugega (in obratno), podobno pa tudi drugi kontakt enega konektorja in šesti kontakt drugega (in obratno).

Če ima torej kabel iz naših vhodnih podatkov na obeh koncih enak vrstni red žic, zapade pod primer (1.1); če dobimo vrstni red na drugem koncu tako, da v vrstnem redu na prvem koncu zamenjamo prvo in tretjo žico ter drugo in šesto, zapade naš kabel pod primer (1.2); če pa ne velja nič od tega dvojega, lahko takoj zaključimo, da kabel ne izpolnjuje prvega pogoja in zato ne bo deloval.

(2) Drugi pogoj se nanaša na vsak konec kabla posebej in pravi, da morata biti žici, ki tvorita par iste barve, povezani na tak par kontaktov, na katerega sta tudi pri shemah T568A in T568B povezani žici iste barve. Če si ogledamo spet tabelo v besedilu naloge, vidimo, da so ti pari kontaktov pri obeh shemah enaki: $\{1, 2\}$, $\{3, 6\}$, $\{4, 5\}$ in $\{7, 8\}$ (tega slednjega nam pove že besedilo naloge). Shemi se razlikujeta le po tem, kateri par žic je povezan na kateri par kontaktov (na primer: na kontaktih $\{1, 2\}$ ima T568A zeleni žici, T568B pa oranžni), vendar za naš namen to ni pomembno. Za vsakega od štirih parov moramo torej preveriti le, ali sta žici na tistem paru kontaktov iste barve.

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

bool IsteBarve(const string &s, const string &t) { return s.back() == t.back(); }

int main()
{
  bool ok = true;
  string konca[2][8];
  for (auto &K : konca)
  {
    // Preberimo ta konec kabla.
    for (auto &barva : K) cin >> barva;

    // Preverimo drugi pogoj na tem koncu kabla.
    if (! IsteBarve(K[0], K[1]) || ! IsteBarve(K[2], K[5]) ||
        ! IsteBarve(K[3], K[4]) || ! IsteBarve(K[6], K[7])) ok = false;
  }

  // Preverimo prvi pogoj.
  auto &A = konca[0], &B = konca[1];
  if (A[3] != B[3] || A[4] != B[4] || A[6] != B[6] || A[7] != B[7]) ok = false;
  if (! (A[0] == B[0] && A[1] == B[1] && A[2] == B[2] && A[5] == B[5]) &&
      ! (A[0] == B[2] && A[1] == B[5] && A[2] == B[0] && A[5] == B[1])) ok = false;

  // Izpišimo rezultat.
  cout << (ok ? "Bo" : "Ne bo") << " deloval." << endl; return 0;
}
```

Mimogrede, če velja prvi pogoj in če velja na enem koncu kabla drugi pogoj, potem bo zagotovo veljal drugi pogoj tudi na drugem koncu kabla.[1] Rešitev bi lahko torej

[1] Če prvi pogoj velja po primeru (1.1), je to povsem očitno, saj sta vsaki dve žici povezani na drugem

preverila prvi pogoj enako kot doslej, drugi pogoj pa le na enem koncu kabla, pa bi bila še vseeno pravilna.

Na vsakem koncu kabla je $8! = 40\,320$ možnosti glede tega, v kakšnem vrstnem redu so žice povezane na kontakte; ker ima kabel dva konca, je torej pri tej nalogi možnih $(8!)^2 \approx 1{,}6$ milijarde različnih vhodnih primerov. Kot zanimivost omenimo, da bi kabel deloval le pri 768 izmed njih.[2]

4. Diagram

Višino črte lahko opišemo s tem, koliko znakov | ima na vsaki strani; nad tem je potem v vsakem primeru še vrstica z znaki /, - in \. Naloga pravi, da mora biti črta za posamezno linijo najnižja, kolikor je mogoče; biti pa mora vsaj za 1 višja od vsake linije, ki je vgnezdena vanjo (če vanjo ni vgnezdena nobena, ima lahko višino 0, torej je narisana tik nad nizom z opisom železnice na dnu slike). Zato je koristno računati višine linij od notranjih (globlje vgnezdenih) proti zunanjim; ko poznamo za neko linijo višine vseh, ki so vgnezdene vanjo, ne bo težko določiti njene višine (kot 1 + maksimum višin vgnezdenih linij).

Opis železnice pregledujmo znak po znak od leve proti desni; pri tem bomo vzdrževali sklad trenutno odprtih linij, torej takih, ki so se že začele, nismo pa še videli njihovega konca. Ko pridemo do velike črke, ki označuje začetek linije, jo dodajmo na vrh sklada in si tudi zapomnimo, pri kateri x-koordinati se je začela (to bo prišlo prav kasneje pri risanju črte zanjo). Ko pa pridemo do male črke, vemo, da se je končala najgloblje vgnezdena izmed trenutno odprtih linij, torej tista z vrha sklada (saj naloga zagotavlja, da bodo linije lepo vgnezdene). Takrat jo torej pobrišimo s sklada, zapomnimo si, pri kateri x-koordinati se je končala, poleg tega pa po potrebi tudi povečajmo višino njej neposredno „nadrejene“ linije, torej tiste, v kateri je bila pravkar končana linija vgnezdena; ta nadrejena linija je ravno tista, ki je na vrhu sklada zdaj, ko smo pravkar končano linijo že pobrisali. Če pa je sklad zdaj prazen, je to znak, da pravkar končana linija ni bila vgnezdena v nobeni drugi; tedaj pa njena višina vpliva na višino celotnega diagrama (število vrstic le-tega je 1 + višina najvišje linije). Na začetku lahko torej višino vsake linije in tudi celotnega diagrama inicializiramo na 0, nato pa jih po potrebi povečujemo, ko pridemo do konca kakšne vgnezdene linije.

Ko na ta način pridemo do konca opisa, imamo vse, kar potrebujemo, da narišemo diagram: vemo, koliko vrstic mora biti visok, in za vsako linijo vemo, kako visoka mora biti njena črta in od katere do katere x-koordinate sega. Pripravimo si dovolj veliko dvodimenzionalno tabelo znakov (oz. kar seznam nizov, za vsako vrstico po enega), jo inicializirajmo s presledki in nato vanjo na primerna mesta vpišimo znake |, /, - in \. Tako dobljeni diagram moramo nato le še izpisati, pod njim pa tudi vrstico z opisom železnice.

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

void NarisiDiagram(const string &s)
{
  struct Linija { int x1, x2, h; }; // x začetka, x konca, višina
  vector<Linija> linije;
```

koncu na enak par kontaktov kot na prvem. Če pa velja prvi pogoj po primeru (1.2), lahko opazimo, da sta žici, ki sta na prvem koncu povezani na kontakta $\{1, 2\}$, na drugem povezani na $\{3, 6\}$ in obratno; žici pa, ki sta na prvem koncu povezani na $\{4, 5\}$ ali na $\{7, 8\}$, sta povezani na isti par tudi na drugem koncu; v vsakem primeru torej vidimo, da če velja drugi pogoj na prvem koncu in sta torej na vsakega od parov $\{1, 2\}$, $\{3, 6\}$, $\{4, 5\}$, $\{7, 8\}$ povezani žici iste barve, bosta tidve žici povezani na enega od teh štirih parov kontaktov (ne nujno istega) tudi na drugem koncu kabla.

[2] Do tega števila lahko pridemo s poskusom, lahko pa tudi z razmislekom. Za drugi pogoj na enem koncu kabla imamo $4! = 24$ načinov, kako razdeliti barve med štiri pare kontaktov. Pri vsakem paru pa imamo tudi dve možnosti glede tega, kateri izmed obeh kontaktov v paru dobi belo-barvno žico; tako imamo skupaj $4! \cdot 2^4 = 24 \cdot 16 = 384$ načinov, kako si izbrati prvi konec kabla. Za nasprotni konec imamo potem le dve možnosti, če hočemo, da bo izpolnjen prvi pogoj; tako dobimo skupaj $2 \cdot 384 = 768$ možnosti.

```
  stack<int> sklad; // indeksi (v 'linije') trenutno odprtih linij
  int w = (int) s.length(), h = 0;
  for (int x = 0; x < w; ++x) // Preglejmo opis železnice.
    if (s[x] >= 'A' && s[x] <= 'Z') {
      // Začenja se linija, dodajmo jo v seznam in na sklad.
      sklad.push((int) linije.size());
      linije.push_back({x, -1, 0}); }
    else if (s[x] >= 'a' && s[x] <= 'z') {
      // Najbolj notranja vgnezdena linija se končuje. Pobrišimo jo s sklada,
      // zapomnimo pa si njeno številko.
      int i = sklad.top(); sklad.pop();
      // Zapomnimo si, da se končuje pri tem x.
      auto &L = linije[i]; L.x2 = x;
      // Višina te linije vpliva na višino linije, v kateri je vgnezdena
      // (to pa je tista, ki je zdaj na vrhu sklada); če pa ni vgnezdena
      // v nobeni, vpliva na višino celotnega diagrama.
      auto &H = sklad.empty() ? h : linije[sklad.top()].h;
      if (L.h >= H) H = L.h + 1; }
  // Pripravimo si diagram kot tabelo nizov.
  vector<string> diagram(h, string(w, ' '));
  for (auto &L : linije) {
    for (int y = 0; y < L.h; ++y) diagram[y][L.x1] = '|', diagram[y][L.x2] = '|';
    for (int x = L.x1; x <= L.x2; ++x)
      diagram[L.h][x] = (x == L.x1) ? '/' : (x == L.x2) ? '\\' : '-'; }
  // Izpišimo diagram (z opisom železnice vred).
  for (int y = h - 1; y >= 0; --y) cout << diagram[y] << endl;
  cout << s << endl;
}
```

5. Alkoholni test

Ogledali si bomo dve (sicer precej podobni) rešitvi te naloge. Ena možnost je, da si za temelj iskanja skupin C–O–H vzamemo atom C, ki ima to lepo lastnost, da je v vsaki spojini en sam. Pojdimo v zanki po vseh znakih mreže; ko najdemo C, preglejmo vse štiri smeri (spomnimo se, da naloga zagotavlja, da ima vsak C štiri vezi in da je vsaka spojina vidna v celoti, torej nam ni treba skrbeti, da bi pri pregledovanju C-jevih sosedov padli čez rob mreže); če v kakšni smeri opazimo, da je C povezan z O-jem, pa moramo zdaj še preveriti, ali je tisti O povezan tudi s H-jem. Ker ni nujno, da sta obe O-jevi vezi v isti smeri, ampak sta lahko pod pravim kotom ena na drugo, moramo tu v še eni vgnezdeni zanki preizkusiti vse možne smeri vezi O–H. V vsaki smeri moramo preveriti tako to, ali je na polju tik ob O-ju pravi znak za vez (torej „-“ ali „|“; pri tem pazimo tudi, da ne pademo čez rob mreže), kot tudi to, ali je potem naslednji znak v tisti smeri H. Če se vse to izide, potem vemo, da smo našli skupino C–O–H in da je torej spojina, ki ji pripada naš trenutni C, alkohol. Isti C je lahko vključen v več takšnih skupin in paziti moramo, da ga štejemo samo enkrat.

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

int main()
{
  const int DX[] = {-1, 1, 0, 0}, DY[] = {0, 0, -1, 1};
  // Preberimo vhodno mrežo.
  int n, m; cin >> n >> m;
  vector<string> A(n); for (auto &s : A) cin >> s;
  // Preštejmo alkohole na sliki.
  int stAlkoholov = 0;
  for (int yC = 0; yC < n; ++yC) for (int xC = 0; xC < m; ++xC) if (A[yC][xC] == 'C')
```

```
  {
    bool jeAlkohol = false;
    // Našli smo C. Poglejmo, ali je povezan s kakšnim O.
    for (int dO = 0; dO < 4 && ! jeAlkohol; ++dO) {
      int xO = xC + 2 * DX[dO], yO = yC + 2 * DY[dO];
      if (A[yO][xO] != 'O') continue;
      // Našli smo C–O. Poglejmo, ali je O povezan s kakšnim H.
      for (int dH = 0; dH < 4 && ! jeAlkohol; ++dH) {
        // Ali ima O v tej smeri vez?
        int xH = xO + DX[dH], yH = yO + DY[dH];
        if (xH < 0 || xH >= m || yH < 0 || yH >= n) continue;
        if (A[yH][xH] != (yH == yO ? '-' : '|')) continue;
        // Ali je na drugem koncu te vezi H?
        if (A[yH + DY[dH]][xH + DX[dH]] == 'H') jeAlkohol = true; } }
    if (jeAlkohol) ++stAlkoholov;
  }
  cout << stAlkoholov << endl; return 0; // Izpišimo rezultat.
}
```

Nalogo pa lahko rešimo tudi tako, da za temelj iskanja skupin C–O–H vzamemo atom O, ki ima to lepo lastnost, da je vezan tako na C kot na H, zato bomo lahko že samo s pregledom njegove soseščine preverili, ali imamo skupino C–O–H ali ne; potrebovali bomo eno gnezdeno zanko manj kot pri prejšnji rešitvi. Ko torej najdemo O, preglejmo v zanki vse štiri smeri in si zapomnimo, ali smo v kakšni smeri videli H in/ali C. Če smo videli oba, je to znak, da naš O pripada neki skupini C–O–H. V tem primeru je koristno tisti C potem pobrisati oz. ga povoziti z nekim neveljavnim znakom, tako da bomo, če je ta C morda vključen v več skupin C–O–H, šteli le eno od njih (ostalih pa ne bomo opazili, ker tam ne bo več C-ja).[3] Ta podrobnost je pomembna, ker naloga zahteva, da štejemo spojine s skupino C–O–H, ne pa teh skupin samih; če jih ima neka spojina več, jo moramo še vseeno šteti le enkrat. (Še ena možnost je tudi, da C-ja ne pobrišemo, pač pa dodamo njegove koordinate v neko množico oz. razpršeno tabelo, kjer bomo zlahka videli, ali smo ta C našli že kdaj prej.)

```
int stAlkoholov = 0;
for (int y = 0; y < n; ++y) for (int x = 0; x < m; ++x) if (A[y][x] == 'O')
{
  bool imaH = false; int dC = -1;
  // Našli smo O. Poglejmo, ali je povezan s kakšnima C in H.
  for (int d = 0; d < 4; ++d) {
    int xx = x + DX[d], yy = y + DY[d];
    if (xx < 0 || xx >= m || yy < 0 || yy >= n) continue;
    if (A[yy][xx] != (yy == y ? '-' : '|')) continue;
    char c = A[yy + DY[d]][xx + DX[d]];
    if (c == 'H') imaH = true;
    else if (c == 'C') dC = d; }
  // Če smo našli C–O–H, povečajmo števec alkoholov,
  // C pa pobrišimo, da ga ne bomo kasneje šteli še kdaj.
  if (imaH && dC >= 0)
    ++stAlkoholov, A[y + 2 * DY[dC]][x + 2 * DX[dC]] = '#';
}
```

Preostanek programa (branje vhodnih podatkov in izpis rezultatov) je enak kot pri prvi rešitvi, zato ga nismo pisali še enkrat.

[3] To pa pomeni, da si moramo zapomniti, v kateri smeri glede na O smo videli C. Pazimo tudi na to, da smemo C pobrisati šele, če smo res našli skupino C–O–H; če najdemo le C, ne pa tudi H-ja, moramo C pustiti, kajti čeprav ni vključen v C–O–H prek trenutnega O-ja, je morda vključen v takšno skupino prek kakšnega drugega O-ja, ki ga bomo pregledali šele v prihodnje.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Giganti

Pri naši nalogi imajo vse gore višino, večjo od 0; če imamo torej dve gori ter zamenjamo eno od njiju z njuno vsoto, je le-ta vsekakor višja od preostale gore (tiste, ki je nismo zamenjali z vsoto).

Na koncu bi radi imeli gori c in d. Recimo brez izgube za splošnost, da je $c < d$; potem je d gora, ki je nastala z združitvijo v zadnjem koraku, c pa tista, ki smo jo imeli že pred tem zadnjim korakom. Druga gora, ki smo jo imeli pred zadnjim korakom, je morala biti torej $d - c$, saj je to edina gora, ki skupaj z goro c dá vsoto d. Vidimo torej, da lahko v stanje $\{c, d\}$ pridemo le iz $\{c, d - c\}$; to je edina možnost.

Če je zdaj morda c še vedno manjša izmed obeh gora, torej $c < d - c$, nam enak razmislek pove, da smo morali še en korak prej imeti gori $\{c, d - 2c\}$, še pred tem $\{c, d - 3c\}$ in tako naprej. Pišimo $k := \lfloor d/c \rfloor$ in $r := d \bmod c$, tako da je $d = k \cdot c + r$; potemtakem smo k korakov pred stanjem $\{c, d\}$ imeli stanje $\{c, d - k \cdot c\}$, kar je isto kot $\{c, r\}$ ali $\{r, c\}$, saj vrstni red gora pri naši nalogi ni pomemben. Ker je r ostanek po deljenju d-ja s c, je r manjši od c, zato je zdaj c višja od obeh gora; še en korak prej smo torej imeli gori $\{r, c - r\}$, pred tem $\{r, c - 2r\}$ in tako naprej vse do $\{r, c \bmod r\}$.

Vidimo lahko, da imamo opravka s prav takšnimi števili, kot če bi z Evklidovim algoritmom računali največjega skupnega delitelja števil c in d. Gori postajata vse nižji in prej ali slej pridemo v položaj, ko je višina ene večkratnik višine druge. Ko takrat odštevamo manjšo od večje, sčasoma postaneta enaki; to pa je stanje, v katero ne moremo priti iz nobenega drugega, zgodnejšega stanja, kajti za kaj takega bi morala biti pred tem ena od gora visoka 0, pri naši nalogi pa so višine vedno večje od 0.

Tako torej vidimo, da je mogoče do stanja $\{c, d\}$ priti po eni sami poti; vse, kar moramo narediti, je, da ji sledimo (v obratni smeri), štejemo prehojene korake in se ustavimo, ko pridemo do stanja $\{a, b\}$. Recimo, da je $a \leq b$ in $u \leq v$; ko gledamo na primer korake nazaj od $\{u, v\}$ do $\{u, v \bmod u\}$, moramo torej preveriti, ali je $u = a$ in ali je b oblike $b = v - k \cdot u$ za neki $k \geq 0$; če je tako, potem vemo, da se do $\{a, b\}$ pride v k korakih nazaj iz $\{u, v\}$.

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

int main()
{
  int q; cin >> q;
  while (q-- > 0)
  {
    long long a, b, c, d; cin >> a >> b >> c >> d;
    // Poglejmo, iz katerih stanj je mogoče priti do (c, d).
    if (a > b) swap(a, b);
    if (c > d) swap(c, d);
    long long stKorakov = 0;
    while (c >= a)
    {
      // Ali leži (a, b) na poti od (c, d) do (c, d mod c)?
      if (a == c && b <= d && (d - b) % c == 0) {
        stKorakov += (d - b) / c; break; }
      // Sicer se premaknimo vse do (c, d mod c).
      stKorakov += d / c; d %= c; swap(c, d);
    }
    // Če se zanka ni ustavila s stavkom break, je ta primer nerešljiv.
    if (c < a) stKorakov = -1;
    cout << stKorakov << endl;
  }
  return 0;
}
```

2. Multiprocesiranje

Naloga je zelo primerna za reševanje z dinamičnim programiranjem. Recimo, da se omejimo le na prvih m (od n) nalog in na prvih q (od p) procesorjev; število razporedov teh nalog na te procesorje označimo s $f(m, q)$. Zadnjo, m-to, od teh nalog moramo dodeliti nekemu procesorju, recimo r, za katerega mora seveda veljati $a_m \leq r \leq b_m$ (kajti nalogo m lahko dodelimo le tem procesorjem) in $r \leq q$ (saj smo se omejili na prvih q procesorjev). Spomnimo se, da naloga pravi, da se morajo pomembnejše naloge (take z večjo številko) izvajati na močnejših procesorjih (takih z večjo številko). Ker smo torej dodelili nalogo m procesorju r, smemo za naloge od 1 do $m-1$ uporabljati le procesorje od 1 do r, ne pa tudi procesorjev od $r+1$ do q. Ostal nam je torej problem, kako razporediti prvih $m-1$ nalog na prvih r procesorjev, za to pa vemo, da je mogoče na $f(m-1, r)$ načinov. Toliko je torej tudi načinov, kako razporediti prvih m nalog na prvih q procesorjev z dodatno omejitvijo, da pride naloga m na procesor r. Da dobimo število vseh razporedov m nalog na q procesorjev, moramo to sešteti po vseh r:

$$f(m, q) = \sum_{r=a_m}^{\min\{b_m, q\}} f(m-1, r).$$

Tako smo dobili rekurzivno zvezo, s katero ni težko računati vrednosti funkcije f. Robni primer nastopi, ko nimamo nobenih nalog več: $f(0, q) = 1$ za vsak q. Rezultat, po katerem sprašuje naloga, je $f(n, p) \bmod M$ za $M = 10^9 + 7$; ker nas bo torej na koncu zanimal le ostanek po deljenju z M in ker za takšne ostanke velja $(u + v) \bmod M = ((u \bmod M) + (v \bmod M)) \bmod M$, lahko ves čas računamo le ostanke $f(m, q) \bmod M$ in nam nikoli ni treba delati s števili, večjimi od $2M$.

Če se bomo lotili izračuna vrednosti $f(m, q)$ preveč naivno, z zanko po r, da izračunamo vsoto iz gornje formule, nam bo ta izračun vzel $O(p)$ časa, zato bo imela celotna rešitev časovno zahtevnost $O(n \cdot p^2)$, kar je za večje testne primere že preveč (s takšno rešitvijo bi na tekmovanju dobili pri tej nalogi 50 točk od stotih). Bolje je, če si po tistem, ko izračunamo $f(m-1, r)$ za vse r, pripravimo delne vsote: naj bo

$$g(m-1, r) = \sum_{t=1}^{r} f(m-1, t),$$

kar lahko seveda računamo kot

$$g(m-1, 0) = 0 \quad \text{in} \quad g(m-1, r) = g(m-1, r-1) + f(m-1, r).$$

Vsoto nekaj zaporednih vrednosti $f(m-1, \cdot)$ lahko potem dobimo kot razliko dveh delnih vsot $g(m-1, \cdot)$:

$$f(m, q) = \sum_{r=a_m}^{\min\{b_m, q\}} f(m-1, r) = g(m-1, \min\{b_m, q\}) - g(m-1, a_m - 1).$$

Tako porabimo pri vsakem m najprej $O(p)$ časa za izračun delnih vsot in nato še $O(p)$ časa za izračun vseh $f(m, \cdot)$; skupaj je časovna zahtevnost celotne rešitve le še $O(n \cdot p)$.

Glede porabe pomnilnika omenimo še, da ko izračunamo vse $f(m, \cdot)$, odtlej ne bomo več potrebovali vrednosti $f(m-1, \cdot)$ in jih lahko pozabimo. Tako je dovolj hraniti vrednosti funkcije f le za dva zaporedna m-ja, torej potrebujemo le $O(p)$ pomnilnika, ne pa $O(n \cdot p)$.

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

enum { M = 1'000'000'007 };

int main()
{
```

```
  int n, p; cin >> n >> p;
  vector<int> f(p, 1), f1(p), g1(p + 1); g1[0] = 0;
  for (int m = 0; m < n; ++m)
  {
    // Preberimo podatke o naslednji, (m + 1)-vi nalogi.
    int am, bm; cin >> am >> bm; --am; --bm;

    // V f[q] je število razporedov prvih m nalog na prvih q + 1 procesorjev.
    // Premaknimo to v f1 in izračunajmo delne vsote tega zaporedja.
    swap(f, f1);
    for (int r = 0; r < p; ++r) g1[r + 1] = (g1[r] + f1[r]) % M;

    // V f[q] izračunajmo število razporedov prvih m + 1 nalog na prvih q + 1 procesorjev.
    for (int q = 0; q < p; ++q)
      f[q] = (q < am) ? 0 : (g1[min(bm, q) + 1] + (M - g1[am])) % M;
  }
  cout << f[p - 1] << endl; return 0; // Izpišimo rezultat.
}
```

3. Tabela ničel

Označimo našo vhodno tabelo s T. Zadnja operacija bo bodisi pustila v tabeli neko vrstico samih enic ali pa neki stolpec samih ničel. Če torej v T ni nobene take vrstice in nobenega takega stolpca, je problem nerešljiv (ampak naša naloga na srečo zagotavlja, da bomo dobili le rešljive primere). Sicer imamo v T bodisi eno ali več vrstic samih enic ali pa enega ali več stolpcev samih ničel (oboje hkrati je nemogoče). Zapomnimo si, da bodo na koncu zaporedja morale biti operacije, ki postavijo te vrstice na enice oz. te stolpce na ničle, nato pa v mislih pobrišimo te vrstice oz. stolpce iz T-ja. Na tako zmanjšani tabeli ponovimo enak razmislek in tako nadaljujemo, dokler ne pobrišemo vseh vrstic ali stolpcev (takrat vemo, da je problem rešljiv) bodisi ne pridemo do nerešljivega problema (kar se sicer, kot rečeno, pri testnih primerih na našem tekmovanju ne bo zgodilo). Če zaporedje, v katerem smo brisali vrstice oz. stolpce, obrnemo, dobimo ravno vrstni red, v katerem je treba izvajati operacije na tabeli, da bo na koncu prišla v stanje T, s katerim smo naš postopek začeli.

Za učinkovito implementacijo tega postopka je koristno, če za vsako vrstico pripravimo števec enic v njej, za vsak stolpec pa števec ničel. Ko pobrišemo neko vrstico samih enic, se števci ničel v stolpcih nič ne spremenijo; in podobno, ko pobrišemo stolpec ničel, se števci enic v vrsticah nič ne spremenijo. Števce moramo zato določiti le na začetku.

$štVrstic := h$; $štStolpcev := w$;
for $y := 1$ **to** h **do** $štEnic[y] := 0$;
for $x := 1$ **to** w **do** $štNičel[x] := 0$;
for $y := 1$ **to** h **do for** $x := 1$ **to** w **do**
  if $T[y, x] = 1$ **then** $štEnic[y] \mathrel{+}= 1$ **else** $štNičel[x] \mathrel{+}= 1$;
$L :=$ prazen seznam;
while $štVrstic > 0$ **and** $štStolpcev > 0$:
  if obstaja kakšna (nepobrisana) vrstica y, ki ima $štEnic[y] = štStolpcev$:
    pobriši vrstico y; $štVrstic := štVrstic - 1$;
    dodaj v L par $(1, y)$;
  else if obstaja kakšen (nepobrisan) stolpec x, ki ima $štNičel[x] = štVrstic$:
    pobriši stolpec x; $štStolpcev := štStolpcev - 1$;
    $štNičel[x] := -\infty$;
  else:
    če pridemo do sem, je problem (za dano vhodno tabelo T) nerešljiv;
izpiši operacije iz L v obratnem vrstnem redu;

Ko moramo preveriti, ali obstaja kakšna (še ne pobrisana) vrstica y, ki ima toliko enic, kolikor je (še ne pobrisanih) stolpcev, imejmo v mislih, da takrat nobena vrstica ne more imeti več kot toliko enic (ker zdaj sploh nimamo več kot toliko stolpcev); dovolj je torej, če pogledamo tisto vrstico (izmed še nepobrisanih), ki ima največ enic. Če jih ona nima dovolj, jih nima nobena (in bo treba namesto tega na podoben način preveriti, ali obstaja kak primeren stolpec). Vidimo torej, da je koristno, če si na začetku pripravimo

seznam vrstic, urejen naraščajoče po številu enic; na vsakem koraku glavne zanke v gornjem postopku moramo potem pogledati vrstico na koncu tega seznama in jo morda tudi pobrisati. Ker gredo števila enic le od 0 do w, lahko uporabimo kar urejanje s štetjem (*counting sort*). Podobno moramo seveda imeti tudi seznam stolpcev, urejen naraščajoče po številu ničel.

Naš gornji postopek porabi $O(w \cdot h)$ časa in še to le zaradi branja vhodnih podatkov in štetja ničel v stolpcih oz. enic v vrsticah; preostanek postopka pa porabi le $O(w + h)$ časa. Oglejmo si še njegovo implementacijo v C++:[4]

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

// V vektorju V[0..n − 1] pričakuje števila z območja 0..M.
// Uredi jih s štetjem in v „S“ vpiše i-je od 0 do n − 1, urejene po V[i].
void Uredi(vector<int> V, int M, vector<int> &S)
{
  int n = (int) V.size(); S.resize(n);

  // Preštejmo, kolikokrat se pojavi posamezna vrednost.
  vector<int> hist(M + 1, 0);
  for (int v : V) ++hist[v];

  // Izračunajmo, kje se bodo začele posamezne vrednosti v urejenem zaporedju.
  for (int v = 0, vsota = 0; v <= M; ++v) {
    int hv = hist[v]; hist[v] = vsota; vsota += hv; }

  // Vpišimo vrednosti v zaporedje v urejenem vrstnem redu.
  for (int i = 0; i < n; ++i) S[hist[V[i]]++] = i;
}

int main()
{
  ios_base::sync_with_stdio(false);
  int w, h; cin >> w >> h;

  // Preberimo vhodne podatke in preštejmo enice v vsaki vrstici
  // ter ničle v vsakem stolpcu.
  vector<int> stEnic(h, 0), stNicel(w, 0);
  for (int y = 0; y < h; ++y) {
    string s; cin >> s;
    for (int x = 0; x < w; ++x) {
      ++(s[x] == '0' ? stNicel[x] : stEnic[y]); } }

  // Uredimo vrstice po številu enic, stolpce pa po številu ničel.
  vector<int> vrstice, stolpci;
  Uredi(stEnic, w, vrstice); Uredi(stNicel, h, stolpci);

  // Brišimo v mislih vrstice samih enic in stolpce samih ničel, dokler je to mogoče.
  vector<pair<int, int>> operacije;
  while (w > 0 && h > 0)
  {
    int x = stolpci[w − 1], y = vrstice[h − 1];
    if (stEnic[y] == w) operacije.emplace_back(1, y), −−h;

    // Če ni izpolnjen pogoj v prejšnji vrstici, mora tukaj veljati stNicel[x] == h.
    // Če ne bi veljalo niti to, bi bil problem nerešljiv, vendar naloga obljublja,
    // da se to pri testnih primerih na našem tekmovanju ne bo dogajalo.
    else operacije.emplace_back(0, x), −−w;
  }

  // Izpišimo operacije, vendar imejmo v mislih, da smo jih dobili v obratnem vrstnem redu.
  cout << operacije.size() << endl;
  for (int i = int(operacije.size()) − 1; i >= 0; −−i)
```

[4] Če izpisujemo rezultate s cout (namesto npr. s printf), pazimo na naslednjo podrobnost: ker bo mogoče treba izpisati veliko vrstic, uporabimo za konec vrstice '\n' namesto endl — slednji namreč kliče tudi flush, zato je precej počasnejši in bi prekoračili časovno omejitev.

```
    cout << operacije[i].first << ' ' << operacije[i].second + 1 << '\n';
  return 0;
}
```

O pravilnosti naše rešitve se lahko prepričamo s pomočjo naslednje leme. Rekli bomo, da je tabela *rešljiva*, če jo je mogoče dobiti iz tabele ničel z nič ali več operacijami na vrsticah in stolpcih, kakršne dopušča naša naloga.

Lema: recimo, da T je tabela velikosti $w \times h$, zapolnjena z ničlami in enicami, in da ima v vrstici y same enice. Naj bo T' tabela velikosti $w \times (h-1)$, ki jo dobimo, če v T pobrišemo vrstico y. Potem je T rešljiva natanko tedaj, ko je rešljiva T'.

Dokaz. ($\Leftarrow$) Recimo, da je T' rešljiva; mislimo si zaporedje operacij, ki iz začetne tabele samih ničel naredi T'. Skozi to zaporedje operacij nam nastane zaporedje tabel $T_0, \ldots, T_k$ (vse velikosti $w \times (h-1)$), kjer je T_0 tabela ničel in $T_k = T'$. Recimo, da bi namesto tega začeli s tabelo ničel $\hat{T}_0$ velikosti $n \times m$ in izvedli enako zaporedje operacij, le s to spremembo, da kjer smo prej naredili operacijo na vrstici y' za $y' \geq y$, zdaj namesto tega naredimo operacijo na vrstici $y'+1$. Dobimo zaporedje tabel $\hat{T}_0, \ldots, \hat{T}_k$ (vse so velikosti $w \times h$). Z indukcijo po i se lahko prepričamo, da če v tabeli $\hat{T}_i$ pobrišemo vrstico y, dobimo ravno tabelo T_i. Iz tega potem sledi, da če v $\hat{T}_k$ pobrišemo vrstico y, dobimo ravno $T_k = T'$. Po predpostavki naše leme pa tudi za T velja, da če v njej pobrišemo vrstico y, dobimo ravno T'. Torej se T in $\hat{T}_k$ razlikujeta le v vrstici y, tam pa ima T (po predpostavki) same enice. Če torej zdaj na tabeli $\hat{T}_k$ izvedemo še operacijo na vrstici y, bomo dobili ravno tabelo T, ki je torej tudi rešljiva.

($\Rightarrow$) Recimo, da je T rešljiva, torej jo je mogoče dobiti iz tabele ničel z nekim zaporedjem operacij. Na koncu so v vrstici y same enice; mislimo si v našem zaporedju zadnjo tako operacijo, ki vpiše kakšno enico v vrstico y. Če bi kdaj kasneje izvedli še kakšno operacijo na nekem stolpcu x, bi ta v $T(x, y)$ vpisal ničlo, ki bi tam potem ostala do konca (saj smo predpostavili, da je bila zadnja operacija, ki v vrstico y vpiše kakšno enico, izvedena že prej), torej v resnici ne bi dobili tabele T; to se torej ne more zgoditi. Po zadnji operaciji na vrstici y zato lahko nastopi še več drugih operacij na vrsticah, gotovo pa ne nastopi nobena operacija na stolpcih. Več zaporednih operacij na vrsticah pa lahko poljubno premešamo, ne da bi se končni rezultat kaj spremenil; naše zaporedje lahko torej predelamo tako, da operacija na vrstici y nastopi čisto na koncu. Poleg tega, če je bila kakšna operacija na vrstici y izvedena tudi bolj zgodaj v zaporedju, jo lahko pobrišemo, saj ni od nje nobene koristi (karkoli se je bolj zgodaj dogajalo na vrstici y, bo to tako ali tako povozila operacija na vrstici y v zadnjem koraku zaporedja).

Recimo, da je naše zaporedje dolgo $k+1$ operacij, in naj bo $\hat{T}_i$ stanje tabele po prvih i operacijah; začne se s tabelo ničel $\hat{T}_0$, konča pa s $\hat{T}_{k+1} = T$. Za vsak i definirajmo T_i kot tabelo, ki nastane, če v $\hat{T}_i$ pobrišemo vrstico y. Ker je bila zadnja operacija na vrstici y, se T (oz. $\hat{T}_{k+1}$) in $\hat{T}_k$ razlikujeta le v tej vrstici; če jo pobrišemo, dobimo torej obakrat enako tabelo; po definiciji dobimo z brisanjem vrstice y iz T tabelo T', iz $\hat{T}_k$ pa T_k, torej je $T_k = T'$. Če zdaj v našem zaporedju operacij odmislimo zadnji korak (na vrstici y) in prejšnje korake spremenimo le v tem, da kjer smo prej izvedli operacijo na kakšni vrstici y' za $y' > y$, jo zdaj izvedemo na vrstici $y'-1$, se lahko spet z indukcijo po i prepričamo, da po prvih i korakih v novem zaporedju (ki se začne s tabelo ničel T_0 velikosti $w \times (h-1)$) dobimo ravno tabelo T_i, po k korakih torej $T_k = T'$, torej je tabela T' tudi rešljiva. □

V gornji lemi smo razmišljali o primerih, ko ima T neko vrstico samih enic. Če bi imela T neki stolpec samih ničel, bi bil razmislek analogen, le da bi morali pač pobrisati tisti stolpec. Če pa T nima niti vrstice samih enic niti stolpca samih ničel, je gotovo nerešljiva, kajti če bi bila rešljiva, bi zadnja operacija v zaporedju, s katerim to tabelo dobimo, pustila v njej bodisi neko vrstico samih enic (če gre za operacijo na vrstici) bodisi neki stolpec samih ničel (če gre za operacijo na stolpcu).

Do precej podobne rešitve lahko pridemo tudi z malo drugačnim razmislekom. Nobene koristi ni od tega, da bi dvakrat (ali večkrat) izvedli operacijo na isti vrstici (ali stolpcu), kajti v tem primeru bi druga (kasnejša) od teh operacij tako ali tako povozila vse, kar je bilo v tisto vrstico (ali stolpec) vpisano pred njo, torej tudi to, kar je vpisala prva (zgodnejša) operacija. Potrebovali bomo torej kvečjemu $w+h$ operacij, po eno za vsako vrstico in vsak stolpec.

Po drugi strani ni nobene škode, če izvedemo točno toliko operacij in ne manj. Recimo namreč, da je mogoče vhodno tabelo T dobiti že z manj operacijami. Če vse „manjkajoče“ operacije vrinemo na začetek tega zaporedja operacij, in sicer najprej tiste na vrsticah in nato tiste na stolpcih, bomo še vedno dobili tabelo T.[5]

Ta razmislek nam je pokazal, da se smemo omejiti na rešitve, ki jih sestavlja *natanko* $w + h$ operacij, po ena za vsako vrstico in vsak stolpec. Vprašanje je le, v kakšnem vrstnem redu naj jih izvedemo. Glede tega pa nam vsaka celica tabele T določa naslednjo omejitev: v celico (x, y) pišeta le dve operaciji, namreč tista na vrstici y (ki vpiše enico) in tista na stolpcu x (ki vpiše ničlo). Če ima torej (x, y) v tabeli T vrednost 0, mora operacija na stolpcu x nastopiti kasneje kot tista na stolpcu y, če pa ima omenjena celica vrednost 1, je ravno obratno. Tako dobljene omejitve so ne le potreben pogoj za to, da nam zaporedje operacij dá tabelo T, ampak tudi zadosten; z drugimi besedami, vsak vrstni red operacij, ki ustreza vsem tem omejitvam, nas bo pripeljal do tabele T.[6]

Lahko si predstavljamo usmerjen graf z $w + h$ točkami, ki predstavljajo vse možne operacije, recimo $\{(0, x) : 1 \leq x \leq w\}$ za stolpce in $\{(1, y) : 1 \leq y \leq h\}$ za vrstice. Za vsako celico (x, y) tabele T dodajmo v graf povezavo med točkama $(0, x)$ in $(1, y)$, smer te povezave pa določimo tako, da kaže iz tiste operacije, ki mora biti zgodnejša, v tisto, ki mora biti kasnejša. Če je torej $T[x, y] = 0$, imejmo povezavo $(1, y) \to (0, x)$, če pa je $T[x, y] = 1$, imejmo povezavo $(0, x) \to (1, y)$.

Vsaka povezava grafa torej predstavlja neko omejitev glede vrstnega reda operacij. Če je v tem grafu kak cikel, je vsem omejitvam hkrati očitno nemogoče ustreči, na srečo pa naloga obljublja, da se pri naših testnih primerih to ne bo zgodilo. Graf bo torej acikličen, lahko ga topološko uredimo in tako dobimo primeren vrstni red operacij.

Spomnimo se, kako poteka topološko urejanje grafa: najprej izračunamo vhodne stopnje točk, nato dodamo v topološki vrstni red (ki je sprva prazen) točke z vhodno stopnjo 0, nato pa te točke (in njihove izhodne povezave) pobrišemo iz grafa. S tem se je zmanjšala vhodna stopnja nekaterih drugih točk in postopek ponavljamo, dokler graf ni prazen. Pri našem grafu je vhodna stopnja točke $(0, x)$ ravno število ničel v stolpcu x tabele T, vhodna stopnja točke $(1, y)$ pa je število ničel v vrstici y tabele T. To pa so prav ista števila, na katera se je opirala tudi naša prva rešitev, le da je ona gradila zaporedje operacij od konca proti začetku (v mislih smo iz tabele T najprej pobrisali stolpce z največ ničlami oz. vrstice z največ enicami), tu pri topološkem urejanju pa ga gradimo od začetka proti koncu (in vanj najprej dodamo stolpce z najmanj ničlami oz. vrstice z najmanj enicami).

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

int main()
{
```

[5] O tem se lahko prepričamo s protislovjem. Pa recimo, da ima po opisani dopolnitvi zaporedja z manjkajočimi operacijami neka celica (x, y) drugačno vrednost kot prej. Očitno torej v prvotnem zaporedju operacij ni bilo niti take na vrstici y niti take na stolpcu x, sicer bi dobila celica (x, y) v dopolnjenem zaporedju še vedno enako vrednost kot v prvotnem (ker smo manjkajoče operacije vrinili na začetek zaporedja in četudi kakšna od njih spremeni celico (x, y), jo bodo kasneje še vedno povozile operacije iz prvotnega zaporedja). Celica (x, y) mora biti torej ena od tistih, ki se jih prvotno zaporedje operacij sploh ne dotakne, in (x, y) ima v T očitno še vedno svojo prvotno vrednost 0. Videli smo, da v prvotnem zaporedju ni niti operacije na vrstici y niti na stolpcu x, torej sta tidve operaciji eni od tistih, ki smo jih pri dopolnitvi zaporedja vrinili na začetek; in spomnimo se, da smo vrinili najprej operacije na vrsticah in za njimi tiste na stolpcih. V dopolnjenem zaporedju je torej zadnja operacija, ki piše v celico (x, y), tista na stolpcu x; operacije na stolpcih pa vpisujejo v celice ničle, torej ima (x, y) pri dopolnjenem zaporedju na koncu vrednost 0, to pa je enaka vrednost kot pri prvotnem zaporedju. Tako smo prišli v protislovje s predpostavko, da dobi (x, y) pri dopolnjenem zaporedju operacij drugačno vrednost kot pri prvotnem. □

[6] Pa recimo, da nas neko zaporedje operacij, ki ustreza vsem tem omejitvam, ne pripelje do tabele T; obstaja torej neka celica (x, y), ki ima na koncu drugačno vrednost kot v tabeli T. Če je na primer $T[x, y] = 0$, ima pri našem hipotetičnem zaporedju operacij celica (x, y) na koncu očitno vrednost 1; torej je zadnja operacija, ki je pisala vanjo, tista na vrstici y, ki se je torej izvedla kasneje kot operacija na stolpcu x. Toda ker je $T[x, y] = 0$, je ena od naših omejitev zahtevala, da mora operacija na stolpcu x nastopiti kasneje kot tista na vrstici y; tako smo prišli v protislovje s predpostavko, da naše zaporedje operacij ustreza vsem omejitvam. Razmislek za primer, ko je $T[x, y] = 1$, je seveda čisto podoben. □

```
  int w, h; cin >> w >> h;
  // Točke od 0 do w − 1 predstavljajo stolpce, točke od w do w + h − 1 pa vrstice.
  // Preberimo vhodne podatke in določimo stopnje točk.
  vector<string> T(h); vector<int> stopnja(w + h, 0);
  for (int y = 0; y < h; ++y) {
    cin >> T[y];
    for (int x = 0; x < w; ++x) {
      ++stopnja[T[y][x] == '0' ? x : w + y]; } }
  // Točke s stopnjo 0 dodajmo v topološki vrstni red.
  vector<int> vrstniRed; int glava = 0;
  for (int u = 0; u < w + h; ++u) if (stopnja[u] == 0) vrstniRed.emplace_back(u);
  // Obdelajmo preostanek grafa.
  while (glava < (int) vrstniRed.size()) {
    int u = vrstniRed[glava++];
    // Ker smo u dodali v topološki red, bomo zdaj v mislih pobrisali to točko in
    // njene izhodne povezave. Nekaterim drugim točkam se zato vhodna stopnja
    // zmanjša in če pade na 0, moramo tudi njih dodati v topološki vrstni red.
    if (u < w) { // u predstavlja stolpec in kaže na tiste vrstice, kjer so v tem stolpcu enice.
      for (int x = u, y = 0; y < h; ++y) if (T[y][x] == '1')
        if (−−stopnja[w + y] == 0) vrstniRed.emplace_back(w + y); }
    else { // u predstavlja vrstico in kaže na tiste stolpce, kjer so v tej vrstici ničle.
      for (int x = 0, y = u − w; x < w; ++x) if (T[y][x] == '0')
        if (−−stopnja[x] == 0) vrstniRed.emplace_back(x); } }
  // Izpišimo rezultate.
  cout << vrstniRed.size() << endl; // To bi moralo biti enako w + h, sicer je problem
    // nerešljiv (vendar besedilo naloge obljublja, da se to pri testnih primerih
    // na tekmovanju ne bo dogajalo).
  for (int u : vrstniRed) cout << (u < w ? 0 : 1) << ' ' << 1 + (u < w ? u : u − w) << '\n';
  return 0;
}
```

Manjša slabost te rešitve v primerjavi s prejšnjo je, da si mora ta izhodno tabelo shraniti v pomnilniku (ker se iz nje vidi, katere izhodne povezave kažejo iz posamezne točke našega grafa), prejšnja rešitev pa jo je samo prebrala, v pomnilniku pa je hranila le število enic v vsaki vrstici in število ničel v vsakem stolpcu.

4. Lepe podmnožice

Rešitev s časovno zahtevnostjo $O(n \cdot b)$. Poti, ki nobenega vozlišča ne obišče več kot enkrat, pravimo včasih tudi *preprosta pot*. Pri tej nalogi (in njeni rešitvi) se bomo ukvarjali le s takšnimi potmi. Pri preprosti poti velja, da ko se enkrat spustimo iz nekega vozlišča v nekega njegovega otroka, se bomo odtlej lahko tudi v prihodnje samo še spuščali; če bi se hoteli spet vzpeti, bi morali iti nazaj v starša, iz katerega smo tik pred tem prišli, tega pa ne smemo. Pot se torej sprva nekaj časa (0 ali več korakov) vzpenja in nato nekaj časa (0 ali več korakov) spušča.

Označimo s $T(u)$ poddrevo, ki ga tvorijo vsi potomci vozlišča u (vključno z u-jem samim).

Recimo, da nas zanimajo lepe podmnožice barve c. Naloga pravi, da se mora dati vsa vozlišča lepe podmnožice obiskati s preprosto potjo; nalogo si lahko torej predstavljamo tudi tako, da namesto da iščemo največjo lepo podmnožico barve c, iščemo (preprosto) pot, ki obišče največ točk barve c. Taka pot pa se, kot smo videli, nekaj časa vzpenja, doseže v neki točki u svoj vrh in se od tam naprej spušča. Recimo, da ima u otroke $v_1, \ldots, v_k$. Pot, o kateri razmišljamo, se torej najprej nekaj časa vzpenja po enem od poddreves $T(v_i)$, nato se vzpne iz v_i v u, se od tam spusti v nekega drugega u-jevega otroka, recimo v_j, in se potem še nekaj časa spušča po $T(v_j)$. Tisti del poti, ki poteka znotraj $T(v_i)$, je torej „navpičen“: poteka med v_i in nekim v_i-jevim potomcem; in podobno je pri $T(v_j)$.

Vidimo torej, da bo koristno, če bomo znali za vsako vozlišče v izračunati največje število vozlišč barve c, ki jih lahko obišče navpična pot, ki se spušča iz v do nekega njegovega potomca; temu številu recimo $f_c(v)$. Tega ni težko računati s preprostim

rekurzivnim razmislekom: če je v list, je $f_c(v)$ enak 1 ali 0, odvisno od tega, ali je v barve c ali ne. Če pa je v notranje vozlišče, dobimo $f_c(v)$ tako, da vrednosti 1 ali 0 (odvisno od tega, ali je v barve c ali ne) prištejemo še maksimum $f_c(w)$ po vseh v-jevih otrocih w.

Ko poznamo vrednosti $f_c(\cdot)$, ni težko za posamezno vozlišče u (z otroki $v_1, \ldots, v_k$) izračunati največjega števila vozlišč barve c, ki jih lahko pokrije pot z vrhom v u: en krak te poti jih pokrije $f_c(v_i)$, drugi krak jih pokrije $f_c(v_j)$, za v_i in v_j pa moramo seveda izbrati tista dva u-jeva otroka, ki imata največjo vrednost $f_c(\cdot)$. Vsoti teh dveh vrednosti moramo, če je u tudi sam barve c, seveda prišteti še 1. Tak razmislek opravimo pri vsakem u in si zapomnimo najboljšo od tako dobljenih rešitev. Zapišimo ta postopek s psevdokodo:

```
globalna spremenljivka: r[c] = velikost največje doslej znane
  lepe podmnožice barve c (na začetku 0);

(* Naslednja funkcija vrne f_c(u) in popravi r[c]. *)
funkcija ObdelajPoddrevo(c, u):
  F_1 := 0; F_2 := 0;
  za vsakega u-jevega otroka v:
    F := ObdelajPoddrevo(c, v);
    if F > F_1 then F_2 := F_1,  F_1 := F
    else if F > F_2 then F_2 := F;
  if je u barve c then p := 1 else p := 0;
  r[c] := max{r[c], F_1 + p + F_2};
  return F_1 + p;
```

Če poženemo to funkcijo na korenu drevesa, bo sčasoma pregledala celotno drevo, izračunala vse $f_c(u)$ in na koncu v $r[c]$ odložila velikost največje lepe podmnožice barve c; za to bo porabila $O(n)$ časa. Lahko bi jo ovili še v eno zanko po vseh barvah in tako rešili problem v $O(n \cdot b)$ časa. Na našem tekmovanju bi bilo to dovolj le za 50 % točk, zato razmislimo, kako to rešitev še izboljšati.

Rešitev s časovno zahtevnostjo $O(n \log n + b)$. Naša dosedanja rešitev bi pri vsakem vozlišču u izračunala vrednosti $f_c(u)$ za vse barve $c = 1, \ldots, b$. Bolje pa bi bilo, če bi pri posameznem u računali te vrednosti le za tiste barve c, ki se sploh pojavljajo v u-jevem poddrevesu, saj za ostale velja $f_c(u) = 0$. Naš podprogram za obdelavo u-jevega poddrevesa naj torej vrne vse te vrednosti, zložene v slovar H_u (oz. razpršeno tabelo; v C++ bomo uporabili razred `unordered_map` iz standardne knjižnice), v katerem bodo ključi barve c (le tiste, ki se pojavljajo v u-jevem poddrevesu), pripadajoči podatki pa bodo vrednosti $f_c(u)$.[7]

```
globalna spremenljivka: r[c] = velikost največje doslej znane
    lepe podmnožice barve c (na začetku 0);

(* Naslednja funkcija vrne H_u in popravi r[c]. *)
funkcija ObdelajPoddrevo2(u):
 1  H := prazen slovar;
 2  za vsakega u-jevega otroka v:
 3    H' := ObdelajPoddrevo(v);
 4    če je H' večji od H, ju med seboj zamenjaj;
 5    za vsako barvo c iz H', s pripadajočo vrednostjo H'[c]:
 6      if je u barve c then p := 1 else p := 0;
 7      r[c] := max{r[c], H[c] + p + H'[c]};
 8      if H'[c] > H[c] then H[c] := H'[c];
 9  naj bo c barva vozlišča u;  H[c] := H[c] + 1;
10  return H;
```

[7] V psevdokodi, ki sledi, uporabljamo oznako $H[c]$ za spremljevalni podatek pri ključu c v slovarju H; če ključa c ni v H, si mislimo, da je $H[c] = 0$; prireditev $H[c] := x$ pomeni, da če c-ja še ni v H, ga zdaj tja dodamo s pripadajočo vrednostjo x, če pa je bil c že od prej v H, mu le postavimo pripadajočo vrednost na x.

Podprogram ObdelajPoddrevo2 moramo pognati pri korenu drevesa in ko se ta klic vrne, bodo v $r[\cdot]$ že zapisani vsi rezultati, ki nas zanimajo.

Oglejmo si pobliže, kako ta podprogram deluje. V zanki gremo po u-jevih otrocih, pri tem pa v H vzdržujemo slovar, kjer so ključi vse barve, ki nastopajo v poddrevesih doslej pregledanih u-jevih otrok, pripadajoče vrednosti $H[c]$ pa so maksimumi $f_c(w)$ po vseh doslej pregledanih u-jevih otrocih w. Ko za naslednjega otroka, v, z rekurzivnim klicem izračunamo slovar H_v (vrstica 4), je treba s podatki iz njega dopolniti slovar H (vrstica 8): če je kakšna barva c prisotna v H_v, ne pa tudi v H, jo je treba v H dodati; če pa je že prisotna v H, vendar z manjšo spremljevalno vrednostjo kot v H', je treba v H vpisati to večjo spremljevalno vrednost iz H'-ja. Ko bomo na ta način pregledali vse u-jeve otroke, bomo imeli v H ravno slovar H_u; le pri barvi u-ja samega je treba spremljevalno vrednost povečati za 1 (vrstica 9), ker navpična pot dol iz u pokrije vozlišče u, ki je barve c in ki ga ne pokrije pot navzdol iz u-jevega otroka.

Poleg tega lahko, ko pregledujemo slovar H', tudi popravljamo rezultate (velikosti največjih doslej znanih lepih podmnožic za posamezne barve) v globalni spremeljivki r (vrstica 7): če je mogoče v $T(v)$ z neko navpično potjo pokriti $H'[c]$ vozlišč barve c, v nekem drugem od doslej pregledanih u-jevih otrok pa $H[c]$ vozlišč barve c, potem imamo lepo podmnožico (barve c) s $H[c] + H'[c]$ vozlišči (ali pa še z enim več, če je tudi u te barve). Ta pristop je na prvi pogled malenkost drugačen kot tisti v prejšnji rešitvi (ObdelajPoddrevo), vendar se hitro vidi, da bo tudi deloval: tam smo pogledali, v katerih dveh u-jevih otrocih je mogoče z navpično potjo pokriti največ vozlišč barve c, in smo tadva prispevka sešteli; recimo, da sta bila to otroka x in y; pri naši novi rešitvi bo naša zanka po u-jevih otrocih prej ali slej obdelala tudi x in y; enega od njiju bo obdelala prvega — recimo brez izgube za splošnost, da je to x; in to pomeni, da ko bo naša zanka prišla do y, bo imela v $H[c]$ vrednost $f_x(c)$, v $H'[c]$ pa vrednost $f_y(c)$, torej bo $H[c] + H'[c]$ (plus 1, če je u tudi sam barve c) ravno rezultat, ki bi ga dobil tudi stari ObdelajPoddrevo. Tako vidimo, da ne bomo spregledali nobene rešitve.

Pomembna podrobnost pri naši novi rešitvi je, da slovarja H in H' po potrebi zamenjamo, tako da vedno pregledujemo manjšega od njiju (vrstica 4). Brez tega bi se namreč lahko pri $O(n)$ vozliščih zgodilo, da bi morali pregledati slovar velikosti $O(b)$, in naša rešitev bi imela spet časovno zahtevnost $O(n \cdot b)$. Razmislimo, kaj se izboljša zaradi tega, ker smo uporabili pristop zlivanja manjše množice (oz. slovarja) v večjo. Vsak od slovarjev H in H' vsebuje kot ključe množico vseh barv, ki se pojavljajo v poddrevesu $T(v)$ ali pa v uniji nekaj disjunktnih poddreves (namreč poddreves vseh doslej obdelanih u-jevih otrok razen v-ja); recimo tej uniji U. Slovar, ki predstavlja množico vozlišč $T(v)$ ali U, vsebuje kvečjemu toliko ključev, kolikor je vozlišč v množici; cena zlivanja (število iteracij zanke v vrsticah 5–8) pa je enaka številu ključev v manjšem od obeh slovarjev in je zato navzgor omejena s številom vozlišč v manjši od obeh množic. Po zlitju slovarja H' v H si lahko predstavljamo, da je H' zdaj prazen oz. ne obstaja več, slovar H pa zdaj predstavlja množico $T(v) \cup U$, katere velikost (število vozlišč) je vsota velikosti množic $T(v)$ in U, kajti slednji dve sta disjunktni; množica $T(v) \cup U$ ima torej vsaj dvakrat toliko elementov kot manjša izmed $T(v)$ in U.

Recimo, da bi v mislih za vsako vozlišče drevesa vzdrževali neki števec, ki je na začetku pri vseh vozliščih enak 0; ko pa zlijemo slovar H' v H, povečajmo v mislih za 1 vrednost števca pri vseh vozliščih v manjši izmed množic $T(v)$ in U. Skupno povečanje števcev je tako večje ali enako od števila iteracij zanke v vrsticah 5–8 pri zlivanju H' v H. Na koncu, ko se postopek konča, je vsota števcev (po vseh vozliščih v drevesu) zgornja meja za skupno število vseh izvedenih iteracij in s tem za časovno zahtevnost celotnega postopka. Kakšna pa je na koncu ta vsota števcev? Ko se števec nekemu vozlišču x poveča za 1, je to zato, ker je bil slovar H', ki je doslej pokrival x, zlit v neki slovar H, ki je pokrival še večjo množico vozlišč, in vozlišče x je zdaj pokrito v zlitem slovarju (ki je še vedno v H), ki (kot smo videli na koncu prejšnjega odstavka) pokriva množico vozlišč $T(v) \cup U$, ki je dvakrat tolikšna kot tista (bodisi $T(v)$ bodisi U), ki jo je pokrival slovar H' pred zlitjem. Vsakič torej, ko se vozlišču x poveča števec za 1, se to vozlišče znajde v slovarju, ki pokriva vsaj dvakrat tolikšno množico vozlišč kot slovar, v katerem je bil x pred povečanjem števca. Ker pa imamo le n vozlišč, lahko pride do takšne podvojitve le $(\log_2 n)$-krat. Vsakemu vozlišču se torej lahko števec poveča največ do $\log_2 n$ in na koncu je vsota števcev po vseh vozliščih kvečjemu $n \log_2 n$. Tako

vidimo, da ima naš postopek časovno zahtevnost $O(n \log n)$, čemur pa moramo seveda prišteti še $O(b)$ za inicializacijo tabele $r[\cdot]$ in izpis rezultatov. Ta rešitev je torej veliko učinkovitejša od prejšnje; na tekmovanju bi pri tej nalogi dobila 80 % točk.

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

typedef int StVozlisca, StBarve;
struct Vozlisce {
  StVozlisca otrok = −1, sor = −1; // prvi otrok, naslednji sorojenec
  StBarve barva; };
int n, b; vector<Vozlisce> vozl;
vector<int> rezultati; // Največja lepa podmnožica posamezne barve.

// Vrne slovar, v katerem za vsako barvo, ki se pojavlja v u-jevem poddrevesu,
// piše, največ koliko vozlišč tiste barve lahko pokrije pot, ki gre iz „u“ navzdol.
// Poleg tega tudi dopolni globalno spremenljivko „rezultati“ z največjimi
// lepimi podmnožicami, ki ležijo znotraj u-jevega poddrevesa.
unordered_map<StBarve, int> ObdelajPoddrevo(StVozlisca u)
{
  auto &U = vozl[u]; unordered_map<StBarve, int> uMaxNaVeji;
  for (StVozlisca v = U.otrok; v >= 0; v = vozl[v].sor) {
    auto vMaxNaVeji = ObdelajPoddrevo(v);
    // Manjšega od obeh slovarjev bomo zlili v večjega.
    if (vMaxNaVeji.size() > uMaxNaVeji.size()) swap(uMaxNaVeji, vMaxNaVeji);
    for (auto [c, rv] : vMaxNaVeji) {
      auto &ru = uMaxNaVeji.emplace(c, 0).first->second;
      // Koliko točk barve c lahko pokrije pot z vrhom pri u, če se en krak poti
      // spusti skozi v, drugi pa skozi enega od že prej obravnavanih u-jevih otrok?
      rezultati[c] = max(rezultati[c], ru + (U.barva == c ? 1 : 0) + rv);
      ru = max(ru, rv); } }
  // Povečajmo uMaxNaVeji[U.barva] za 1.
  auto pr = uMaxNaVeji.emplace(U.barva, 1);
  if (! pr.second) ++pr.first->second;
  // Naslednja vrstica pride prav le, če je „u“ list.
  rezultati[U.barva] = max(rezultati[U.barva], pr.first->second);
  return uMaxNaVeji;
}

int main()
{
  // Preberimo vhodne podatke.
  ios_base::sync_with_stdio(false);
  int n, b; cin >> n >> b;
  vozl.resize(n); rezultati.resize(b, 0); StVozlisca koren = −1;
  for (int u = 0; u < n; ++u) {
    auto &U = vozl[u]; StVozlisca p;
    cin >> p >> U.barva; −−U.barva;
    if (−−p < 0) koren = u;
    else { auto &P = vozl[p]; U.sor = P.otrok; P.otrok = u; } }
  // Rešimo nalogo za vsa poddrevesa.
  ObdelajPoddrevo(koren);
  // Izpišimo rezultate.
  for (int r : rezultati) cout << r << '\n';
  return 0;
}
```

Podpore za velika drevesa (z več kot 200 000 vozlišči), opisana parametrično, v gornji program nismo dodali, ker največjih takih testnih primerov tako ali tako ne bi mogel

rešiti dovolj hitro (pri naših poskusih je zmogel na ocenjevalnem strežniku znotraj časovne omejitve rešiti primere z do tremi milijoni vozlišč, morda tremi in pol; največji testni primeri, ki smo jih zares uporabili na tekmovanju, pa so imeli od štiri in pol do pet milijonov vozlišč).

Rešitev s časovno zahtevnostjo $O(n + b)$. Vrnimo se v mislih spet k naši prvi rešitvi, vendar recimo, da nas zanima največja lepa podmnožica le za eno konkretno barvo, na primer c. Če neko vozlišče u ni barve c in ima starša v, ki tudi ni barve c, lahko u pobrišemo, bivši u-jevi otroci pa naj pri tem postanejo v-jevi otroci (medtem ko so prej bili v-jevi vnuki), pa se ne bo pri lepih podmnožicah barve c zaradi tega nič spremenilo; vsaka podmnožica barve c, ki je bila lepa prej, je lepa tudi zdaj in obratno.

Prepričajmo se, da je to res. ($\Rightarrow$) Recimo, da je bila A prej lepa, torej je obstajala neka pot, ki je obiskala vsa njena vozlišča. Če ta pot ni šla skozi u, obstaja še zdaj in A je še vedno lepa. Če pa je ta pot šla skozi u, ločimo nekaj primerov: (1) če se je pot v u začela ali končala, bo ta pot tudi brez u-ja še vedno obiskala vsa vozlišča iz A, torej je A še vedno lepa. (2) Če u na poti leži med dvema svojima otrokoma, recimo x in y, sta torej na stari poti bila koraka $x \to u \to y$; stara pot torej gotovo ni šla skozi v, ker bi potem morala u obiskati vsaj dvakrat; po brisanju lahko na poti koraka $x \to u \to y$ zamenjamo z $x \to v \to y$, pa je tako popravljena pot spet veljavna in preprosta in obišče vsa vozlišča A-ja, torej je A še vedno lepa. (3) Če pa u na poti leži med nekim svojim otrokom x in svojim staršem v, lahko namesto korakov $x \to u \to v$ po novem naredimo samo $x \to v$ in pot bo spet veljavna, preprosta in obiskala bo vsa vozlišča A-ja, torej je A še vedno lepa.

($\Leftarrow$) Recimo, da je A po brisanju vozlišča u lepa, torej obstaja neka pot, ki obišče vsa vozlišča A-ja. (1) Če ta pot nikoli ne naredi koraka med v in kakšnim bivšim u-jevim otrokom (ki je zdaj postal v-jev otrok), potem je ta pot obstajala že pred brisanjem u-ja in je bila torej A lepa tudi takrat. (2) Če pa ta pot nekoč naredi korak $v \to x$ za nekega bivšega u-jevega otroka x, bi lahko ta korak zamenjali z $v \to u \to x$ in dobili veljavno pot v starem drevesu (pred brisanjem u-ja), ki bi obiskala vsa vozlišča A-ja, torej je bila A lepa že takrat. □

S podobnim, a še preprostejšim razmislekom se lahko prepričamo tudi, da če neko vozlišče u ni barve c in nima nobenega potomca barve c, lahko to vozlišče in vse njegove potomce pobrišemo, pa ne bo to nič vplivalo na lepe podmnožice barve c; podrobnosti prepustimo bralcu za vajo.

Vidimo torej, da bi se dalo drevo precej skrčiti, ne da bi se velikost največje lepe podmnožice barve c kaj spremenila. Skrčenemu drevesu, ki ga dobimo, če pobrišemo vsa vozlišča na podlagi dosedanjih opažanj, recimo T_c. Koliko vozlišč pa ima to drevo? Recimo, da je bilo v prvotnem drevesu n_c vozlišč barve c; vsa ta so seveda še vedno prisotna tudi v T_c. Poleg njih je morda v T_c tudi nekaj vozlišč, ki niso barve c; toda če neki $u \in T_c$ ni barve c, potem ima gotovo vsaj enega otroka (če ne, bi tak u lahko pobrisali) in vsi njegovi otroci so gotovo barve c (drugače bi lahko pobrisali tiste njegove otroke, ki niso barve c). Na vsako vozlišče (recimo x) barve c lahko pride torej še kvečjemu eno vozlišče, ki ni barve c in je prisotno v T_c kot x-ov starš. Tako ima torej T_c kvečjemu $2n_c$ vozlišč. Če zanj poženemo postopek ObdelajPoddrevo iz naše prve rešitve, bomo v $O(n_c)$ časa dobili največjo lepo podmnožico barve c. Če to seštejemo po vseh barvah in upoštevamo, da je $\sum_{c=1}^{b} n_c = n$, vidimo, da lahko lepe podmnožice za vse barve poiščemo v $O(n)$ časa.

Razmisliti je treba le še o tem, kako dovolj hitro pripraviti drevesa T_c za vse barve. Označimo s $P[u]$ starša vozlišča u v drevesu T_c, kjer je c barva vozlišča u. (1) Če je u-jev starš v prvotnem drevesu (pred brisanjem) tudi sam barve c, bo ta starš ostal tudi v T_c in bo tudi tam u-jev starš. (2) Sicer pojdimo v mislih po u-jevih prednikih (v prvotnem drevesu); (2.1) če ni nobeden od njih barve c, bomo pobrisali vse razen zadnjega, to je korena prvotnega drevesa, ki bo ostal koren tudi v T_c; tedaj je torej $P[u]$ ravno koren drevesa. (2.2) Sicer pa recimo, da prvih nekaj u-jevih prednikov, npr. $p_1, \ldots, p_\ell$, ni barve c, naslednji prednik $p_{\ell+1}$ pa je barve c. Potem bomo pobrisali vozlišča $p_1, \ldots, p_{\ell-1}$, ne pa tudi vozlišča p_ℓ; slednje bo zato postalo u-jev starš v T_c.

Vidimo torej, da bi lahko $P[\cdot]$ vseh vozlišč določili z rekurzivnim postopkom navzdol po drevesu. Pri tem bomo vzdrževali pot L od korena do trenutnega vozlišča (to so torej predniki trenutnega vozlišča) in še tabelo Z, ki za vsako barvo c pove najglobljega

prednika (v L) barve c; naslednji prednik pod njim je potem tisti, ki mora postati (v T_c) starš trenutnega vozlišča, če je le-to barve c.

podprogram DoločiStarše(seznam L, vozlišče u, tabela Z):
  vhod: $L = (v_0, \ldots, v_{k-1})$ je pot od korena v_0 do u-jevega starša v_{k-1};
    $Z[c]$ je indeks zadnjega vozlišča barve c v L (ali -1, če takega sploh ni);
  $c :=$ barva vozlišča u;
  if $Z[c] < k - 1$ **then**
    $i := Z[c] + 1$ (* *Pobrisali bomo u-jeve prednike* $v_{k-1}, \ldots, v_{i+1}$*, ne pa* v_i*, ki bo zato postal u-jev starš v* T_c*.* *)
  else
    $i := k - 1$; (* *u-jev starš je barve c in bo ostal u-jev starš tudi v* T_c*.* *)
  $P[u] := v_i$; (* *To bo u-jev starš v* T_c*.* *)
  dodaj u na konec seznama L; $z := Z[c]$; $Z[c] := k$;
  za vsakega u-jevega otroka v:
    DoločiStarše(L, v, Z);
  $Z[c] := z$;

Če ne štejemo dela z vgnezdenimi rekurzivnimi klici, imamo tu z vsakim vozliščem le $O(1)$ dela. Tako lahko torej v $O(n + b)$ časa določimo starše vseh vozlišč (v drevesih T_c) in potem, kot smo videli že prej, v $O(n + b)$ časa tudi zgradimo vsa drevesa T_c in poiščemo v njih največje lepe podmnožice. Tako imamo rešitev s časovno zahtevnostjo le $O(n + b)$, ki je dovolj hitra tudi za največje testne primere na našem tekmovanju.

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

typedef int StVozlisca, StBarve;
struct Vozlisce
{
  StBarve barva;            // Barva tega vozlišča.
  StVozlisca naslIsteBarve = -1; // Naslednje vozlišče iste barve.
  // Starš, prvi otrok in naslednji sorojenec tega vozlišča v vhodnem drevesu.
  StVozlisca stars = -1, otrok = -1, sor = -1;
  StVozlisca starsB = -1; // Starš v skrčenem drevesu za barvo „barva“.
  StBarve dodan = -1;   // Zadnje skrčeno drevo, v katero je bilo to vozlišče dodano.
  // Prvi otrok in naslednji sorojenec v skrčenem drevesu za barvo „dodan“.
  StVozlisca otrokS = -1, sorS = -1;
};
int n, b; StVozlisca koren = -1;
vector<Vozlisce> vozl;

// Naslednji dve spremenljivki uporablja podprogram DolociStarseB.
vector<int> veja; // vozlišča na poti od korena do u
vector<int> zadnjiPoBarvi; // zadnjiPoBarvi[b] je indeks zadnjega vozlišča barve b v „veja“
void DolociStarseB(int u)
{
  auto &U = vozl[u];
  int pi = zadnjiPoBarvi[U.barva]; // Najbližji u-jev prednik enake barve kot u
  int ui = (int) veja.size(); veja.emplace_back(u); // Dodajmo u na konec veje.
  // Če je u-jev starš enake barve kot u, bo ostal njegov starš tudi v skrčenem
  // drevesu u-jeve barve. Sicer pa bo u-jev starš v skrčenem drevesu postal
  // zadnji prednik pred „p“ (najbližjim prednikom enake barve kot u).
  int ri = min(pi + 1, ui - 1); U.starsB = (ri < 0) ? -1 : veja[ri];
  // Rekurzivno obdelajmo še u-jeve potomce.
  zadnjiPoBarvi[U.barva] = ui;
  for (StVozlisca v = U.otrok; v >= 0; v = vozl[v].sor) DolociStarseB(v);
  zadnjiPoBarvi[U.barva] = pi; veja.pop_back(); // Pospravimo za sabo.
```

```
}

StBarve c = −1; // Barva, za katero trenutno obravnavamo skrčeno drevo.

// Vrne par ⟨največja lepa podmnožica barve c, največje število vozlišč barve c na poti od
pair<int, int> ObdelajSkrcenoDrevo(StVozlisca u)                    // u navzdol⟩.
{
  auto &U = vozl[u];
  // Za vsakega u-jevega otroka „v“ poiščimo največje število vozlišč barve c na poti od „v“
  // navzdol; največji dve od teh števil si zapomnimo v „maxNaVeji1“ in „maxNaVeji2“.
  // V „maxLepa“ hranimo največjo lepo podmnožico po poddrevesih u-jevih otrok.
  int maxNaVeji1 = 0, maxNaVeji2 = 0, maxLepa = 0;
  for (StVozlisca v = U.otrokS; v >= 0; )
  {
    auto &V = vozl[v];
    auto [vMaxLepa, vMaxNaVeji] = ObdelajSkrcenoDrevo(v);
    maxLepa = max(maxLepa, vMaxLepa);
    if (vMaxNaVeji > maxNaVeji1) maxNaVeji2 = maxNaVeji1, maxNaVeji1 = vMaxNaVeji;
    else if (vMaxNaVeji > maxNaVeji2) maxNaVeji2 = vMaxNaVeji;
    v = V.sorS;
  }
  // Znotraj u-jevega poddrevesa je možna tudi taka lepa podmnožica, ki
  // jo pokriva pot z vrhom pri u, ki se pred in po tem spušča v dva različna u-jeva
  // otroka (v enem pokrije maxNaVeji1 vozlišč barve c, v drugem pa maxNaVeji2).
  maxLepa = max(maxLepa, maxNaVeji1 + maxNaVeji2 + (U.barva == c ? 1 : 0));
  return {maxLepa, maxNaVeji1 + (U.barva == c ? 1: 0)};
}

int main()
{
  // Preberimo vhodne podatke.
  ios_base::sync_with_stdio(false);
  cin >> n >> b; vozl.resize(n);
  vector<StVozlisca> prviPoBarvi(b, −1); // prviPoBarvi[b] = prvo vozlišče barve b

  const bool velikoDrevo = (n > 200'000);
  if (! velikoDrevo) for (int u = 0; u < n; ++u) {
    auto &U = vozl[u]; cin >> U.stars >> U.barva; −−U.stars; −−U.barva; }

  else { // Velika drevesa so predstavljena parametrično.
    long long A, B, M, K, AA, BB, MM, r = 0, rr = 0;
    cin >> A >> B >> M >> K >> AA >> BB >> MM;
    for (int u = 0; u < n; ++u) {
      r = (A * r + B) % M; rr = (AA * rr + BB) % MM;
      auto &U = vozl[u]; U.barva = rr % b;
      U.stars = (u == 0) ? −1 : (u − 1 − r % min((long long) u, K)); } }

  // Pripravimo sezname vozlišč posamezne barve.
  for (int u = 0; u < n; ++u) { auto &U = vozl[u];
    if (U.stars < 0) koren = u;
    else { auto &P = vozl[U.stars]; U.sor = P.otrok; P.otrok = u; }
    U.naslIsteBarve = prviPoBarvi[U.barva]; prviPoBarvi[U.barva] = u; }

  // Za vsako vozlišče u določimo njegovega starša v skrčenem drevesu za u-jevo barvo.
  zadnjiPoBarvi.resize(b, −1); DolociStarseB(koren);

  // Obdelajmo vse barve.
  long long vsota = 0;
  for (c = 0; c < b; ++c)
  {
    vector<StVozlisca> vozlS; // vozlišča v skrčenem drevesu

    // Pripravimo skrčeno drevo za barvo c.
    for (StVozlisca u = prviPoBarvi[c]; u >= 0; u = vozl[u].naslIsteBarve)
    {
      // Vozlišče u je barve c, zato ga je treba dodati v skrčeno drevo,
      // kjer bo njegov starš vozlišče „U.starsB“.
      auto &U = vozl[u];
```

```
        U.dodan = c; vozlS.emplace_back(u);
        if (U.starsB < 0) continue;
        auto &P = vozl[U.starsB]; U.sorS = P.otrokS; P.otrokS = u;
        // Če u-jev starš v skrčenem drevesu ni barve c, ga bo treba šele
        // dodati v drevo (razen če ga nismo že dodali pri kakšnem prejšnjem u).
        if (P.barva != c && P.dodan < c) {
          P.dodan = c; vozlS.emplace_back(U.starsB);
          // P-jev starš v skrčenem drevesu bo isto vozlišče, ki je njegov
          // starš tudi v prvotnem drevesu (in ki je gotovo barve c).
          if (P.stars >= 0) { auto &G = vozl[P.stars];
            P.sorS = G.otrokS; G.otrokS = U.starsB; } }
      }
      // Rešimo nalogo v skrčenem drevesu.
      int maxLepa = ObdelajSkrcenoDrevo(koren).first;
      if (velikoDrevo) vsota += maxLepa; else cout << maxLepa << '\n';
      // Pospravimo skrčeno drevo.
      for (StVozlisca u : vozlS) { auto &U = vozl[u]; U.otrokS = -1; U.sorS = -1; }
    }
    if (velikoDrevo) cout << vsota << '\n';
    return 0;
}
```

5. Otoki

Karirasto mrežo bomo predstavili z grafom, vendar tako, da bodo točke grafa predstavljale celice mreže — točka (x, y) našega grafa naj predstavlja celico, ki pokriva kvadratno območje $[x, x+1] \times [y, y+1]$. Povezave med dvema točkama v grafu pa naj bodo prisotne tam, kjer tisti dve celici mreže mejita druga na drugo s stranico, vendar na tej stranici ni več povezave (v mreži), ker je bila že pobrisana. Na začetku, ko še ni bila pobrisana nobena povezava mreže, ni v našem grafu nobene povezave; vsaka točka grafa je ločena od ostalih. Ko pa povezave v mreži brišemo, se ustrezne povezave dodajajo v graf. Če na primer pobrišemo v mreži navpično povezavo med (x, y) in $(x, y+1)$, moramo v graf dodati povezavo med celicama $(x-1, y)$ in (x, y); podobno pa, če pobrišemo v mreži vodoravno povezavo med (x, y) in $(x+1, y)$, moramo v graf dodati povezavo med celicama $(x, y-1)$ in (x, y).

V našem grafu sta dve celici dosegljivi ena iz druge (v enem ali več korakih po povezavah našega grafa) natanko tedaj, ko je mogoče v mreži priti od ene celice do druge, ne da bi prečkali kakšno od še nepobrisanih povezav. Recimo zdaj, da v naslednjem koraku pobrišemo v mreži povezavo na meji med celicama u in v, ki pa sta bili v grafu dosegljivi ena iz druge že pred tem brisanjem. Ta pot od u do v pa bo skupaj z novo povezavo med u in v, ki jo bomo zdaj dodali v graf (ker bomo v mreži pobrisali povezavo na meji med u in v), tvorila cikel. Če se v mislih sprehodimo po celicah, ki tvorijo ta cikel, bomo lahko prišli naokrog po celem ciklu, ne da bi kdaj prečkali kakšno povezavo mreže — vse so bile že pobrisane. Tisti del mreže torej, ki leži znotraj cikla, je zdaj odrezan od preostanka mreže (ki leži zunaj cikla). Pred zadnjim brisanjem pa še ni bil povsem odrezan; notranjost cikla je bila sicer morda sestavljena iz več ločenih delov, eden od teh delov pa je bil še povezan z zunanjostjo, kajti eno od krajišč povezave, ki smo jo v mreži pravkar pobrisali, leži v notranjosti cikla, drugo pa v zunanjosti in pred tem zadnjim brisanjem sta bili tidve krajišči seveda povezani. Po brisanju pa nista več, torej se število delov, na katere je mreža razdeljena, poveča za 1.

Povzemimo dosedanji razmislek s psevdokodo:

```
1  štDelov := 1; (* Na začetku je celotna mreža povezana. *)
2  G := graf s točkami (x, y) za vse x, y ∈ ℤ, vendar brez povezav;
3  za vsako pobrisano povezavo mreže:
4    naj bo sta u in v celici, na meji med katerima leži ta povezava;
5    če sta u in v v grafu G dosegljivi ena iz druge:
6      štDelov := štDelov + 1;
7    dodaj v G povezavo med u in v;
```

Na koncu nam spremenljivka *štDelov* pove, na koliko delov je razpadla naša mreža zaradi brisanja povezav.

Razmisliti moramo le še o tem, kako ta postopek učinkovito implementirati. Graf G je sicer neskončen, vendar večina točk (celic) v njem nima nobene povezave, zato iz njih tudi ni dosegljiva nobena druga točka (celica); takih točk lahko tudi nimamo v grafu, pa se ne bo to nič poznalo, ko bomo v vrstici 5 preverjali, ali sta u in v dosegljivi ena iz druge. V grafu bomo zato v resnici imeli le tiste točke, ki imajo v grafu vsaj eno povezavo — ali, z drugimi besedami: tiste celice, za katere smo v mreži že pobrisali vsaj eno od povezav na njihovih štirih stranicah. Ko torej v vrstici 4 ugotovimo, s katerima celicama u in v imamo trenutno opravka, moramo najprej pogledati, če ju sploh že imamo v grafu, in če ju še nimamo, ju moramo zdaj dodati.

Ko nas potem v vrstici 5 zanima, ali sta u in v dosegljivi ena iz druge, je to enako vprašanju, ali pripadata isti povezani komponenti grafa. Če sta pripadali različnima komponentama, se bosta tidve komponenti nato, ko bomo v vrstici 7 dodali v graf povezavo med u in v, združili v eno. Naš graf je torej koristno predstaviti z dobro znano podatkovno strukturo — gozdom disjunktnih množic (*disjoint-set forest*). Ta podpira prav te operacije, ki nas zanimajo: za vsako celico lahko ugotovimo, kateri komponenti pripada, in dve komponenti lahko združimo v eno. Za zaporedje $O(n)$ takšnih operacij porabimo pri tej podatkovni strukturi $O(n\,\alpha(n))$ časa, pri čemer je α inverzna Ackermannova funkcija, ki narašča tako počasi, da jo lahko v praksi obravnavamo bolj ali manj kot konstanto.

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

typedef pair<long long, long long> Par;

// S pomočjo razpršene tabele „slovarCelic“ pripišemo vsaki celici (x, y)
// enolično zaporedno številko, ki potem predstavlja to celico v
// podatkovni strukturi za disjunktne množice.
template<> struct std::hash<Par> {
  size_t operator () (Par P) const {
    constexpr size_t M = 1'000'000'007;
    return size_t(P.first % M) * M + size_t(P.second % M); } };

unordered_map<Par, int> slovarCelic;

// „celice“ je naš gozd disjunktnih množic; za vsako celico vzdržujemo
// podatek o njenem staršu in velikost njenega poddrevesa; slednja je sicer
// veljavna le, če je trenutna celica koren svojega drevesa v gozdu (pri
// taki kaže „stars“ nazaj na to isto celico).
struct Celica { int stars, velikost = 0; };
vector<Celica> celice;

// Naslednja funkcija vrne koren drevesa, ki mu celica „u“ pripada v
// gozdu disjunktnih množic; poleg tega tudi popravi podatke o starših na
// poti od „u“ do korena, da vse te celice kažejo naravnost na koren.
int Komponenta(int u)
{
  int v = u; while (celice[v].stars != v) v = celice[v].stars;
  while (u != v) { int p = celice[u].stars; celice[u].stars = v; u = p; }
  return v;
}

int main()
{
  ios_base::sync_with_stdio(false);
  int n; cin >> n; int stDelov = 1; // Sprva je cela mreža en povezan del.
  while (n-- > 0)
  {
    // Preberimo naslednjo povezavo, ki jo bomo pobrisali.
```

```
    long long x1, y1, x2, y2; cin >> x1 >> y1 >> x2 >> y2;
    // Med katerima dvema celicama leži ta povezava?
    // Celica (x, y) predstavlja kvadrat [x, x + 1] × [y, y + 1].
    Par U, V; int u, v;
    if (x1 == x2) U = {x1 − 1, min(y1, y2)}, V = {x1, min(y1, y2)};
    else U = {min(x1, x2), y1 − 1}, V = {min(x1, x2), y1};
    // Briše se meja med celicama U in V; poskrbimo, da bosta tidve celici prisotni
    // v naših podatkovnih strukturah. S pomočjo slovarja poiščimo njuni zaporedni
    // številki, če pa ju še ni tam, dodajmo celici v slovar in v gozd (ko celico prvič
    // dodamo v gozd, nima še nobenih povezav, zato tvori majhno komponento sama zase).
    { auto [it, jeNova] = slovarCelic.emplace(U, (int) celice.size());
    u = it->second; if (jeNova) celice.push_back({u, 1}); }
    { auto [it, jeNova] = slovarCelic.emplace(V, (int) celice.size());
    v = it->second; if (jeNova) celice.push_back({v, 1}); }
    // Poglejmo, katerima komponentama pripadata u in v.
    u = Komponenta(u); v = Komponenta(v);
    // Če sta že v isti komponenti, nastane cikel, ki odreže neki del mreže
    // (v notranjosti cikla) od preostanka (zunanjosti cikla).
    if (u == v) ++stDelov;
    else {
      // Sicer se komponenti, ki jima pripadata naši celici, združita.
      // Manjšo od njiju bomo pridružili k večji (kot poddrevo korena te večje).
      if (celice[u].velikost < celice[v].velikost) swap(u, v);
      celice[v].stars = u; celice[u].velikost += celice[v].velikost; }
  }
  cout << stDelov << endl; return 0; // Izpišimo rezultat.
}
```

Naloge so sestavili: WC-kabine — Matija Grabnar; diagram, multiprocesiranje — Tomaž Hočevar; merilec elektrike — Vid Kocijan in Janez Brank; pangramski stavki — Polona Novak; Giganti — Jakob Omahen; profil pokrajine, ChordPro v2, parica — Jakob Schrader; palačinke, alkoholni test — Jošt Smrtnik; zaporedje števk, otoki — Luka Urbanc; tabela ničel, lepe podmnožice — Jakob Žorž.

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