11. tekmovanje ACM v znanju računalništva za srednješolce

19. marca 2016

NASVETI ZA 1. IN 2. SKUPINO

Nekatere naloge so tipa **napiši program** (ali **napiši podprogram**), nekatere pa tipa **opiši postopek**. Pri slednjih ti ni treba pisati programa ali podprograma v kakšnem konkretnem programskem jeziku, ampak lahko postopek opišeš tudi kako drugače: z besedami (v naravnem jeziku), psevdokodo (glej spodaj), diagramom poteka itd. Glavno je, da je tvoj opis dovolj natančen, jasen in razumljiv, tako da je iz njega razvidno, da si dejansko našel in razumel pot do rešitve naloge.

Psevdokodi pravijo včasih tudi strukturirani naravni jezik. Postopek opišemo v naravnem jeziku, vendar opis strukturiramo na podoben način kot pri programskih jezikih, tako da se jasno vidi strukturo vejitev, zank in drugih programskih elementov.

Primer opisa postopka v psevdokodi: recimo, da imamo zaporedje besed in bi ga radi razbili na več vrstic tako, da ne bo nobena vrstica preširoka.

```
naj bo trenutna vrstica prazen niz;
pregleduj besede po vrsti od prve do zadnje:
    če bi trenutna vrstica z dodano trenutno besedo (in presledkom
    pred njo) postala predolga,
        izpiši trenutno vrstico in jo potem postavi na prazen niz;
    dodaj trenutno besedo na konec trenutne vrstice;
če trenutna vrstica ni prazen niz, jo izpiši;
```

(Opomba: samo zato, ker je tu primer psevdokode, to še ne pomeni, da moraš tudi ti pisati svoje odgovore v psevdokodi.)

Če pa v okviru neke rešitve pišeš izvorno kodo programa ali podprograma, obvezno poleg te izvorne kode v nekaj stavkih opiši, kako deluje (oz. naj bi delovala) tvoja rešitev in na kakšni ideji temelji.

Pri ocenjevanju so vse naloge vredne enako število točk. Svoje odgovore dobro utemelji. Prizadevaj si predvsem, da bi bile tvoje rešitve pravilne, ob tem pa je zaželeno, da so tudi čim bolj učinkovite; take dobijo več točk kot manj učinkovite (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). Za manjše sintaktične napake se ne odbije veliko točk. Priporočljivo in zaželeno je, da so tvoje rešitve napisane pregledno in čitljivo. Če je na listih, ki jih oddajaš, več različic rešitve za kakšno nalogo, jasno označi, katera je tista, ki naj jo ocenjevalci upoštevajo.

Če naloga zahteva branje ali obdelavo vhodnih podatkov, lahko tvoja rešitev (če v nalogi ni drugače napisano) predpostavi, da v vhodnih podatkih ni napak (torej da je njihova vsebina in oblika skladna s tem, kar piše v nalogi).

Nekatere naloge zahtevajo branje podatkov s standardnega vhoda in pisanje na standardni izhod. Za pomoč je tu nekaj primerov programov, ki delajo s standardnim vhodom in izhodom:

- Program, ki prebere s standardnega vhoda dve števili in izpiše na standardni izhod njuno vsoto:

```
program BranjeStevil;
var i, j: integer;
begin
  ReadLn(i, j);
  WriteLn(i, ' + ', j, ' = ', i + j);
end. {BranjeStevil}
```

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

- Program, ki bere s standardnega vhoda po vrsticah, jih šteje in prepisuje na standardni izhod, na koncu pa izpiše še skupno dolžino:

```
program BranjeVrstic;
var s: string; i, d: integer;
begin
  i := 0; d := 0;
  while not Eof do begin
    ReadLn(s);
    i := i + 1; d := d + Length(s);
    WriteLn(i, '. vrstica: "', s, '"');
  end; {while}
  WriteLn(i, ' vrstic, ', d, ' znakov.');
end. {BranjeVrstic}
```

```
#include <stdio.h>
#include <string.h>
int main() {
  char s[201]; int i = 0, d = 0;
  while (gets(s)) {
    i++; d += strlen(s);
    printf("%d. vrstica: \"%s\"\n", i, s);
  }
  printf("%d vrstic, %d znakov.\n", i, d);
  return 0;
}
```

Opomba: C-jevska različica gornjega programa predpostavlja, da ni nobena vrstica vhodnega besedila daljša od dvesto znakov. Funkciji gets se je v praksi bolje izogibati, ker pri njej nimamo zaščite pred primeri, ko je vrstica daljša od naše tabele s. Namesto gets bi bilo bolje uporabiti fgets; vendar pa za rešitev naših tekmovalnih nalog v prvi in drugi skupini zadošča tudi gets.

- Program, ki bere s standardnega vhoda po znakih, jih prepisuje na standardni izhod, na koncu pa izpiše še število prebranih znakov (ne vštevši znakov za konec vrstice):

```
program BranjeZnakov;
var i: integer; c: char;
begin
  i := 0;
  while not Eof do begin
    while not Eoln do
      begin Read(c); Write(c); i := i + 1 end;
    if not Eof then begin ReadLn; WriteLn end;
  end; {while}
  WriteLn('Skupaj ', i, ' znakov.');
end. {BranjeZnakov}
```

```
#include <stdio.h>

int main() {
  int i = 0, c;
  while ((c = getchar()) != EOF) {
    putchar(c); if (i != '\n') i++;
  }
  printf("Skupaj %d znakov.\n", i);
  return 0;
}
```

Še isti trije primeri v pythonu:

```
# Branje dveh števil in izpis vsote:
import sys

a, b = sys.stdin.readline().split()
a = int(a); b = int(b)
print "%d + %d = %d" % (a, b, a + b)
```

```
# Branje standardnega vhoda po vrsticah:
import sys

i = d = 0
for s in sys.stdin:
  s = s.rstrip('\n') # odrežemo znak za konec vrstice
  i += 1; d += len(s)
  print "%d. vrstica: \"%s\"" % (i, s)
print "%d vrstic, %d znakov." % (i, d)
```

```
# Branje standardnega vhoda znak po znak:
import sys

i = 0
while True:
  c = sys.stdin.read(1)
  if c == "": break # EOF
  sys.stdout.write(c)
  if c != '\n': i += 1
print "Skupaj %d znakov." % i
```

Še isti trije primeri v javi:

```
// Branje dveh števil in izpis vsote:
import java.io.*;
import java.util.Scanner;

public class Primer1
{
   public static void main(String[] args) throws IOException
   {
      Scanner fi = new Scanner(System.in);
      int i = fi.nextInt(); int j = fi.nextInt();
      System.out.println(i + " + " + j + " = " + (i + j));
   }
}
```

```
// Branje standardnega vhoda po vrsticah:
import java.io.*;

public class Primer2
{
   public static void main(String[] args) throws IOException
   {
      BufferedReader fi = new BufferedReader(new InputStreamReader(System.in));
      int i = 0, d = 0; String s;
      while ((s = fi.readLine()) != null) {
         i++; d += s.length();
         System.out.println(i + ". vrstica: \"" + s + "\""); }
      System.out.println(i + " vrstic, " + d + " znakov.");
   }
}
```

```
// Branje standardnega vhoda znak po znak:
import java.io.*;

public class Primer3
{
   public static void main(String[] args) throws IOException
   {
      InputStreamReader fi = new InputStreamReader(System.in);
      int i = 0, c;
      while ((c = fi.read()) >= 0) {
         System.out.print((char) c); if (c != '\n' && c != '\r') i++; }
      System.out.println("Skupaj " + i + " znakov.");
   }
}
```

11. tekmovanje ACM v znanju računalništva za srednješolce

19. marca 2016

NALOGE ZA PRVO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del prek računalnika. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku (npr. kopiraj in prilepi v Notepad in shrani v datoteko). **Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave in bi rad, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici.**

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

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

1. Tipkanje

Znašel si se v vlogi zapisnikarja, ki mora na računalniku vnesti seznam n besed. To počneš tako, da s pritiski na tipkovnico vnašaš črko po črko v vnosno polje, ki je na začetku prazno. Ko je v polju izpisana zahtevana beseda, zaključiš vnos s pritiskom na tipko Enter. Beseda pri tem ostane v vnosnem polju. Nato s pritiski na tipko Backspace pobrišeš nekaj (morda vse, ali pa nobene) zadnjih črk in nadaljuješ z vnosom naslednje besede. **Napiši program**, ki bo izpisal, najmanj koliko pritiskov tipk boš potreboval, da vneseš podane besede. Tvoj program lahko vhodne podatke bere s standardnega vhoda ali pa iz datoteke `besede.txt` (kar ti je lažje); v prvi vrstici je število besed, nato pa sledi ustrezno število vrstic, v vsaki od njih je po ena beseda. Besede vsebujejo le male črke angleške abecede. Posamezna beseda je dolga največ 100 znakov.

Primer vhoda:

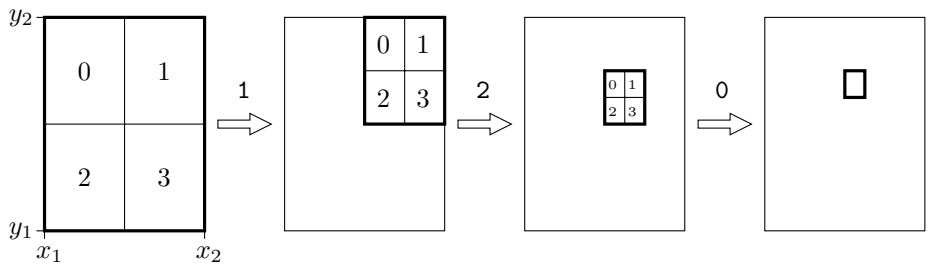
```
3
abc
aaaa
ba
```

Pripadajoči izhod:

```
17
```

2. Zoom

Na računalniku gledamo zemljevid, ki pokriva v koordinatnem sistemu območje $[x_1, x_2] \times [y_1, y_2]$, kot kaže spodnja slika. Razdelimo ga na štiri četrtine in jih označimo: 0 je zgornja leva, 1 je zgornja desna, 2 je spodnja leva in 3 je spodnja desna. Izberemo si eno od četrtin in pozumiramo prikaz zemljevida tako, da vidimo le še to četrtino. Tudi njo razdelimo na četrtine in pozumiramo v eno od njih. Ta korak še nekajkrat ponovimo. Z nizom, ki ga sestavljajo znaki `0`, `1`, `2` in `3`, lahko opišemo, v katero četrtino smo se premaknili na vsakem koraku. Naslednja slika kaže primer za niz `"120"`:



Napiši podprogram Zoom(s, x1, y1, x2, y2), ki za dani niz s (zaporedje znakov `0`, `1`, `2`, `3`, ki opisujejo potek zumiranja) in koordinate x_1, x_2, y_1, y_2 celotnega zemljevida (pri tem zagotovo velja $x_1 < x_2$ in $y_1 < y_2$) izpiše koordinate tistega območja, ki ga gledamo po zadnjem koraku zumiranja.

Tvoj podprogram naj bo takšne oblike:

```
procedure Zoom(s: string; x1, y1, x2, y2: double);                    { v pascalu }
void Zoom(char *s, double x1, double y1, double x2, double y2);       /* v C/C++ */
void Zoom(string s, double x1, double y1, double x2, double y2);      // v C++
public static void Zoom(String s, double x1, double y1, double x2, double y2);   // v javi
public static void Zoom(string s, double x1, double y1, double x2, double y2);   // v C#
def Zoom(s, x1, y1, x2, y2): ...      # v pythonu; s je tipa str, ostali pa tipa float
```

3. Zaklepajski izrazi

Oklepajski izrazi so nizi, ki jih sestavljajo sami oklepaji in zaklepaji, morajo pa biti pravilno gnezdeni. Oklepaji in zaklepaji so lahko različnih oblik: okrogli `( )`, oglati `[ ]`, zaviti `{ }` in kotni `< >`. „Pravilno gnezdeni“ pomeni, da mora imeti vsak oklepaj tudi pripadajoč zaklepaj enake oblike (in obratno), podniz med njima pa mora biti tudi sam zase oklepajski izraz.

Nekaj primerov oklepajskih izrazov: `<<>>`, `[()(<>)]()`, `{[{}()]<>}`.

Nekaj primerov nizov, ki *niso* oklepajski izrazi: `(()(`, `([)]`, `<>><<>`.

Dan je nek niz `s`, v katerem se pojavljajo le zaklepaji različnih oblik — `) ] } >` — in zvezdice `*`. **Napiši podprogram** Dopolni(s), ki vsako zvezdico v nizu `s` spremeni v enega od oklepajev — torej znakov `( [ { <` — tako, da bo iz tega na koncu nastal pravilno gnezden oklepajski izraz. Podprogram naj tako popravljeni niz izpiše, če pa se izkaže, da ga ni mogoče ustrezno popraviti (da bi nastal oklepajski izraz), naj izpiše, da je problem nerešljiv.

Primer: iz `"**]*))"` lahko naredimo `"([]())"`. Iz `"***>*)"` pa ne moremo narediti veljavnega oklepajskega izraza ne glede na to, kako spreminjamo zvezdice v oklepaje.

Tvoj podprogram naj bo takšne oblike:

```
procedure Dopolni(s: string);          { v pascalu }
void Dopolni(char *s);                 /* v C/C++ */
void Dopolni(string s);                // v C++
public static void Dopolni(String s);  // v javi
public static void Dopolni(string s);  // v C#
def Dopolni(s): ...                    # v pythonu; s je tipa str
```

4. Izštevanka

Otroci so se že malo naveličali vsakokrat uporabljati preprosto izštevanko „An ban pet podgan“ za določitev tistega, ki lovi, zato so se odločili za manjšo spremembo: vsak naj ima dve življenji, kdor ostane brez, je „izbrani“.

V krog se postavi n otrok, vsak stoji z obema nogama na tleh (t.j. ima dve življenji). Naj bodo oštevilčeni z zaporednimi številkami od 1 do n. Začnemo pri otroku številka 1 in od njega naredimo k korakov po krogu (pri tem torej njega ne štejemo, ampak štejemo šele otroke od 2 naprej) — k je pozitivno celo število, lahko je tudi večje od n. Otrok na tako določenem mestu mora dvigniti nogo (izgubi eno življenje). Če mora dvigniti še drugo in zato pasti, je izštevanke konec in ta otrok postane „izbrani“. Če pa še vedno stoji na eni nogi (je pravkar izgubil šele prvo življenje), se izštevanka nadaljuje (spet začne šteti) pri otroku neposredno za njim, hkrati pa izštevanko podaljšamo za eno besedo, torej povečamo k za 1.

Napiši program, ki bo prebral dve pozitivni celi števili — n in začetno vrednost k, opravil korake izštevanke in izpisal številko „izbranega“ otroka, to je tistega, ki je padel po tleh, ker je izgubil obe življenji. Število otrok n je vsaj 1 in kvečjemu 100.

Primer: če imamo $n = 5$ in $k = 3$, morajo po vrsti dvigovati noge otroci 4, 3, 3 in igre je konec — izbran je otrok številka 3.

5. H-indeks

Hirschev indeks (krajše tudi h-indeks) je ena izmed številnih ocen, ki poskušajo meriti uspešnost raziskovalcev. Cilj ocene h-indeks je uravnotežiti produktivnost (število objavljenih člankov) in vplivnost (citiranost njegovih člankov) posameznega raziskovalca. H-indeks je definiran kot največje celo število h, za katerega velja, da je raziskovalec objavil h člankov, kjer je bil vsak izmed teh člankov citiran vsaj h-krat. **Opiši postopek** (ali napiši program ali podprogram, kar ti je lažje), ki bo iz podanega seznama citiranosti člankov izračunal h-indeks. Posamezni elementi tega seznama so števila, ki za posamezne članke povedo, kolikokrat so bili citirani.

Primer: če imamo seznam $[6, 5, 3, 2, 5, 10, 5, 7]$, je njegov h-indeks enak 5, kajti v seznamu obstaja vsaj pet elementov, večjih ali enakih 5, ne obstaja pa v njem vsaj šest elementov, večjih ali enakih 6.

11. tekmovanje ACM v znanju računalništva za srednješolce

19. marca 2016

NALOGE ZA DRUGO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del prek računalnika. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku (npr. kopiraj in prilepi v Notepad in shrani v datoteko). **Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave in bi rad, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici.**

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

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

1. Sorodstvo

Navdušencu za rodoslovje se je sesula baza podatkov o njegovih prednikih in sorodstvenih povezavah med njimi. Vse, kar je uspel rešiti, so letnice rojstev in smrti posameznih ljudi. Ima torej seznam parov celih števil (r_i, s_i) za $i = 1, \ldots, n$, ki povedo, da se je oseba i rodila v letu r_i in umrla v letu s_i. Vrstni red ljudi v seznamu je lahko poljubno premešan, torej ni nujno, da so npr. urejeni po letu rojstva ali kaj podobnega.

Predpostavimo, da je razlika v starosti staršev in otrok vsaj d let (d je neko celo število, večje od 0). Zato bomo rekli, da je oseba j morebitni otrok osebe i natanko tedaj, ko velja $r_i + d \leq r_j \leq s_i$.

Opiši postopek, ki za dani d in podatke o letih rojstev in smrti $(r_1, s_1), \ldots, (r_n, s_n)$ sestavi najdaljše zaporedje oseb, v katerem velja, da je vsaka naslednja oseba v zaporedju morebitni otrok prejšnje osebe v zaporedju.

Tvoja rešitev naj se ne opira na kakšne realistične predpostavke o tem, kako dolgo ljudje živijo in koliko otrok imajo, saj imajo nekateri uporabniki v svojem rodbinskem drevesu tudi vesoljce, sumerske polbogove in podobna dolgoživa bitja.

2. Suhi dnevi

Avtomatska vremenska postaja meri količino padavin. Večkrat na dan se tako odčita, koliko padavin je padlo od prejšnjega odčitka, ta podatek se vsakokrat zapiše v datoteko kot dve pozitivni celi števili: prvo število v vrstici je datum (kako je to število sestavljeno, ni določeno, zagotovljeno je le, da ima isti dan vedno enako številko, različno od drugih dni); drugo število v vrstici je količina novozapadlih padavin od prejšnjega odčitka. Zagotovimo lahko, da je v vsakem dnevu vsaj en odčitek in da je zadnji odčitek vsakega dneva opolnoči (torej ni neizmerjenih ostankov, ki bi se prenašali v naslednji dan). Podatki se zapisujejo kronološko, torej niso časovno pomešani med seboj.

Tako se je nabralo za točno eno leto podatkov. **Napiši program**, ki bo prebral te podatke in izpisal, koliko je bilo suhih dni — to so dnevi, v katerih ni zapadlo nič padavin. Poleg tega naj izpiše tudi, koliko je bilo največje število zaporednih suhih dni v tem letu. Tvoj program naj bere iz datoteke `meritve.txt` ali pa iz standardnega vhoda, kar ti je lažje; meritve naj bere vse do konca podatkov (EOF).

Primer podatkov:

```
20160101 0
20160101 0
20160101 12
20160101 30
02012016 10
02012016 0
02012016 120
12345 0
12345 0
12345 0
20160104 0
20160105 0
20160105 23
...
20161231 0
```

3. Virus

V računalniškem podjetju so naredili 1000 zgoščenk z nekim programom, namenjenim za prodajo. Žal pa je na eno izmed zgoščenk zašel tudi virus, ki ga želimo odkriti in tisto zgoščenko uničiti. Okuženo zgoščenko bi radi našli čim prej tako, da zgoščenke testiramo na enem ali več računalnikih. Na enem računalniku lahko poženemo poljubno mnogo zgoščenk. Če je na nekem računalniku med zagnanimi zgoščenkami bila tudi taka z virusom, se bo računalnik do naslednjega dne sesul. Takrat (torej naslednji dan) ga lahko ponovno usposobimo in ga uporabimo za nadaljnje poskuse.

Čas za zagon zgoščenk je zanemarljivo majhen (lahko ga odmisliš). Virus iz okuženega računalnika se ne bo razširil na druge zgoščenke.

Opiši postopek, kako na računalnikih poganjati zgoščenke, da ugotovimo, katera zgoščenka je okužena z virusom. Pravzaprav opiši dva postopka (vsak je vreden polovico točk pri tej nalogi) z naslednjimi omejitvami:

(a) Za eksperimentiranje imamo na voljo le en računalnik, zgoščenko pa bi radi našli v minimalnem številu dni.

(b) Zgoščenko moramo najti v enem dnevu, za eksperimentiranje pa želimo uporabiti čim manj računalnikov (po možnosti precej manj kot 1000).

4. Analiza enot

Pogosto se zgodi, da dijak pri pouku fizike na tablo napiše napačno formulo, npr. za hitrost: $v = s \cdot t$. Nato profesor prav tako pogosto vzklikne: „Pa, saj to se že od daleč vidi, da je narobe. Na levi strani so enote metri na sekundo, na desni pa imaš metre krat sekunde, seveda je narobe, saj se enote ne ujemajo!“ **Napiši program** ali podprogram (kar ti je lažje), ki prebere fizikalno formulo in preveri, ali se enote na obeh straneh enačaja ujemajo.

Formula je dana kot enakost dveh ulomkov, na primer

```
a b c d / x y z = h / i j k
```

kar ustreza formuli $\frac{abcd}{xyz} = \frac{h}{ijk}$. Pri tem male črke angleške abecede (od `a` do `z`) predstavljajo fizikalne količine. Imenovalci ulomkov niso nujno prisotni, če pa so, je na vsaki strani znaka `/` gotovo vsaj ena količina.

Poleg tega imaš na začetku podane tudi enote za vsako fizikalno količino, ki v formuli nastopa. Enote so podane v obliki ulomka, za katerega veljajo enaka pravila kot za ulomke v formuli. Na primer:

```
a : m / s s
```

Vse fizikalne količine bodo označene z malimi črkami angleške abecede, prav tako pa tudi njihove enote. Enote in količine imajo lahko enake črke. Tvoj program lahko bere podatke s standardnega vhoda ali pa iz datoteke `enote.txt` (kar ti je lažje). V prvi vrstici je število količin n, ki bodo uporabljene v enačbi. V naslednjih n vrsticah sledijo izražave teh količin v osnovnih enotah. Nato sledi enačba v obliki, kot je opisana zgoraj. Posamezni znaki v formulah in opisih količin so ločeni s po enim presledkom. Tvoj program naj izpiše samo „`Formula je pravilna.`“ ali „`Formula ni pravilna.`“, odvisno od tega, ali se enote ujemajo ali ne.

Primer vhoda:

```
3
h : m
g : m / s s
v : m / s
h = v v / g
```

Pripadajoči izhod:

```
Formula je pravilna.
```

Še en primer:

```
4
f : g m / s s
m : g
t : s
s : m
f / m = s / t
```

Pripadajoči izhod:

```
Formula ni pravilna.
```

5. Za žužke gre

Mirko je navdušen žužkofil in podpornik pravic otrok. Doma ima terarij z n žužki, oštevilčenimi s števili od 1 do n, toda ne pozna njihovega spola. Kljub temu pa trdi, da so gotovo vsi žužki heteroseksualni. To želi dokazati tako, da en mesec strmi v terarij in si beleži interakcije med žužki.

Natančno **opiši postopek**, ki sprejme seznam interakcij med žužki in pove, ali obstaja taka razporeditev spolov, da so bile vse interakcije heteroseksualne (torej med žužkoma različnih spolov). Tvoj postopek kot vhodne podatke dobi število žužkov n, število interakcij med njimi m in seznam teh m interakcij — vsaka interakcija je opisana z dvema številoma s_i in t_i, ki povesta, da sta v i-ti interakciji sodelovala žužka s številkama s_i in t_i.

V posamezni interakciji vedno sodelujeta natanko dva žužka in noben žužek ni nikoli v interakciji sam s sabo. Žužki so lahko samo dveh spolov in spola ne spreminjajo med mesecem opazovanja.

11. tekmovanje ACM v znanju računalništva za srednješolce

19. marca 2016

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*.`in` in izpisujejo svoje rezultate v *imenaloge*.`out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil mapo s svojim uporabniškim imenom ter navodili, ki jih pravkar prebiraš. Ko boš sedel pred računalnik, boš dobil nadaljnja navodila za prijavo v sistem.

Na vsakem računalniku imaš na voljo disk `D:\`, v katerem lahko kreiraš svoje datoteke in imenike. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali VB.NET, mi pa jih bomo preverili s 64-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz OpenJDK 8 in s prevajalnikom Mono 4.2 za C# in VB.NET. Za delo lahko uporabiš Lazarus (IDE za pascal), `gcc`/`g++` (GNU C/C++ — command line compiler), `javac` (za javo 1.8), Visual Studio, Eclipse in druga orodja.

Na spletni strani `http://rtk2016.fri1.uni-lj.si/` boš dobil nekaj testnih primerov.

Prek iste strani lahko oddaš tudi rešitve svojih nalog, tako da tja povlečeš datoteko z izvorno kodo svojega programa. Ime datoteke naj bo takšne oblike:

imenaloge.`pas`
imenaloge.`c`
imenaloge.`cpp`
ImeNaloge.`java`
ImeNaloge.`cs`
ImeNaloge.`vb`

Datoteka z izvorno kodo, ki jo oddajaš, ne sme biti daljša od 30 KB.

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih (praviloma desetih). Za vsak testni primer se bo izpisalo, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund ali pa porabil več kot 200 MB pomnilnika, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku, razen s predpisano vhodno in izhodno datoteko. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku), prenosnih računalnikov, prenosnih telefonov itd.

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih; pri prvi in tretji nalogi je po deset testnih primerov in pri vsakem od njih dobi program 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk; pri drugi, četrti in peti nalogi je po 20 testnih primerov in pri vsakem od njih dobi program 5 točk, če je izpisal pravilen odgovor, sicer pa 0 točk.

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

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

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

Poskusna naloga (ne šteje k tekmovanju) (`poskus.in`, `poskus.out`)

Napiši program, ki iz vhodne datoteke prebere dve celi števili (obe sta v prvi vrstici, ločeni z enim presledkom) in izpiše desetkratnik njune vsote v izhodno datoteko.

Primer vhodne datoteke:

```
123 456
```

Ustrezna izhodna datoteka:

```
5790
```

Primeri rešitev (dobiš jih tudi kot datoteke na `http://rtk2016.fri1.uni-lj.si/`):

- V pascalu:

```
program PoskusnaNaloga;
var T: text; i, j: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i, j); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * (i + j)); Close(T);
end. {PoskusnaNaloga}
```

- V C-ju:

```
#include <stdio.h>

int main()
{
  FILE *f = fopen("poskus.in", "rt");
  int i, j; fscanf(f, "%d %d", &i, &j); fclose(f);
  f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * (i + j));
  fclose(f); return 0;
}
```

- V C++:

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

int main()
{
  ifstream ifs("poskus.in"); int i, j; ifs >> i >> j;
  ofstream ofs("poskus.out"); ofs << 10 * (i + j);
  return 0;
}
```

(*Primeri rešitev se nadaljujejo na naslednji strani.*)

- V javi:

```
import java.io.*;
import java.util.Scanner;

public class Poskus
{
  public static void main(String[] args) throws IOException
  {
    Scanner fi = new Scanner(new File("poskus.in"));
    int i = fi.nextInt(); int j = fi.nextInt();
    PrintWriter fo = new PrintWriter("poskus.out");
    fo.println(10 * (i + j)); fo.close();
  }
}
```

- V C#:

```
using System.IO;

class Program
{
  static void Main(string[] args)
  {
    StreamReader fi = new StreamReader("poskus.in");
    string[] t = fi.ReadLine().Split(' '); fi.Close();
    int i = int.Parse(t[0]), j = int.Parse(t[1]);
    StreamWriter fo = new StreamWriter("poskus.out");
    fo.WriteLine("{0}", 10 * (i + j)); fo.Close();
  }
}
```

- V Visual Basic.NETu:

```
Imports System.IO

Module Poskus

  Sub Main()
    Dim fi As StreamReader = New StreamReader("poskus.in")
    Dim t As String() = fi.ReadLine().Split() : fi.Close()
    Dim i As Integer = Integer.Parse(t(0)), j As Integer = Integer.Parse(t(1))
    Dim fo As StreamWriter = New StreamWriter("poskus.out")
    fo.WriteLine("{0}", 10 * (i + j)) : fo.Close()
  End Sub

End Module
```

11. tekmovanje ACM v znanju računalništva za srednješolce

19. marca 2016

NALOGE ZA TRETJO SKUPINO

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

1. Letala (`letala.in`, `letala.out`)

Z letali bi radi prepeljali več zabojnikov iz kraja A v kraj B. Posamezno letalo lahko vsak dan odpelje največ en zabojnik iz A v B in se nato vrne nazaj v A. Imamo n zabojnikov in k letal. Za vsak zabojnik i poznamo njegovo maso m_i, za vsako letalo j pa poznamo njegovo nosilnost c_j (to pomeni, da lahko to letalo pelje le tiste zabojnike, katerih masa je manjša ali enaka c_j). V istem dnevu lahko pošljemo na pot več letal. **Napiši program**, ki izračuna najmanjše število dni, v katerih je mogoče v okviru teh omejitev prepeljati vse zabojnike iz A v B. Če pa jih sploh ni mogoče prepeljati, naj tvoj program izpiše -1.

Vhodna datoteka: v prvi vrstici sta dve celi števili, n in k, ločeni s presledkom. Zanju bo veljalo $1 \leq n \leq 100\,000$ in $1 \leq k \leq 100\,000$. V drugi vrstici je n celih števil, ločenih s po enim presledkom, ki podajajo mase zabojnikov. Za vsako maso velja $1 \leq m_i \leq 10^9$. V tretji vrstici je k celih števil, ločenih s po enim presledkom, ki podajajo nosilnosti letal. Za vsako nosilnost velja $1 \leq c_j \leq 10^9$.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število dni, v katerih je mogoče prepeljati vse zabojnike iz kraja A v kraj B. Če sploh ni mogoče prepeljati vseh zabojnikov, izpiši -1.

Primer vhodne datoteke:

```
10 3
20 100 60 30 40 90 80 50 10 70
45 120 30
```

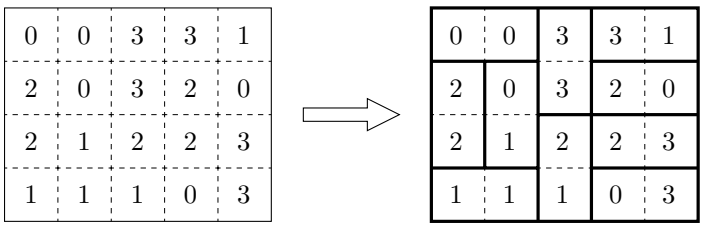
Pripadajoča izhodna datoteka:

```
6
```

2. Dominosa (`dominosa.in`, `dominosa.out`)

Pri igri Dominosa imamo pravokotno mrežo s sodim številom kvadratnih polj, v katerih so vpisana števila. Po dve sosednji polji lahko povežemo tako, da tvorita domino. Vsako domino lahko tvorimo vodoravno ali navpično. Naša naloga je, da iz vseh števil v mreži tvorimo domine tako, da uporabimo vsa polja (vsako natanko enkrat) in da nimamo dveh enakih domin (domini, ki vsebujeta isti dve številki).

Naslednja slika kaže primer mreže 5×4 polj, ki smo jo uspešno razdelili na same različne domine:



Če se omejimo na domine, ki imajo na posameznem polju od 0 do n pik, je največji možni pravokotnik, pri katerem je problem sploh še lahko rešljiv, velikosti $(n+2) \times (n+1)$. Pri naši nalogi bodo testni primeri vedno takšne maksimalne velikosti; pri vsakem testnem primeru torej dobiš nek n in nato pravokotnik števil (od 0 do n) z $n+1$ vrsticami in $n+2$ stolpci.

Na primer, pri $n = 3$ je možnih 10 domin — od $[0\,|\,0]$ do $[3\,|\,3]$ — in igrali bi se na mreži velikosti 5×4; pri $n = 4$ je možnih 15 domin — od $[0\,|\,0]$ do $[4\,|\,4]$ — in igrali bi se na mreži velikosti 6×5; itd.

Vhodna datoteka: v prvi vrstici je celo število n (zanj velja $3 \leq n \leq 9$), ki pove maksimalno število pik na posameznem polju pri tem testnem primeru. Sledi $n + 1$ vrstic, v vsaki od teh pa je $n + 2$ števil (cela števila od 0 do n), ločenih s po enim presledkom. Te vrstice podajajo vsebino pravokotne mreže, ki jo moraš razdeliti na domine.

Izhodna datoteka: izpiši mrežo tako, da med vsaki dve sosednji vrstici oz. stolpca mreže vrineš še po eno vrstico oz. stolpec znakov „`.`“ (pika); kjer pa dve sosednji polji tvorita eno domino, izpiši med njiju znak „`-`“ (če sta v isti vrstici) ali „`|`“ (če sta v istem stolpcu).

Vsi testni primeri pri tej nalogi bodo izbrani tako, da je mrežo gotovo mogoče razdeliti na domine v skladu z zahtevami naloge. Če je možnih več rešitev (več različnih razdelitev mreže na domine), je vseeno, katero od njih izpišeš.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
0-0.3.3-1
....|....
2.0.3.2-0
|.|......
2.1.2.2-3
....|....
1-1.1.0-3
```

3. Galaktična zavezništva (`xor.in`, `xor.out`)

V galaksiji je več stoletij vladal red. Galaktična republika je svojo vojsko že davno razpustila ter ga skozi čas uspešno vzdrževala na miren in diplomatski način. A razmere so se začele spreminjati, saj število pristašev temne strani strmo narašča. Senat republike je zato na izrednem zasedanju enoglasno odločil, naj se ponovno ustanovi enotna republikanska vojska, dovolj mogočna, da bo zatrla vsak morebiten poskus vzpona temne strani sile. Vsakemu izmed planetov, včlanjenih v republiko, je bilo določeno, naj zbere svoje najboljše vojaške strategе, ki se bodo na izboru na planetu Croissant potegovali za zasedbo elitnih položajev v novonastali vojaški hierarhiji. Položaji so oštevilčeni po pomembnosti od 1 do m, kjer je mesto 1 najpomembnejše.

Vsak izmed n planetov na izbor pošlje vojaške strategе, kjer je vsak izurjen za zasedbo natanko določenega položaja. Kandidate planeta a lahko tako opišemo z nizom m bitov, $a = a_1 a_2 \ldots a_m$, kjer je $a_i = 1$, če ima planet a za i-ti položaj izurjenega stratega, sicer pa je $a_i = 0$. Ker si planeti želijo imeti v vojski čim večji vpliv, so pripravljeni sklepati zavezništva.

V zavezništvo se planeti vselej povezujejo v trojicah. Predstavljajmo si tri planete, ki nameravajo skleniti zavezništvo, s pripadajočimi nizi kandidatov $a = a_1 \ldots a_m$, $b = b_1 \ldots b_m$ in $c = c_1 \ldots c_m$. Če imata na i-tem položaju natanko dva izmed planetov svoja kandidata, se ta dva spreta in odideta, tako da potem celotno zavezništvo na tem položaju nima nikogar. Če ima na i-tem položaju svojega kandidata tudi tretji planet, le-ta zasede ravnokar izpraznjeno mesto. Položaje, na katerih ima zavezništvo svoje kandidate, lahko torej opišemo z binarnim nizom $d = d_1 \ldots d_m$, definiranim takole: bit d_i je prižgan ($d_i = 1$), če je izmed bitov a_i, b_i, c_i prižgan natanko eden ali pa vsi trije; če pa sta od treh bitov prižgana natanko dva ali nobeden, potem je bit d_i ugasnjen ($d_i = 0$). (Z drugimi besedami, niz d dobimo tako, da XORamo med sabo nize a, b in c).

Dve zavezništvi, recimo $d = d_1 d_2 \ldots d_m$ in $d' = d'_1 d'_2 \ldots d'_m$, lahko po moči primerjamo takole: poiščimo najpomembnejši položaj, pri katerem se ti dve zavezništvi razlikujeta (najmanjši i, pri katerem je $d_i \neq d'_i$). Tisto zavezništvo, ki ima na tem mestu prižgan bit, je močnejše od tistega, ki ima na tem mestu ugasnjen bit.

Napiši program, ki prebere podatke o več planetih in sestavi iz njih najmočnejše možno zavezništvo treh planetov.

Vhodna datoteka: v prvi vrstici sta dve celi števili, najprej n in nato m, ločeni s presledkom. Pri tem je n število planetov (zanj velja $3 \leq n \leq 7500$), m pa število položajev (zanj velja $1 \leq m \leq 50$). Sledi n vrstic, za vsak planet po ena. V vsaki od teh vrstic je m znakov `0` ali `1`; če je i-ti znak v neki vrstici enak `1`, to pomeni, da ima tisti planet na položaju i svojega človeka, sicer (torej i-ti znak v tej vrstici enak `0`) pa ga nima.

Izhodna datoteka: vanjo izpiši niz m ničel in enic, ki opisuje najmočnejše možno zavezništvo treh planetov, kar jih je mogoče sestaviti iz n planetov v vhodni datoteki.

Primer vhodne datoteke:

```
5 8
10101000
01001100
10101000
00101110
11000010
```

Pripadajoča izhodna datoteka:

```
11001010
```

Komentar: najmočnejše zavezništvo je pri tem primeru tisto, ki ga sestavljajo prvi, drugi in četrti planet.

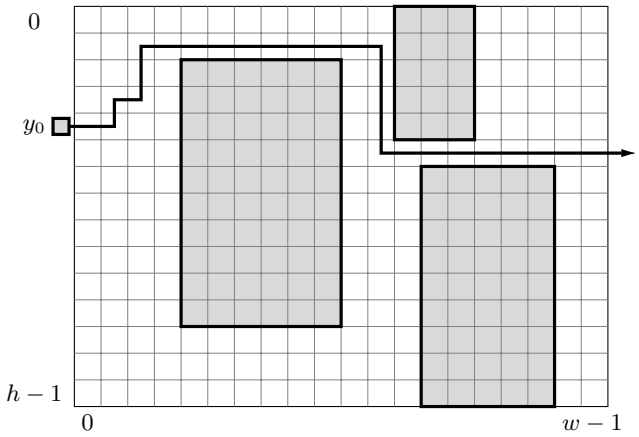
4. Asteroidi (`asteroidi.in`, `asteroidi.out`)

V dvodimenzionalni vesoljski simulaciji krmilimo majhno vesoljsko plovilo skozi polje n asteroidov. Medtem ko počasi plujemo skozi polje naprej vzdolž x-osi (v smeri nazaj se ni dovoljeno premikati), se s hitrimi premiki vzdolž y-osi izogibamo asteroidom. Omejeni smo tudi z zgornjim in spodnjim robom polja asteroidov, ki v simulaciji sovpada z zgornjim in spodnjim robom zaslona, v realnem svetu pa bi nas onkraj roba čakala meglica, v kateri bi se izgubili, ali pa bi nas pojedla vesoljska koza. Eden od posebnih izzivov je prečkati polje z minimalnim premikanjem vzdolž y-osi, to je z minimalno vsoto absolutnih vrednosti premikov vzdolž y-osi.

Napiši program, ki bo prebral dimenzije polja asteroidov, položaje in velikosti asteroidov ter izpisal najmanjšo vsoto premikov, s katero lahko prečkamo polje.

V našem poenostavljenem prikazu simulacije je polje asteroidov predstavljeno kot velika pravokotna karirasta mreža dimenzij $w \times h$ (širina w, višina h). V mrežo vpeljimo koordinatni sistem: vrstice oštevilčimo od 0 (najbolj zgornja vrstica) do $h-1$ (najbolj spodnja vrstica), stolpce pa od 0 (najbolj levi stolpec) do $w-1$ (najbolj desni stolpec). Asteroide predstavljajo manjši pravokotniki, podani s koordinatami zgornjega levega in spodnjega desnega oglišča. Raketo si predstavljamo kot kvadrat velikosti 1×1. Začetni položaj rakete je $(-1, y_0)$ za neko podano koordinato (številko vrstice) y_0; od tam se raketa lahko premika po en kvadratek v desno, med temi premiki pa naredi še premike za poljubno število kvadratkov gor ali dol. Prvi premik mora vedno biti v desno, z $(-1, y_0)$ na $(0, y_0)$. Če se raketa po nekem premiku gor ali dol ne more premakniti naprej v smeri desno, ne da bi se zaletela v asteroid, je obtičala in je igre konec. Vse koordinate in dimenzije so nenegativna cela števila.

Naslednja slika kaže primer polja asteroidov (z $w = 20$, $h = 15$, $y_0 = 4$ in $n = 3$) in ene od možnih optimalnih poti rakete skozenj:



Vhodna datoteka: v prvi vrstici so podane višina h, širina w, začetna y-koordinata rakete y_0 (zanjo velja $0 \le y_0 \le h-1$) ter število asteroidov n. Veljalo bo $1 \le w \le 10^9$, $0 \le y_0 < h \le 10^9$ ter $1 \le n \le 300$. V 80 % primerov bo veljalo tudi $w \le 10^6$ in $h \le 10^5$. V 60 % primerov bo veljalo tudi $w \le 10^6$ in $h \le 10^3$. V 40 % primerov bo veljalo tudi $w \le 10^3$, $h \le 10^3$ ter $n \le 100$.

V naslednjih n vrsticah pa so podane koordinate zgornjega levega in spodnjega desnega kota i-tega asteroida $x_{1i}, y_{1i}, x_{2i}, y_{2i}$. Pri vsakem asteroidu i bo veljalo $0 \le x_{1i} \le x_{2i} < w$ in $0 \le y_{1i} \le y_{2i} < h$.

Asteroidi se ne prekrivajo, lahko pa se dotikajo.

Izhodna datoteka: poišči pot z najmanjšo skupno vsoto absolutnih vrednosti premikov v y-smeri in izpiši to vsoto v izhodno datoteko. Če nikakor ni možno priti skozi polje asteroidov, izpiši -1.

Primer vhodne datoteke:
(to je primer z gornje slike)

```
15 20 4 3
4 2 9 11
12 0 14 4
13 6 17 14
```

Pripadajoča izhodna datoteka:

```
7
```

5. Brisanje niza (`brisanje.in`, `brisanje.out`)

Dan je nek niz, ki ga sestavljajo same male črke angleške abecede. Iz njega smemo pobrisati enega ali več zaporednih znakov, vendar le, če so vsi enaki. Tako dobimo nek krajši niz, iz katerega lahko spet kaj pobrišemo in tako naprej. Prej ali slej lahko na ta način pridemo do praznega niza (iz katerega ne moremo pobrisati ničesar več).

Napiši program, ki prebere vhodni niz in ugotovi, kolikšno je najmanjše potrebno število brisanj, s katerimi lahko iz njega naredimo prazen niz.

Primer: če začnemo z nizom `aabbbacaa`, lahko na primer brišemo takole (na vsakem koraku je podčrtan tisti del niza, ki ga bomo naslednjega pobrisali):

$$\texttt{aabbb}\underline{\texttt{a}}\texttt{caa} \to \texttt{aa}\underline{\texttt{bbb}}\texttt{caa} \to \underline{\texttt{aa}}\texttt{caa} \to \underline{\texttt{c}}\texttt{aa} \to \underline{\texttt{aa}} \to \texttt{""}$$

Tu smo torej porabili 5 brisanj. Gre pa tudi s samo 3 brisanji:

$$\texttt{aabbba}\underline{\texttt{c}}\texttt{aa} \to \texttt{aa}\underline{\texttt{bbb}}\texttt{aaa} \to \underline{\texttt{aaaaa}} \to \texttt{""}$$

Vhodna datoteka: v prvi vrstici je celo število n, ki pove dolžino vhodnega niza pri tem testnem primeru. Veljalo bo $1 \leq n \leq 1000$. Pri 40 % testnih primerov bo veljalo tudi $n \leq 10$. V drugi vrstici je vhodni niz, ki bi ga radi pobrisali; dolg je n znakov, vsi ti znaki pa so male črke angleške abecede (od `a` do `z`).

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število brisanj, s katerimi je mogoče niz iz vhodne datoteke popolnoma pobrisati (tako, da iz njega nastane prazen niz).

Primer vhodne datoteke:

```
9
aabbbacaa
```

Pripadajoča izhodna datoteka:

```
3
```

11. tekmovanje ACM v znanju računalništva za srednješolce

19. marca 2016

REŠITVE NALOG ZA PRVO SKUPINO

1. Tipkanje

Ko beremo zaporedje besed, je poleg trenutne besede koristno hraniti še prejšnjo besedo in njeno dolžino. Ko preberemo novo besedo, primerjajmo istoležne znake prejšnje in trenutne besede, dokler ne opazimo prvega neujemanja (ali pa pridemo do konca kakšne od besed). Recimo, da je bila prejšnja beseda dolga p znakov, trenutna je dolga d znakov, ujemata pa se v prvih u znakih. To pomeni, da bi moral naš zapisnikar po izpisu prejšnje besede pobrisati zadnjih $p-u$ znakov iz vnosnega polja (s tipko Backspace), nato natipkati zadnjih $d-u$ znakov nove besede in nato pritisniti Enter. Število pritiskov na tipke se torej poveča za $(p-u)+(d-u)+1$. Ta postopek ponavljamo v zanki, dokler ne obdelamo vseh n besed.

Spodnji program v C-ju hrani obe besedi (prejšnjo in trenutno) v spremenljivki tabela; vsak od kazalcev beseda in prejBeseda kaže na eno vrstico tabele, pred branjem nove besede pa kazalca zamenjamo, tako da prejBeseda kaže na tisto besedo, na katero je prej kazala beseda (ker tista beseda ni več trenutna, ampak je zdaj prejšnja), ta pa kaže na vrstico, na katero je prej kazala prejBeseda; tam je zdaj že predprejšnja beseda, ki jo lahko povozimo z novo besedo (ki jo bomo vsak hip prebrali). Tako si prihranimo nekaj časa, ker besed ni treba kopirati iz ene tabele v drugo.

```
#include <stdio.h>
#define MaxDolz 100

int main()
{
  char tabela[2][MaxDolz + 1], *beseda = tabela[0], *prejBeseda = tabela[1], *t;
  int n, dolz, prejDolz, ujemanje, rezultat = 0;

  scanf("%d\n", &n); /* Preberimo število besed. */
  *beseda = 0; dolz = 0; /* Na začetku je vnosno polje prazno. */

  while (n-- > 0)
  {
    /* Prejšnjo besedo si zapomnimo v prejBeseda, njeno dolžino pa v prejDolz. */
    t = beseda; beseda = prejBeseda; prejBeseda = t; prejDolz = dolz;

    /* Preberimo novo besedo. */
    scanf("%s\n", beseda);

    /* Preštejmo, v koliko znakih se ujema s prejšnjo. */
    ujemanje = 0;
    while (beseda[ujemanje] && beseda[ujemanje] == prejBeseda[ujemanje]) ujemanje++;

    /* Poglejmo, kako dolga je ta beseda. */
    dolz = ujemanje; while (beseda[dolz]) dolz++;

    /* Po izpisu prejšnje besede moramo torej (prejDolz - ujemanje)-krat pritisniti Backspace,
       natipkati zadnjih (dolz - ujemanje) znakov nove besede in nato še pritisniti Enter. */
    rezultat += (prejDolz - ujemanje) + (dolz - ujemanje) + 1;
  }

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

2. Zoom

Niz s, ki opisuje potek zoomiranja, pregledujmo v zanki po znakih in vsakič ustrezno popravimo koordinate opazovanega območja. Izračunajmo mejo med levo in desno polovico, $x_m = (x_1 + x_2)/2$; in mejo med zgornjo in spodnjo polovico, $y_m = (y_1 + y_2)/2$. Če

se moramo premakniti v eno od levih dveh četrtin (0 in 2), se desni rob našega območja premakne z x_2 na x_m, sicer (če se premikamo v eno od desnih dveh četrtin, to sta 1 in 3)) pa se levi rob premakne z x_1 na x_m. Podobno je tudi pri y-koordinatah. Odvisno od trenutnega znaka niza s moramo torej popraviti eno od koordinat x_1, x_2 ter eno od koordinat y_1, y_2. Ko pridemo do konca niza s, moramo dobljene koordinate le še izpisati.

```
#include <stdio.h>
void Zoom(char *s, double x1, double y1, double x2, double y2)
{
  double xm, ym;
  for ( ; *s; s++)
  {
    xm = (x1 + x2) / 2; ym = (y1 + y2) / 2;
    if (*s == '0' || *s == '1') y1 = ym; else y2 = ym;
    if (*s == '0' || *s == '2') x2 = xm; else x1 = xm;
  }
  printf("[%g, %g] x [%g, %g]\n", x1, x2, y1, y2);
}
```

3. Zaklepajski izrazi

Vhodni niz je koristno brati od konca proti začetku, pri tem pa vzdrževati seznam oz. sklad s podatki o tem, kateri oklepaji (kakšnih oblik) so trenutno odprti (torej da smo prebrali tisti zaklepaj, nismo pa še ustvarili pripadajočega oklepaja zanj). Ko pridemo do zvezdice, poglejmo na vrh sklada; oklepaj, ki ga bomo naredili iz te zvezdice, se mora ujemati z zaklepajem na vrhu sklada, sicer naš niz ne bo pravilno gnezden. Tako torej vemo, kakšne vrste oklepaj narediti iz trenutne zvezdice, zaklepaj z vrha sklada pa lahko pobrišemo, saj smo ga zdaj zaprli z oklepajem. Če se kdaj zgodi, da pridemo do zvezdice, sklad pa je prazen, lahko takoj odnehamo in vemo, da se niza ne da predelati v veljaven oklepajski izraz. Ko smo pregledali že cel niz od konca proti začetku, moramo le še preveriti, če je sklad takrat prazen; če ni, to pomeni, da je v vhodnem nizu preveč zaklepajev in ga tudi ne moremo predelati v veljaven oklepajski izraz.

Spodnji podprogram hrani sklad v tabeli sklad, število znakov na njem pa v spremenljivki sp.

```
void Dopolni(char *s)
{
  int n = strlen(s), i, sp = 0;
  char *sklad = (char *) malloc(n);
  for (i = n - 1; i >= 0; i--)
  {
    /* Če je trenutni znak zaklepaj, ga dodamo na sklad. */
    if (s[i] != '*') { sklad[sp++] = s[i]; continue; }
    /* Sicer imamo zvezdico. Če je sklad prazen, je niz neveljaven. */
    if (sp == 0) break;
    /* Pobrišimo zaklepaj z vrha sklada in spremenimo trenutno zvezdico
       v pripadajoči oklepaj. */
    sp--;
    if (sklad[sp] == ')') s[i] = '(';
    else if (sklad[sp] == ']') s[i] = '[';
    else if (sklad[sp] == '}') s[i] = '{';
    else if (sklad[sp] == '>') s[i] = '<';
  }
  if (i < 0 && sp == 0) printf("%s\n", s);
  else printf("Problem je nerešljiv.\n");
  free(sklad);
}
```

Gre pa tudi brez ločene tabele s skladom. Vsi znaki našega sklada so v resnici zaklepaji, ki smo jih dobili iz že pregledanega dela vhodnega niza s. Dovolj je, če si zapomnimo indeks, na katerem se v nizu s nahaja zaklepaj, ki je trenutno na vrhu sklada; spodnji

podprogram ga hrani v spremenljivki z. Poleg tega pa si v spremenljivki d zapomni, koliko elementov je na skladu oz. z drugimi besedami: kako globoko je v nizu s vgnezden zaklepaj na indeksu z. Ko pobrišemo element s sklada, se moramo le zapeljati z z-jem naprej po nizu s, dokler ne naletimo na prvi tak zaklepaj, ki je vgnezden za en nivo plitveje.

```
void Dopolni2(char *s)
{
  int i, z, n = 0, d, dd; while (s[n]) n++;
  z = 0; d = 0; /* z = trenutni zaklepaj na vrhu sklada; d = globina sklada. */
  for (i = n - 1; i >= 0; i--)
  {
    /* Če je trenutni znak zaklepaj, ga dodamo na sklad. */
    if (s[i] != '*') { z = i; d++; continue; }
    /* Sicer imamo zvezdico. Če je sklad prazen, je niz neveljaven. */
    if (d == 0) break;
    /* Popravimo trenutni znak na oklepaj, ki se ujema z zaklepajem z vrha sklada. */
    if (s[z] == ')') s[i] = '(';
    else if (s[z] == ']') s[i] = '[';
    else if (s[z] == '}') s[i] = '{';
    else if (s[z] == '>') s[i] = '<';
    /* Pobrišimo zaklepaj z vrha sklada. */
    z++; d--; dd = d;
    for ( ; z < n; z++)
      if (s[z] == '(' || s[z] == '[' || s[z] == '{' || s[z] == '<') d++;
      else if (d == dd) break;
      else d--;
  }
  if (i < 0 && d == 0) printf("%s\n", s);
  else printf("Problem je nerešljiv.\n");
}
```

4. Izštevanka

Med simulacijo izštevanja bomo v tabeli ziv hranili za vsakega otroka podatek o tem, koliko življenj še ima. Na začetku postavimo vse elemente tabele na 2, ko pa pri izštevanju pridemo do nekega otroka, mu število življenj zmanjšamo za 1. Če pri tem pade na 0, se ustavimo in tega otroka izpišemo.

Spodnji program ima otroke oštevilčene od 0 do $n-1$ namesto od 1 do n. Tako lahko njihove številke neposredno uporabimo kot indekse v tabelo ziv. Z njimi je tudi zelo lahko računati: če smo bili pri otroku i in se pomaknemo za k mest naprej, pridemo do otroka $i+k$. Ker otroci stojijo v krogu, številka n predstavlja spet otroka 0, številka $n+1$ otroka 1 in tako naprej; z drugimi besedami, po vsakem koraku izštevanja moramo od številke otroka obdržati le ostanek po deljenju z n. Na koncu moramo pri izpisu paziti še na to, da številki prištejemo 1, tako da bomo imeli izpis v razponu od 1 do n (kot zahteva naloga) namesto od 0 do $n-1$.

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

int main()
{
  int n, k, i, *ziv;
  scanf("%d %d", &n, &k);    /* Preberimo vhodne podatke. */
  /* Pripravimo si tabelo, v kateri bomo za vsakega otroka
     hranili število življenj. Na začetku ima vsak 2 življenji. */
  ziv = (int *) malloc(n * sizeof(int));
  for (i = 0; i < n; i++) ziv[i] = 2;
  /* Odsimulirajmo potek izštevanja. Namesto od 1 do n imamo
     otroke oštevilčene od 0 do n - 1. */
```

```
  i = 0;
  while (ziv[i] > 0)
  {
    i = (i + k) % n;          /* Naredimo n korakov naprej od trenutnega otroka. */
    --ziv[i];                 /* Zmanjšajmo mu število življenj za 1. */
    k++;                      /* Povečajmo dolžino izštevanke. */
  }
  printf("%d\n", i + 1);      /* Izpišimo, pri katerem otroku smo končali. */
  free(ziv); return 0;        /* Pospravimo za sabo. */
}
```

5. H-indeks

Preprosta rešitev je, da v zanki gledamo vse večje h-je in za vsakega preverimo, ali obstaja vsaj h člankov, od katerih ima vsak po vsaj h citatov. Za to preverjanje potrebujemo še eno vgnezdeno zanko, ki gre po vseh člankih in šteje, koliko jih ima vsaj h citatov:

```
int HIndeks1(int seznam[], int n)
{
  int i, k, h;
  for (h = 1; ; h++)
  {
    /* Na tem mestu vemo, da obstaja vsaj h − 1 člankov s po vsaj h − 1 citati.
       Preverimo, ali obstaja vsaj h člankov s po vsaj h citati. */
    for (i = 0, k = 0; i < n; i++)
      if (seznam[i] >= h) k++;
    /* Zdaj vemo, da obstaja k člankov s po vsaj h citati. Če jih je manj kot h,
       potem h že ni več primeren h-indeks, torej moramo vrniti h − 1. */
    if (k < h) return h − 1;
  }
}
```

Zanka se gotovo prej ali slej konča, saj k ne more biti večji od dolžine seznama, torej n, tako da bo pogoj **if** (k < h) najkasneje pri $h = n + 1$ gotovo izpolnjen. Časovna zahtevnost tega postopka je v najslabšem primeru $O(n^2)$ — zunanja zanka izvede največ $n + 1$ iteracij, notranja pa pri vsaki od njih po n iteracij. Približno tako rešitev smo na tekmovanju tudi pričakovali od tekmovalcev prve skupine; vseeno pa si zdaj oglejmo še nekaj učinkovitejših rešitev.

Ni nujno, da h-je preverjamo po vrsti od 1 naprej, kot to počne gornji postopek. Največji primerni h lahko poiščemo tudi z bisekcijo:

```
int HIndeks1b(int seznam[], int n)
{
  int i, k, h, L, R;
  L = 0; R = n + 1;
  while (R − L > 1)
  {
    /* Na tem mestu vemo, da obstaja vsaj L člankov s po vsaj L citati
       in da ne obstaja vsaj R člankov s po vsaj R citati.
       Pravi h-indeks je torej nekje na območju od L do R − 1. */
    h = (L + R) / 2;
    for (i = 0, k = 0; i < n; i++)
      if (seznam[i] >= h) k++;
    /* Zdaj vemo, da obstaja k člankov s po vsaj h citati. */
    if (k >= h) L = h;
    else R = h;
  }
  return L;
}
```

Ta postopek torej vzdržuje spodnjo in zgornjo mejo možnih vrednosti h-ja: v vsakem trenutku vemo, da obstaja vsaj L člankov s po L citati in da ne obstaja vsaj R člankov

s po R citati, torej je pravi h-indeks večji ali enak L ter manjši od R. Nato vzamemo nek h, ki je približno na polovici med L in R. Če obstaja vsaj h člankov s po h citati, lahko L povečamo do tega h, sicer pa lahko R zmanjšamo do tega h. Na ta način v vsaki iteraciji zunanje zanke približno razpolovimo interval možnih vrednosti h-ja. Po približno $\log_2 n$ iteracijah se ta interval zmanjša na eno samo vrednost, torej $R = L + 1$, pri čemer vemo, da obstaja L člankov s po L citati, ne obstaja pa $L + 1$ člankov s po $L + 1$ citati. Torej je L ravno iskani h-indeks in ga lahko vrnemo kot rezultat našega postopka. Časovna zahtevnost je zdaj le $O(n \log n)$, ker ima zunanja zanka le $O(\log n)$ iteracij (notranja pa še vedno vsakič po $O(n)$, enako kot pri prvotni rešitvi).

Še ena možnost je, da članke uredimo padajoče po številu citatov. Recimo, da jih v tem vrstnem redu označimo z $x_1 \geq x_2 \geq \ldots \geq x_n$. Če zdaj pri nekem h velja neenačba $x_h \geq h$, potem vemo, da ima vsaj h člankov (namreč članki $x_1, \ldots, x_h$) po vsaj h citatov, torej je h eden od možnih kandidatov za h-indeks. Po drugi strani, če velja $x_h < h$, potem gotovo ne drži, da bi imelo vsaj h člankov po h citatov, kajti članki $x_{h+1}, \ldots x_n$ nimajo nič več citatov kot x_h, tako da so edini, ki bi utegnili imeti vsaj h citatov, članki $x_1, \ldots, x_{h-1}$, teh pa je premalo (samo $h-1$, ne pa h). V tem primeru (pri $x_h < h$) torej h ni primeren kandidat za h-indeks. Vse, kar moramo narediti, je torej poiskati največji h, pri katerem velja $x_h \geq h$.

```
int HIndeks2(int seznam[], int n)
{
  int h;
  UrediSeznamPadajoce(seznam, n);
  for (h = 0; h < n; h++)
    if (seznam[h] <= h) break;
  return h;
}
```

S podrobnostmi urejanja se tu ne bomo ukvarjali — lahko sami implementiramo kakšnega od postopkov urejanja, lahko pa uporabimo kaj iz standardne knjižnice našega programskega jezika. V C-ju imamo na primer funkcijo qsort:

```
int Primerjaj(const void *a, const void *b) { return *(int *) b - *(int *) a; }
⋮
qsort(seznam, n, sizeof(seznam[0]), &Primerjaj);
```

V C++ pa je še bolje uporabiti funkcijo sort:

```
std::sort(seznam, seznam + n, std::greater<int>());
```

V časovni zahtevnosti naše nove rešitve prevladuje čas urejanja. Učinkoviti postopki urejanja porabijo $O(n \log n)$ časa, zato je taka zdaj tudi časovna zahtevnost naše rešitve.

Ko je seznam enkrat urejen padajoče, lahko največji h poiščemo tudi z bisekcijo, podobno kot prej v rešitvi HIndeks1b:

```
int HIndeks2b(int seznam[], int n)
{
  int h, L, R;
  UrediSeznamPadajoce(seznam, n);
  L = 0; R = n + 1;
  while (R - L > 1)
  {
    h = (L + R) / 2;
    if (seznam[h - 1] >= h) L = h;
    else R = h;
  }
  return L;
}
```

Od tega sicer v tej rešitvi ni velike koristi, ker večino časa tako ali tako porabimo za urejanje. Je pa ta ideja z bisekcijo koristna, ker nas pripelje do naslednje, še boljše

rešitve. Izkaže se, da seznama sploh ni treba popolnoma urediti. Našemu prvotnemu seznamu recimo X; izberimo si poljuben element tega seznama, na primer m. Preštejmo, koliko elementov seznama X je večjih od m (recimo, da jih je a), koliko je enakih m (recimo, da jih je b), in koliko jih je manjših od m (recimo, da jih je c); pri tem seveda velja $a+b+c=n$, kjer je n skupna dolžina seznama X.

Seznama X s tem nismo uredili, vendarle pa zdaj vemo, da če bi ga uredili padajoče, bi bili v njem elementi $x_1, \ldots, x_a$ večji od m, elementi $x_{a+1}, \ldots, x_{n-c}$ enaki m (indeks $n-c$ je seveda enak $a+b$) in elementi $x_{n-c+1}, \ldots, x_n$ manjši od m.

Zdaj lahko torej pogoj $x_h \geq h$ preverimo za poljuben h z območja od $a+1$ do $a+b$ (oz. do $n-c$, kar je enako), saj za te indekse vemo, kakšne so tam vrednosti elementov x_h — namreč enake so m. Če velja $m \geq a+b$, to pomeni, da je pogoj $x_h \geq h$ izpolnjen pri $h = a+b$ in zato tudi pri vseh nižjih h, tako da se moramo v nadaljevanju ukvarjati le z višjimi indeksi h, torej v tistem delu urejenega seznama, kjer so elementi z vrednostmi, manjšimi od m.

Podobno, če velja $m \leq a$, to pomeni, da pogoj $x_h \geq h$ ni izpolnjen pri nobenem h-ju z območja od $a+1$ do $a+b$, zato moramo naš h-indeks iskati pri nižjih vrednostih h, torej v tistem delu urejenega seznama, kjer so elementi z vrednostmi, večjimi od m.

Ostane pa še možnost, da je m nekje vmes: $a+1 \leq m < a+b$. To pomeni, da pri indeksu $h = m$ pogoj $x_h \geq h$ še velja (tam velja celo enakost: obe strani sta enaki m), pri $h = m+1$ pa ne več (leva stran je še vedno enaka m, desna pa je zdaj večja od m); takrat torej lahko zaključimo, da je h-indeks našega seznama enak m in lahko postopek končamo.

Zapišimo dobljeni postopek s psevdokodo.

vhod: tabela $X = [x_1, \ldots, x_n]$, ne nujno urejena;
1 $L := 0$; $R := n$;
2 **while true**:
(* *Na tem mestu velja:*
elementi v $X[1 \ldots L]$ *so strogo večji od tistih v* $X[L+1 \ldots n]$*;*
elementi v $X[L+1 \ldots R]$ *so strogo večji od tistih v* $X[R+1 \ldots n]$*;*
če bi tabelo povsem uredili padajoče, bi veljal pogoj $x_h \geq h$ *za vse* h
z območja od 1 do L *in za noben* h *z območja od* $R+1$ *do* n. *)
3 **if** $R = L$ **then return** L;
4 $m :=$ poljuben element izmed $x_{L+1}, \ldots, x_R$;
5 prerazporedi vsebino območja $X[L+1 \ldots R]$ tako, da pridejo najprej tisti elementi, ki so večji od m (recimo, da je takih a), nato tisti, ki so enaki m (recimo, da je takih b), in nato tisti, ki so manjši od m (recimo, da je takih c);
6 **if** $m \geq L+a+b$ **then** $L := L+a+b$
7 **else if** $m \leq L+a$ **then** $R := R-b-c$
8 **else return** m;

V prvi iteraciji te zanke torej gledamo celoten seznam X, kasneje pa nek njegov krajši del, od indeksa $L+1$ do R. V vsakem trenutku si torej lahko mislimo seznam sestavljen iz treh delov, $[1 \ldots L]$, $[L+1 \ldots R]$ in $[R+1 \ldots n]$. Seznama ne bomo zares uredili padajoče, so pa ti trije deli že v pravem vrstnem redu: vsi elementi v prvem delu so strogo večji od vseh v drugem in tretjem, vsi elementi v drugem delu so strogo večji od tistih v tretjem. Če bi torej vsakega od teh delov samega zase uredili padajoče, bi bila s tem tudi cela tabela urejena padajoče. Vemo tudi, da bi bil tedaj pogoj $x_h \geq h$ izpolnjen povsod v levem delu in nikjer v desnem delu, za srednji del pa še ne vemo in z njim se bomo ukvarjali v nadaljevanju zanke.

V vrstici 5 razdelimo srednji del, $[L+1 \ldots R]$, na tri kose s pomočjo elementa m: elementi na območju $[L+1 \ldots L+a]$ so večji od m, tisti na območju $[L+a \ldots L+a+b]$ so enaki m in tisti na območju $[L+a+b+1 \ldots R]$ so manjši od m. Če je pogoj v vrstici 6 izpolnjen, to pomeni, da velja pogoj $x_h \geq h$ povsod v levih dveh kosih, zato lahko tadva kosa pritaknemo k levemu delu in se v naslednji iteraciji ukvarjamo le z desnim kosom. Podobno, če je izpolnjen pogoj v vrstici 7, to pomeni, da ne velja pogoj $x_h \geq h$ nikjer v desnih dveh kosih, zato lahko tadva kosa pritaknemo k desnemu delu in se v naslednji iteraciji ukvarjamo le z levim kosom. Če pa pridemo do vrstice 8, vemo, da velja pogoj

$x_h \geq h$ povsod v levem kosu, nikjer v desnem, v srednjem kosu pa so vsi elementi enaki m in lahko takoj zaključimo, da je to naš H-indeks.

Vrstica 3 preveri, če se je srednji del že izpraznil ($L = R$); takrat poznamo stanje pogoja $x_h \geq h$ že za celo (padajoče urejeno) tabelo: velja povsod od 1 do L in nikjer od $R+1$ do n, vmes pa ni nobenega indeksa več. Takrat torej vemo, da je največji primerni h enak $h = L$.

Postopek se gotovo ustavi: ker za m vsakič izberemo neko vrednost, ki se v srednjem delu tabele res pojavlja, bo $b > 0$ in zato se bo po tej iteraciji zanke gotovo povečal bodisi levi bodisi desni del tabele. Razpon $R - L$ se torej v vsaki iteraciji zanke zmanjša in če se zanka ne konča že prej z vrstico 8, se bo končala najkasneje tedaj, ko bo $R - L$ padel na 0 (v vrstici 3).

Da bo postopek hiter, je pametno v vrstici 5 za m vzeti mediano, torej srednjo vrednost izmed vseh na območju $[L+1 \ldots R]$. Za mediano velja, da je kvečjemu polovica elementov večjih od nje in kvečjemu polovica manjših od nje. Del tabele, s katerim se bomo ukvarjali v naslednji iteraciji, bo zato kvečjemu pol tolikšen kot v trenutni iteraciji. V prvi iteraciji imamo torej del dolžine n, v drugi del dolžine $\leq n/2$, nato $\leq n/4$ in tako naprej; vse to se sešteje v $\leq 2n$. Tako za iskanje mediane v vrstici 4 kot za preurejanje srednjega dela tabele v vrstici 5 potrebujemo linearno mnogo časa v odvisnosti od $R-L$, torej trenutne dolžine srednjega dela. Ker se te dolžine, kot smo ravnokar videli, po vseh iteracija skupaj seštejejo v največ $2n$, lahko zaključimo, da naš postopek porabi vsega skupaj le $O(n)$ časa.

REŠITVE NALOG ZA DRUGO SKUPINO

1. Sorodstvo

Za začetek je koristno zapise urediti naraščajoče po letu rojstva, saj iz omejitev v nalogi sledi, da je lahko oseba potencialni otrok le tistih oseb, ki so bile rojene pred njo. Recimo, da v tem vrstnem redu zapise oštevilčimo od 1 do n. Naj bo a_i dolžina najdaljše take verige, ki se začne z osebo i. Ena možnost je, da se veriga s to osebo tudi konča, tedaj je njena dolžina 1. Lahko pa se veriga nadaljuje pri neki osebi j (kar je sicer mogoče le, če velja $r_i + d \leq r_j < s_i$); ker je najdaljša veriga z začetkom pri osebi j dolga a_j, bo zdaj naša veriga z začetkom pri i dolga $a_j + 1$. Med vsemi temi možnostmi vzemimo za a_i tisto, ki je največja (torej ki dá najdaljšo verigo). Lahko si tudi zapomnimo (recimo kot n_i), pri katerem j smo ta maksimum dosegli; s pomočjo teh podatkov bomo kasneje lahko najdaljšo verigo tudi izpisali.

Vidimo lahko, da so nam pri izračunu a_i prišle prav vrednosti a_j za tiste osebe j, ki so rojene kasneje kot i (vsaj d let kasneje). Zato bomo te dolžine verig računali od konca zaporedja proti začetku, torej od kasneje rojenih oseb proti zgodneje rojenim. Tako bomo vedno imeli pri roki vse a_j, ki jih potrebujemo pri izračunu nekega a_i.

Zapišimo dobljeni postopek s psevdokodo:

```
uredi osebe po letu rojstva in jih v tem vrstnem redu oštevilči od 1 do n;
for i := n downto 1:
  a_i := 1; n_i := −1;
  for j := i + 1 to n:
    if r_j < r_i + d then continue;
    if r_j > s_i then break;
    if a_j + 1 > a_i then a_i := a_j + 1,   n_i := j;
```

Nato moramo poiskati tisti i, pri katerem smo dosegli največjo vrednost a_i. Njegovo verigo zdaj lahko izpišemo takole:

```
while i ≠ −1:
  izpiši i;
  i := n_i;
```

Ta rešitev ima časovno zahtevnost $O(n^2)$ zaradi vgnezdenih zank po i in j. Dalo bi se jo še izboljšati, če bi nad tabelo $a = (a_1, \ldots, a_n)$ gradili drevesasto strukturo z

maksimumi po dveh, štirih, osmih itd. zaporednih elementov. Interval j-jev, ki nas pri nekem i zanima (torej tistih, ki ustrezajo pogoju $r_i + d \leq r_j \leq s_i$), lahko poiščemo z bisekcijo v času $O(\log n)$, nato pa z omenjenim drevesom maksimumov v $O(\log n)$ časa tudi poiščemo maksimum vrednosti a_j po vseh teh j. Skupaj z urejanjem na začetku postopka bomo tako imeli časovno zahtevnost le še $O(n \log n)$.

2. Suhi dnevi

Vhodne podatke bomo brali vrstico po vrstico. V spremenljivki nVseh bomo šteli vse suhe dni, v spremenljivki maxSkupina hranimo dolžino najdaljše skupine suhih dni doslej, v spremenljivki trenSkupina pa dolžino strnjene skupine suhih dni, v kateri se trenutno nahajamo (če trenutni dan ni suh, je trenSkupina = 0).

Podatek o tem, ali je trenutni dan suh ali ne, hranimo v spremenljivki suh. Pri prvi meritvi tistega dneva jo postavimo na **true** ali **false** odvisno od tega, ali je meritev enaka 0 ali ne, odtlej pa pri vsaki nadaljnji meritvi za ta dan preverimo, če je različna od 0, in če je, postavimo suh na **false**.

To, ali je dan res suh, dokončno vemo šele takrat, ko preberemo vse njegove meritve. Ko torej pridemo do konca meritev tistega dne, lahko (če je bil dan res suh) povečamo števec nVseh in dolžino trenutne skupine suhih dni trenSkupina; in če je ta skupina zdaj najdaljša doslej, si jo zapomnimo v maxSkupina. Če pa trenutni dan ni bil suh, pustimo nVseh pri miru in postavimo dolžino trenutne skupine suhih dni na 0.

Kako vemo, kdaj smo prebrali vse meritve dosedanjega dneva? To v resnici vemo šele takrat, ko naletimo na prvo meritev naslednjega dneva ali pa na konec vhodnih podatkov (EOF). To pomeni, da moramo primerjati številko dneva pri trenutni meritvi s tisto pri prejšnji meritvi; če sta različni, pomeni, da je konec meritev tistega prejšnjega dneva. Pri tem moramo paziti še na dva robna primera: na začetku vhodne datoteke sploh še nimamo prejšnje meritve (da zaznamo ta primer, si pomagamo s spremenljivko prvi), na koncu vhodne datoteke pa nimamo trenutne meritve (ta primer pokrijemo s pomočjo spremenljivke ok, ki pove, ali je branje trenutne meritve uspelo ali ne).

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

int main()
{
  int maxSkupina = 0, trenSkupina = 0, nVseh = 0, dan, prejDan, meritev;
  bool suh, ok, prvi = true;

  do
  {
    /* Poskusimo prebrati naslednjo meritev. */
    ok = (2 == scanf("%d %d", &dan, &meritev));
    if (ok && ! prvi && dan == prejDan) {
      /* Nadaljuje se isti dan kot pri prejšnji meritvi.
         Če je meritev neničelna, si zapomnimo, da dan ni suh. */
      if (meritev != 0) suh = false; }
    else
    {
      /* Prejšnjega dneva je konec. */
      if (! prvi) {
        /* Če je bil ta dan suh, povečajmo števec suhih dni in dolžino trenutne
           skupine suhih dni. */
        if (suh) nVseh++, trenSkupina++;
        /* Če dan ni bil suh, postavimo dolžino skupine suhih dni na 0. */
        else trenSkupina = 0;
        /* Če je to najdaljša skupina suhih dni doslej, si jo zapomnimo. */
        if (trenSkupina > maxSkupina) maxSkupina = trenSkupina; }
      /* Začenja se nov dan. */
      if (ok) {
        prvi = false;          /* Zdaj gotovo nismo več pred prvo meritvijo. */
        prejDan = dan;         /* Zapomnimo si številko trenutnega dneva. */
```

```
            suh = (meritev == 0);  /* Ali je dan doslej suh? */ }
        }
    }
    while (ok);

    /* Izpišimo rezultate. */
    printf("%d suhih dni, največ %d skupaj.\n", nVseh, maxSkupina); return 0;
}
```

3. Virus

(a) Če imamo en sam računalnik, lahko okuženo zgoščenko najdemo z bisekcijo v največ 10 dneh. Prvi dan razdelimo zgoščenke na dve skupini po 500 zgoščenk in poženemo vse zgoščenke iz ene skupine. Če je računalnik okužen, vemo, da je bila okužena zgoščenka v tisti skupini, sicer pa v drugi (tisti, ki je nismo poganjali). Naslednji dan torej nadaljujemo z eno od teh dveh skupin (tisto, v kateri je okužena zgoščenka); razdelimo jo na dve skupini po 250 zgoščenk in tako naprej. Tako našo skupino vsak dan razpolovimo in deseti dan nam ostaneta največ dve zgoščenki; eno od njiju poženemo in če se računalnik sesuje, je bil virus na njej, sicer pa na tisti drugi.

(b) Če imamo le en dan časa, lahko virus poiščemo z 10 računalniki. Vsaki zgoščenki pripišimo enolično številko od 1 do 1000. Vsako številko zapišimo v dvojiškem sestavu kot niz 10 bitov (na primer, 123 v desetiškem je 0001111011 v dvojiškem). Na prvem računalniku poženimo vse zgoščenke, pri katerih je v dvojiškem zapisu prižgan prvi bit; na drugem vse zgoščenke, pri katerih je prižgan drugi bit; in tako naprej. Iz tega, kateri računalniki se sesujejo, kateri pa ne, lahko takoj razberemo številko zgoščenke, na kateri je bil virus.

4. Analiza enot

Ker so enote in količine predstavljene z malimi črkami angleške abecede, lahko v nalogi nastopa le 26 različnih enot in 26 različnih količin. Pripravimo si dvodimenzionalno tabelo (v spodnjem programu je to def), v kateri bo za vsako možno količino in za vsako možno enoto pisalo, s kakšno stopnjo se ta enota pojavlja v tej količini. Na začetku postavimo vse elemente te tabele na 0, nato pa berimo definicije količin in ustrezno povečujmo ali zmanjšujmo elemente tabele. Če v definiciji količine k naletimo na enoto e, moramo stopnjo def[k][e] povečati za 1, če e nastopa v števcu, oz. zmanjšati za 1, če nastopa v imenovalcu. Vhodne podatke bomo brali znak po znak in pri tem v spremenljivkah levo in zgoraj hranili podatke o tem, ali smo levo od dvopičja ali desno od njega ter ali smo (ko smo enkrat desno od dvopičja) v števcu ulomka (levo od znaka /) ali v imenovalcu (desno od znaka /).

Nato moramo prebrati še formulo samo; to je zelo podobno kot pri branju definicij količin, le da namesto dvopičja nastopa enačaj in da se lahko znak / pojavi tudi na levi strani, ne le na desni. Med branjem formule bomo v tabeli stopnje računali stopnje vseh enot v formuli. Ker formula sama v resnici ne vsebuje enot, pač pa količine, se moramo vsakič, ko v formuli naletimo na količino k, zapeljati v zanki po vseh enotah in k tabeli stopnje prišteti (ali odšteti) stopnje teh enot v definiciji količine k iz tabele def[k][e]. Prištevali bomo stopnje pri tistih količinah, ki se pojavljajo v števcu leve strani formule ali v imenovalcu desne strani, odštevali pa pri tistih, ki se pojavljajo v imenovalcu leve ali v števcu desne strani.

Na koncu moramo le še preveriti, če so vse stopnje v tabeli stopnje enake 0; če niso, potem vemo, da se enote na levi in desni strani formule ne ujemajo.

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

int main()
{
    int def[26][26], stopnje[26], nKolicin, k, e, c;
    bool levo, zgoraj;

    /* Inicializirajmo tabelo za definicije količin. */
    for (k = 0; k < 26; k++) for (e = 0; e < 26; e++) def[k][e] = 0;
```

```
  /* Preberimo definicije količin. */
  scanf("%d\n", &nKolicin);
  while (nKolicin-- > 0)
  {
    levo = true;
    /* Brali bomo po znakih vse do konca vrstice. */
    while ((c = fgetc(stdin)) != '\n')
      if (c == ':') levo = false, zgoraj = true;
      else if (c == '/') zgoraj = false;
      else if ('a' <= c && c <= 'z') {
        /* Črka levo od dvopičja nam pove količino, ki je definirana v tej vrstici;
           to si zapomnimo v spremenljivki k. */
        if (levo) k = c - 'a';
        /* Črka desno od dvopičja pa pomeni enoto, ki ji moramo zdaj stopnjo v
           definiciji količine k povečati ali zmanjšati za 1 (odvisno od tega,
           ali smo v števcu ali v imenovalcu — to nam pove spremenljivka „zgoraj“). */
        else def[k][c - 'a'] += (zgoraj) ? 1 : -1; }
  }
  /* V tabeli stopnje bomo hranili stopnje enot v formuli.
     Za začetek postavimo vse na 0. */
  for (e = 0; e < 26; e++) stopnje[e] = 0;
  /* Zdaj preberimo še formulo, ki jo moramo preveriti.
     Brali bomo po znakih do konca vrstice (ali EOF). */
  levo = true; zgoraj = true;
  while ((c = fgetc(stdin)) != '\n' && c != EOF)
    if (c == '=') levo = false, zgoraj = true;
    else if (c == '/') zgoraj = false;
    else if ('a' <= c && c <= 'z')
      /* Črka pomeni eno od količin; poglejmo v tabelo def, iz katerih enot je ta
         količina sestavljena, in njihove stopnje prištejmo ali odštejmo k tabeli
         stopnje. Prištevajo se stopnje količin v števcu leve strani formule
         in imenovalcu desne strani, odštevajo pa se stopnje količin v imenovalcu
         leve strani in števcu desne strani formule. */
      for (e = 0; e < 26; e++) stopnje[e] += (zgoraj == levo ? 1 : -1) * def[c - 'a'][e];
  /* Preverimo, če so vse stopnje enake 0, in izpišimo rezultat. */
  for (e = 0; e < 26; e++) if (stopnje[e] != 0) break;
  printf("Formula %s pravilna.\n", (e < 26) ? "ni" : "je");
  return 0;
}
```

5. Za žužke gre

V nalogi se skriva problem barvanja grafa z dvema barvama. Recimo, da spol žužkov označimo z 1 in 2, saj ne moremo vedeti, kateri je kateri. Zdaj lahko razmišljamo takole: izberimo si poljubnega žužka in mu dodelimo spol 1; iz tega lahko takoj zaključimo, da morajo vsi, ki so bili v kakšni interakciji z njim, imeti spol 2. Podobno pa, ko kakšnemu žužku pripišemo spol 2, lahko zaključimo, da so morali biti tisti, ki so bili v interakciji z njim, spola 1. Tako nadaljujemo in prej ali slej bodisi pripišemo spol vsem žužkom, ki se jih je dalo prek teh interakcij sploh doseči, bodisi naletimo na primer, ko nekemu žužku pripišemo en spol, kasneje pa ugotovimo, da bi mu morali pripisati še nasprotni spol. V tem primeru vemo, da žužkom ni mogoče pripisati spola v skladu z zahtevami naloge.

Paziti moramo še na možnost, da z gornjim postopkom še nismo pripisali spola vsem žužkom, ker je njihova populacija mogoče razdeljena na več manjših skupin, ki druga z drugo niso imele stikov. Zato moramo gornji postopek ponavljati, dokler je še kje kak žužek, ki mu nismo pripisali spola.

Zapišimo dobljeni postopek še s psevdokodo. Spol žužka u bomo hranili v b_u ($b_u = 0$ pomeni, da mu spola še nismo dodelili). V množici Q hranimo žužke, ki smo jim že pripisali spol, nismo pa še pregledali, kaj to pomeni za spol ostalih žužkov, ki so bili v stiku z njimi.

```
for u := 1 to n do b_u := 0;
for z := 1 to n do if b_z = 0:
  (* Žužek z še nima spola. Pripišimo mu ga in ga dodajmo v Q. *)
  b_z := 1; Q := {z};
  while Q ni prazna:
    u := poljuben element množice Q; pobriši u iz Q;
    za vsakega žužka v, ki je bil kdaj v interakciji z u:
      if b_v = 0 then
        (* v-ju pripišimo spol (nasprotnega od b_u) in ga dodajmo v Q. *)
        b_v := 3 − b_u; dodaj v v Q;
      else if b_v = b_u then
        (* Žužkom ni mogoče pripisati spola v skladu z omejitvami naloge. *)
        return false;
return true;
```

V praksi lahko Q predstavimo s kakršnim koli seznamom, skladom, vrsto ali čim podobnim. Za naštevanje žužkov v, ki so bili kdaj v interakciji z žužkom u, je načeloma koristno, če si na začetku za vsakega žužka u pripravimo seznam vseh, ki so bili kdaj v interakciji z njim (recimo temu seznamu $L[u]$). Tako nam ne bo treba vsakič iti po vhodnem seznamu vseh m interakcij v terariju, ampak bomo morali iti le po tistih, ki so res bili v interakciji z u. Zapišimo še ta postopek:

```
for u := 1 to n do L[u] := prazen seznam;
for i := 1 to m:
  dodaj s_i v seznam L[t_i];
  dodaj t_i v seznam L[s_i];
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Letala

Za začetek uredimo zabojnike padajoče po teži, letala pa po nosilnosti in jih v tem vrstnem redu oštevilčimo: $m_1, \ldots m_n$ in $c_1, \ldots, c_k$.

Recimo, da si izberemo nek t in bi radi sestavili razpored zabojnikov med letala, s katerim bi vse zabojnike razvozili v t dneh. (To je seveda smiselno le, če je $k \cdot t \geq n$, drugače takoj vemo, da imamo preprosto premalo letal.) Najmočnejšemu letalu c_1 dodelimo najtežjih t zabojnikov, torej $m_1, \ldots, m_t$; drugemu najmočnejšemu letalu, c_2, dodelimo naslednjih t zabojnikov, torej $m_{t+1}, \ldots, m_{2t}$; in tako naprej. Če se pri kakšnem letalu izkaže, da kakšnega od zabojnikov, ki smo mu jih dodelili, ne more nesti, lahko takoj zaključimo, da veljavnega razporeda za izbrani t sploh ni.[1] Dovolj je, če pri vsakem letalu preverimo le najtežji zabojnik, ki smo mu ga dodelili — pri letalu c_j je to zabojnik $m_{t\cdot(j-1)+1}$. Takšno preverjanje nam torej vzame le $O(n/t)$ časa.

Najmanjši primerni t lahko zdaj poiščemo z bisekcijo. Na začetku preverimo, če je $m_1 > c_1$; če je, lahko takoj obupamo, saj je zabojnik m_1 pretežak za vsa letala. Drugače pa vemo, da bi se dalo zabojnike razvoziti v n dneh (vse s prvim letalom); po drugi strani vemo, da se jih ne da razvoziti v samo 0 dneh. Najmanjši t je torej nekje na območju $0 < t \leq n$. V nadaljevanju to območje razpolavljamo, dokler ne ostane ena sama vrednost t-ja.

[1] O tem se lahko prepričamo takole: naj bo c_j naše preobremenjeno letalo; najtežji zabojnik, ki smo mu ga dodelili, je m_i za $i = t \cdot (j-1) + 1$ in če je kakšen zabojnik za naše letalo pretežak, mora biti to ravno m_i. Recimo, da bi vendarle obstajal nek veljaven razpored za t dni; naj bo c_u letalo, ki v tem razporedu pelje zabojnik m_i. Ker je m_i pretežak za letalo c_j, je pretežak tudi za letala $c_{j+1}, \ldots, c_k$ (ki nimajo nič večje nosilnosti od c_j), torej mora biti c_u eno izmed letal $c_1, \ldots, c_{j-1}$. Ta letala lahko v t dneh razvozijo največ $t \cdot (j-1)$ zabojnik; ker je med njimi tudi zabojnik m_i (ki ga pelje letalo c_u) in ker je $i > t \cdot (j-1)$, to pomeni, da mora biti med zabojniki $m_1, \ldots, m_{t\cdot(j-1)}$ vsaj en tak, ki ga ne pelje nobeno od letal $c_1, \ldots, c_{j-1}$; recimo temu zabojniku m_v. Ker je $v > i$, je ta zabojnik vsaj toliko težak kot m_i; in ker je m_i pretežak za vsa letala od vključno c_j naprej, je zanje pretežak tudi m_v. Zabojnika m_v torej ne pelje nobeno od prvih $j-1$ letal, ostala pa ga sploh ne morejo peljati, torej tak razpored sploh ne more obstajati.

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

int Primerjaj(const void *x, const void *y) { return *(const int *) y - *(const int *) x; }

#define MaxN 100000
#define MaxK 100000
int mi[MaxN], cj[MaxK], n, k;

int main()
{
  int i, j, tL, tH, t;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("letala.in", "rt");
  fscanf(f, "%d %d\n", &n, &k);
  for (i = 0; i < n; i++) fscanf(f, "%d", &mi[i]);
  for (j = 0; j < k; j++) fscanf(f, "%d", &cj[j]);
  fclose(f);

  /* Uredimo zabojnike in letala padajoče. */
  qsort(mi, n, sizeof(mi[0]), &Primerjaj);
  qsort(cj, k, sizeof(cj[0]), &Primerjaj);

  /* Poiščimo najmanjši t z bisekcijo. */
  if (cj[0] < mi[0]) tL = -1, tH = -1; else tL = 0, tH = n;
  while (tH - tL > 1)
  {
    /* V tH dneh je mogoče razvoziti vse zabojnike, v tL dneh pa ne. */
    t = (tL + tH) / 2;

    /* Dajmo najmočnejšemu letalu najtežjih t zabojnikov, naslednjemu
       naslednjih t in tako naprej. Preverimo, če bo vsak zmogel peljati
       zabojnike, ki smo mu jih dodelili. */
    for (j = 0; j * t < n; j++)
      if (cj[j] < mi[j * t]) break;

    /* Če smo prišli do konca, ne da bi se pri kakšnem letalu izkazalo, da
       ne bo zmoglo, je mogoče tovor razvoziti v t dneh, sicer pa ne. */
    if (j * t < n) tL = t; else tH = t;
  }

  /* Izpišimo rezultat. */
  f = fopen("letala.out", "wt"); fprintf(f, "%d\n", tH); fclose(f); return 0;
}
```

Časovna zahtevnost te rešitve je $O(n \log n + k \log k)$ za urejanje obeh tabel, nato pa še $O(n \log n)$ za iskanje najmanjšega t-ja z bisekcijo. Pravzaprav bi ostali pri časovni zahtevnosti $O(n \log n)$ celo, če bi namesto bisekcije preizkušali kar vse možne t-je po vrsti od $t = n$ navzdol. Podobno bi ostali pri časovni zahtevnosti $O(n \log n)$, če bi pri posameznem t preverili vse zabojnike in ne le vsakega t-tega. Če pa naredimo obe poenostavitvi naenkrat (pregledamo vse zabojnike in ne uporabimo bisekcije), pademo v časovno zahtevnost $O(n^2)$, kar je za večje testne primere pri naši nalogi že prepočasi.

Oglejmo si zdaj še malo drugačno rešitev te naloge. Preštejmo, za koliko najtežjih zabojnikov velja, da jih lahko pelje le prvo letalo (tisto z največjo nosilnostjo, c_1), ostala letala pa ne. Recimo, da je teh zabojnikov p_1; ker jih ne bo mogoče razvoziti drugače kot tako, da prvo letalo vsak dan pelje po enega od njih, bomo za razvoz teh zabojnikov potrebovali p_1 dni, torej tudi za razvoz vsega tovora potrebujemo vsaj p_1 dni.

Nato poglejmo, koliko je takih zabojnikov, ki jih lahko peljeta najzmogljivejši dve letali, ostala pa ne; recimo, da je takih zabojnikov p_2. Za razvoz teh zabojnikov torej gotovo potrebujemo vsaj $\lceil p_2/2 \rceil$ dni, to pa je zato tudi spodnja meja za čas razvoza vsega tovora. Podobno, če se za p_3 zabojnikov izkaže, da jih lahko nesejo najzmogljivejša tri letala, ostala pa ne, lahko zaključimo, da za razvoz teh zabojnikov potrebujemo vsaj $\lceil p_3/3 \rceil$ dni, zato pa tudi za razvoz vsega tovora potrebujemo vsaj toliko dni. Tako nadaljujemo za vse več letal, vse do k, in med tako dobljenimi mejami vzamemo najvišjo:

$t = \max_{0 \le j \le k} \lceil p_j / j \rceil$.

Iz dosedanjega razmisleka že vidimo, da v manj kot t dneh ne bomo mogli razvoziti vsega tovora. V t dneh pa gre: že pri prvi rešitvi smo videli, da lahko tak razpored sestavimo tako, da prvo letalo prepelje prvih t zabojnikov, drugo naslednjih t in tako naprej. Najtežji zabojnik, ki ga dobi letalo j, je zabojnik številka $t(j-1)+1$. To je naprej $\geq \lceil p_{j-1}/(j-1) \rceil (j-1) + 1 \geq p_{j-1} + 1$. Ta zabojnik torej ni eden od najtežjih p_{j-1} zabojnikov, v prejšnjem odstavku pa smo p_{j-1} definirali tako, da letalo j lahko nese vse zabojnike razen najtežjih p_{j-1}. Torej zabojnik $t(j-1)+1$ za letalo j gotovo ni pretežak. Ker ta razmislek velja za vsako letalo, lahko zaključimo, da je naš razpored veljaven in je torej vse zabojnike res mogoče prepeljati v t dneh.

Lepo pri tej rešitvi je, da potrebujemo zanjo le en prehod po urejenem seznamu zabojnikov in hkrati še po urejenem seznamu letal. Časovna zahtevnost tega dela postopka je torej le $O(n+k)$, pred tem pa še vedno porabimo $O(n \log n + k \log k)$ časa za urejanje obeh seznamov. Zapišimo to rešitev še v C-ju:

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

#define MaxN 100000
#define MaxK 100000

int Primerjaj(const void *x, const void *y) { return *(const int *) y - *(const int *) x; }

int mi[MaxN], cj[MaxK], n, k;

int main()
{
  int i, j, t, kand;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("letala.in", "rt");
  fscanf(f, "%d %d\n", &n, &k);
  for (i = 0; i < n; i++) fscanf(f, "%d", &mi[i]);
  for (j = 0; j < k; j++) fscanf(f, "%d", &cj[j]);
  fclose(f);

  /* Uredimo zabojnike in letala padajoče. */
  qsort(mi, n, sizeof(mi[0]), &Primerjaj);
  qsort(cj, k, sizeof(cj[0]), &Primerjaj);

  /* Poiščimo najmanjši t z bisekcijo. */
  if (mi[0] > cj[0]) t = -1;
  else for (i = 0, j = 1, t = 0; j <= k; j++)
  {
    /* Za koliko zabojnikov velja, da jih lahko peljejo letala od 0 do j - 1,
       letala od j do k - 1 pa ne? */
    if (j == k) i = n;
    else while (i < n && mi[i] > cj[j]) i++;

    /* Teh i zabojnikov lahko torej razvozi le najmočnejših j letal,
       za kar bo potrebnih vsaj i/j dni. Ker mora biti število dni celo,
       bomo količnik i/j zaokrožili navzgor. */
    kand = (i + j - 1) / j;
    if (kand > t) t = kand;
  }

  /* Izpišimo rezultat. */
  f = fopen("letala.out", "wt"); fprintf(f, "%d\n", t); fclose(f); return 0;
}
```

2. Dominosa

Domine lahko polagamo z rekurzijo. Na vsakem koraku poiščemo najbolj zgornje še nepokrito polje, če je takih več, pa vzamemo najbolj levo od njih. Ker sta njegov zgornji in levi sosed (če ju sploh ima) že pokrita, lahko to polje pokrijemo le tako, da ga povežemo z desnim ali pa s spodnjim sosedom. Za vsako od teh dveh možnosti preverimo, če take domine še nismo uporabili, in z rekurzivnim klicem poskusimo pokriti preostanek mreže.

Spodnji program uporablja globalno spremenljivko uporabljena za označevanje tega, katere domine je že uporabil. Ko položimo novo domino v mrežo, postavimo ustrezno vrednost v tej tabeli na **true**, po vrnitvi iz rekurzivnega klica pa nazaj na **false**. V tabeli pokrito pa označujemo, katera polja so že pokrita in katera ne; pri že pokritih si v tej tabeli tudi zapomnimo, kateri od njegovih sosedov pripada isti domini kot to polje — ta podatek bo prišel prav na koncu, ko bo treba mrežo domin narisati.

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

#define MaxPik 9
#define MaxW (MaxPik + 2)
#define MaxH (MaxPik + 1)

int t[MaxH][MaxW]; /* vhodna mreža */
typedef enum { Ne, N, W, E, S } PokritoT;
PokritoT pokrito[MaxH][MaxW];
int w, h, maxPik;
/* Veljavni elementi so uporabljena[p1][p2], 0 ≤ p1 ≤ p2 ≤ maxPik. */
bool uporabljena[MaxPik + 1][MaxPik + 1];

bool Rekurzija(int x, int y)
{
  int i, p1, p2, tmp;

  /* Poiščimo naslednje nepokrito polje. */
  p1 = x; p2 = y;
  while (y < h)
  {
    while (x < w && pokrito[y][x] != Ne) x++;
    if (x < w) break;
    x = 0; y++;
  }
  if (y >= h) return true; /* Vse je pokrito, našli smo resitev. */

  /* Poskusimo postaviti vodoravno domino. */
  if (x + 1 < w && pokrito[y][x + 1] == Ne) {
    p1 = t[y][x]; p2 = t[y][x + 1];
    if (p2 < p1) tmp = p1, p1 = p2, p2 = tmp;
    if (! uporabljena[p1][p2])
    {
      uporabljena[p1][p2] = true; pokrito[y][x] = E; pokrito[y][x + 1] = W;
      if (Rekurzija(x + 2, y)) return true;
      uporabljena[p1][p2] = false; pokrito[y][x] = Ne; pokrito[y][x + 1] = Ne;
    }
  }

  /* Poskusimo postaviti navpično domino. */
  if (y + 1 < h && pokrito[y + 1][x] == Ne)
  {
    p1 = t[y][x]; p2 = t[y + 1][x];
    if (p2 < p1) tmp = p1, p1 = p2, p2 = tmp;
    if (! uporabljena[p1][p2])
    {
      uporabljena[p1][p2] = true; pokrito[y][x] = S; pokrito[y + 1][x] = N;
      if (Rekurzija(x + 1, y)) return true;
      uporabljena[p1][p2] = false; pokrito[y][x] = Ne; pokrito[y + 1][x] = Ne;
    }
  }
  return false;
}

int main()
{
  int y, x;

  /* Preberimo vhodne podatke. */
```

```
  FILE *f = fopen("dominosa.in", "rt");
  fscanf(f, "%d", &maxPik);
  h = maxPik + 1; w = h + 1;
  for (y = 0; y < h; y++) for (x = 0; x < w; x++) {
    fscanf(f, "%d", &t[y][x]); pokrito[y][x] = Ne; }
  fclose(f);

  for (x = 0; x <= maxPik; x++) for (y = 0; y <= maxPik; y++) uporabljena[y][x] = false;
  Rekurzija(0, 0);

  /* Izpišimo rezultate. */
  f = fopen("dominosa.out", "wt");
  for (y = 0; y < h; y++) {
    for (x = 0; x < w; x++) fprintf(f, "%d%s", t[y][x],
                    (x + 1 < w && pokrito[y][x] == E) ? "-" : (x == w - 1 ? "\n" : "."));
    if (y < h - 1) for (x = 0; x < w; x++) fprintf(f, "%s%s",
              (y + 1 < h && pokrito[y][x] == S) ? "|" : ".", (x == w - 1) ? "\n" : "."); }
  fclose(f); return 0;
}
```

3. Galaktična zavezništva

Oglejmo si najprej preprosto in malo manj učinkovito rešitev, ki je dovolj dobra za manjše testne primere (recimo do $n \approx 2000$; na našem tekmovanju bi dovolj hitro rešila polovico testnih primerov). Ker so naši nizi dolgi največ 50 bitov, lahko posamezni niz predstavimo kar kot 64-bitno celoštevilsko spremenljivko (npr. tip **long long** v C/C++, long v javi ipd.). S tremi gnezdenimi zankami lahko pregledamo vse možne trojice planetov, pri vsaki izračunamo xor njihovih števil in si zapomnimo največjo med njimi.

```
#include <stdio.h>

#define MaxM 50
#define MaxN 7500

int main()
{
  int n, m, i, j, tren, b, a1, a2, a3;
  long long kand, kand2, naj, nizi[MaxN];
  char buf[MaxM + 3];

  /* Preberimo n in m. */
  FILE *f = fopen("xor.in", "rt");
  fscanf(f, "%d %d\n", &n, &m);

  /* Preberimo nize. */
  for (i = 0; i < n; i++) {
    fgets(buf, sizeof(buf), f);
    nizi[i] = 0;
    for (j = 0; j < m; j++) if (buf[j] == '1')
      nizi[i] |= ((long long) 1) << (m - 1 - j); }
  fclose(f);

  /* Preglejmo vse trojice. */
  for (naj = 0, a1 = 0; a1 < n - 2; a1++) for (a2 = a1 + 1; a2 < n - 1; a2++)
    for (a3 = a2 + 1, kand2 = nizi[a1] ^ nizi[a2]; a3 < n; a3++) {
      kand = kand2 ^ nizi[a3];
      if (kand > naj) naj = kand; }

  /* Izpišimo rezultate. */
  f = fopen("xor.out", "wt");
  for (j = 0; j < m; j++) fputc('0' + (int) ((naj >> (m - 1 - j)) & 1), f);
  fclose(f); return 0;
}
```

Težava pri tem postopku je, da je vseh trojic $\binom{n}{3} = n(n-1)(n-2)/6$, tako da ima ta postopek časovno zahtevnost $O(n^3)$. (V splošnem pravzaprav $O(n^3 m)$, ker za xor dveh m-bitnih nizov potrebujemo $O(n)$ časa, če naši nizi niso tako kratki, da gredo v eno samo celoštevilsko spremenljivko.) Pri večjih n zato ta postopek postane prepočasen.

Razmislimo za začetek o malo poenostavljeni različici naloge: recimo, da namesto zavezništev treh planetov gledamo zavezništva dveh. Iščemo torej dva niza a in b tako, da bo $c = a \operatorname{xor} b$ čim večji. Mogoče se nekateri vhodni nizi začnejo na ničlo, nekateri pa na enico. Če tedaj izberemo niza a in b tako, da se bo eden začel na ničlo, drugi pa na enico, se bo c začel na enico; če pa izberemo a in b tako, da se bosta oba začela na ničlo ali pa oba na enico, se bo c začel na ničlo. Vsak c, ki se začne na enico, je večji (zavezništvo je močnejše) od vsakega c, ki se začne na ničlo.

Podobno razmišljamo tudi pri nižjih bitih: vedno poskušamo niza a in b nadaljevati tako, da se v istoležnem bitu razlikujeta, tako da bo imel c na tistem mestu enico. Šele če to ne gre (ker primernih vhodnih nizov sploh ni), poskusimo tudi možnost, da oba (a in b) nadaljujemo z ničlo ali pa oba z enico.

Ta ideja nas pripelje do naslednjega rekurzivnega postopka. Spodnja funkcija vrne največji $a' \operatorname{xor} b'$, ki ga je mogoče dobiti, če za a' vzamemo kakšen tak vhodni niz, ki se začne na a, in za b' vzamemo kakšen tak vhodni niz, ki se začne na b. Če primernih vhodnih nizov sploh ni, funkcija vrne -1.

funkcija REK(globina g, niza $a = a_1 a_2 \ldots a_g$ in $b = b_1 b_2 \ldots b_g$):
  if se noben vhodni niz ne začne na a ali pa noben na b **then return** -1;
  if $g = m$ **then return** $a \operatorname{xor} b$;
  $c := \max\{\text{REK}(g+1, a\texttt{0}, b\texttt{1}), \text{REK}(g+1, a\texttt{1}, b\texttt{0})\}$;
  if $c = -1$ **then**
    $c := \max\{\text{REK}(g+1, a\texttt{0}, b\texttt{0}), \text{REK}(g+1, a\texttt{1}, b\texttt{1})\}$;
  return c;

Postopek poženemo z globino $g = 0$ in praznima nizoma a in b.

Da bomo lahko učinkovito preverjali, če obstaja kak vhodni niz, ki se začne na $a_1 \ldots a_g$ (ali $b_1 \ldots b_g$), je koristno zložiti vhodne nize v drevo (*trie*). Vsako vozlišče drevesa ima načeloma dva otroka z oznakama `0` in `1` (nista pa nujno oba prisotna); za vsak vhodni niz obstaja v drevesu veja (pot od korena do nekega lista), na kateri so oznake vozlišč ravno biti tega niza.

Naš rekurzivni postopek lahko zdaj namesto niza $a_1 a_2 \ldots a_g$ uporabi vozlišče drevesa, do katerega pridemo, če začnemo v korenu in po vrsti sledimo povezavam $a_1, a_2, \ldots, a_g$. To, da se nizu a na koncu pritakne še en bit, zdaj pomeni, da se moramo iz tistega vozlišča premakniti v ustreznega otroka. Če otroka, ki ga potrebujemo, ni, pa vemo, da med vhodnimi nizi ni nobenega takega, ki bi se začel na naš novi niz a.

funkcija REK(globina g, vozlišči a in b):
1 **if** $a =$ NIL **or** $b =$ NIL **then return** -1;
2 **if** $g = m$ **then return** $a \operatorname{xor} b$;
3 $c := \max\{\text{REK}(g+1, a.otrok[\texttt{0}], b.otrok[\texttt{1}]), \text{REK}(g+1, a.otrok[\texttt{1}], b.otrok[\texttt{0}])\}$;
4 **if** $c = -1$ **then**
5    $c := \max\{\text{REK}(g+1, a.otrok[\texttt{0}], b.otrok[\texttt{0}]), \text{REK}(g+1, a.otrok[\texttt{1}], b.otrok[\texttt{1}])\}$;
6 **return** c;

Kakšna je časovna zahtevnost tega postopka? Pri rekurzivnih klicih v vrstici 3 je tako, da če kakšen od otrok manjka, bo tisti klic to takoj ugotovil (v vrstici 1) in se vrnil (takemu klicu recimo, da je trivialen); če pa sta oba otroka prisotna, bo ta klic gotovo našel neko rešitev in torej vrnil nek veljaven niz c, ne pa -1; in v tem primeru se vrstica 5 ne bo izvedla. To pa pomeni, da se na primer z otrokom $a.otrok[\texttt{0}]$ izvede samo en netrivialen rekurziven klic: če je to tisti v vrstici 3, se vrstica 5 sploh ne bo izvedla; če pa je tisti v vrstici 3 trivialen, se bo izvedel tisti v vrstici 5. Podobno je tudi z drugimi otroki. Iz tega vidimo, da se lahko v celotnem času izvajanja našega postopka vsako vozlišče drevesa pojavi kot parameter a pri največ enem rekurzivnem klicu, v paru z največ enim drugim vozliščem b. Vseh rekurzivnih klicev skupaj je torej le toliko, kolikor je vozlišč v drevesu, to pa je $O(n \cdot m)$. (Vsi vhodni nizi skupaj so dolgi $n \cdot m$ znakov, drevo pa ima lahko manj kot toliko vozlišč, če se več nizov ujema v prvih nekaj bitih.) Koliko dela pa imamo s posameznim klicem (če odmislimo čas, porabljen v morebitnih vgnezdenih klicih)? V vrstici 2 moramo računati xor dveh m-bitnih nizov, v vrsticah 3 in 5 pa max dveh m-bitnih nizov; vse to so načeloma operacije, ki porabijo $O(m)$ časa, vendar pa se lahko z nekaj pazljivosti temu faktorju $O(m)$ izognemo. Niz $a \operatorname{xor} b$ lahko

računamo sproti, bit po bit, že med samimi rekurzivnimi klici, in ga hranimo v neki globalni spremenljivki, tako da ko bi morala vrstica 2 računati $a \operatorname{xor} b$, je ta vrednost v resnici že izračunana. S pomočjo take globalne spremenljivke lahko funkcija REK tudi sproti preverja, ali ima vrednost $a \operatorname{xor} b$, ki nam trenutno nastaja, sploh kakšne možnosti, da bi bila boljša od najboljše doslej znane. Tako nam v vrsticah 3 in 5 ne bo treba računati max, ker bodo rekurzivni klici sami pravočasno obupali in vrnili -1, če ne bodo mogli vrniti nove najboljše rešitve. (Podrobnosti tega si bomo ogledali v izvorni kodi rešitve malo kasneje.) Tako torej vidimo, da ima naš postopek zahtevnost le $O(nm)$, če ga pazljivo implementiramo.

Doslej smo razmišljali o iskanju največjega xor-a dveh nizov, naloga pa zahteva največji xor treh nizov. Ta problem lahko rešimo z eno dodatno zanko: prvi niz fiksirajmo, nato pa s postopkom, podobnim zgornjemu, poiščimo še dva taka niza, ki bosta skupaj s prvim dala čim večji xor.

$naj := -1$;
za vsakega od n vhodnih nizov, recimo mu p:
  pobriši p iz drevesa;
  $c :=$ največja možna vrednost $p \operatorname{xor} a \operatorname{xor} b$ po vseh preostalih nizih a, b; (†)
  if $c > naj$ **then** $naj := c$;
  dodaj p nazaj v drevo;

Brisanje iz drevesa in dodajanje vanj vzameta vsakič po $O(m)$ časa, prav tako primerjava rešitve c z najboljšo rešitvijo doslej. Vrstico (†) pa lahko izvedemo z rekurzivnim postopkom, zelo podobnim tistemu, ki smo ga razvili malo prej za iskanje največje vrednosti $a \operatorname{xor} b$. Zdaj nas namesto te zanima $p \operatorname{xor} a \operatorname{xor} b$. Razlika je predvsem v tem, da če ima p na nekem indeksu enico, si zdaj od a-ja in b-ja želimo, da bi imela tam oba enako vrednost (oba ničlo ali oba enico), ne pa eden ničlo in eden enico; takrat moramo torej vrstici 3 in 5 funkcije REK ravno zamenjati. Še ena razlika pa je naslednja: pri dosedanji funkciji REK se je lahko zgodilo, da kažeta a in b na eno in isto vozlišče drevesa, ki predstavlja en sam vhodni niz; to načeloma ni dopustna rešitev, saj ne predstavlja zavezništva dveh planetov. Vendar je bil v takem primeru $a \operatorname{xor} b = a \operatorname{xor} a = 0$, mi pa smo iskali maksimalni xor, tako da to ni moglo vplivati na naše rezultate. Zdaj, pri zavezništvih treh planetov, pa moramo biti bolj pazljivi. Če je v p veliko bitov prižganih, a in b pa predstavljata en in isti vhodni niz, bo $p \operatorname{xor} a \operatorname{xor} b = p$ lahko precej velika vrednost, mogoče celo večja od katerega koli xora treh različnih nizov. To bi lahko povzročilo, da bi naš postopek neupravičeno vrnil rezultat, ki ga v resnici ni mogoče doseči z nobenim zavezništvom treh planetov, ampak ga lahko doseže le nek planet sam zase. Da se tej težavi izognemo, mora funkcija REK, preden v vrstici 2 vrne rezultat, preveriti, če a in b kažeta na isto vozlišče; če da in če do tega vozlišča pripelje en sam vhodni niz, potem rezultat $a \operatorname{xor} b$ ne bi bil veljaven in moramo namesto njega vrniti -1. Če pa imamo v vhodnih podatkih res dva popolnoma enaka niza, je rešitev z $a = b$ lahko čisto sprejemljiva.

Skupaj je torej časovna zahtevnost naše rešitve za zavezništva treh planetov $O(n^2 m)$. Zapišimo dobljeni postopek še v C-ju:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MaxM 50
#define MaxN 7500

typedef struct { int otrok[2], listov; } Vozlisce;
Vozlisce *vozl;
int koren, nVozlisc, n, m, prvi, naj[MaxM];
char nizi[MaxN][MaxM + 3];

bool Rekurzija(int u, int v, int g, bool jeNaj)
{
  /* Na tem mestu velja: u in v sta vozlišči v drevesu na globini g (koren je na globini 0,
     listi na globini m). Xor nizov, ki ju predstavljata ti dve vozlišči, in prvih g bitov niza
     „prvi“, je shranjen v prvih g elementih tabele „naj“ in je enak (če je jeNaj = false)
```

```
    oz. večji (če je jeNaj = true) od prvih g bitov najmočnejšega doslej znanega zavezništva.
    Če je jeNaj = true, to pomeni, da moramo z rekurzijo nadaljevati, tudi če bomo v
    preostalih bitih dobili manjše vrednosti od dosedanje „naj“, saj bo trenutno
    zavezništvo že zaradi prvih g bitov boljše od najboljšega doslej znanega.
    Funkcija Rekurzija vrne true, če uspe najti novo najboljšo rešitev doslej. */
  int pb, prejNajBit; bool kajNasli, prejJeNaj;
  Vozlisce *U = vozl + u, *V = vozl + v;

  /* Če teh vozlišč sploh ni (ali pa sta prazni, ker smo tam iz
     drevesa začasno zbrisali niz „prva“), se takoj vrnimo. */
  if (u < 0 || v < 0) return false;
  if (u == v) { if (U->listov < 2) return false; }
  else if (U->listov == 0 || V->listov == 0) return false;
  /* Če smo prišli do listov, se ta veja rekurzije konča. */
  if (g == m) return jeNaj;

  pb = nizi[prvi][g]; /* Trenutni bit planeta „prvi“. */

  /* Poskusimo najprej nadaljevati pot tako, da bo imel xor vseh
     treh planetov na mestu g prižgan bit. */
  prejJeNaj = jeNaj; prejNajBit = naj[g];
  /* Če je imelo najboljše dosedanje zavezništvo tu ugasnjen bit, bo naše nastajajoče
     zavezništvo (ki bo imelo tu prižgan bit) gotovo boljše od njega. */
  if (naj[g] == 0) jeNaj = true;
  naj[g] = 1; /* Naše zavezništvo bo imelo tu prižgan bit. */
  /* Izvedimo zdaj oba rekurzivna klica. */
  kajNasli = false; /* Ali smo v gnezdenih klicih našli kakšno rešitev? */
  if (Rekurzija(U->otrok[0], V->otrok[pb ^ 1], g + 1, jeNaj)) kajNasli = true, jeNaj = false;
  if (u != v || pb == 1)
    if (Rekurzija(U->otrok[1], V->otrok[pb], g + 1, jeNaj)) kajNasli = true;
  /* Če smo našli kakšno rešitev, se lahko takoj vrnemo in nam ni treba preizkušati
     še tistih, ki bi imele na mestu g ničlo namesto enice. */
  if (kajNasli) return true;
  /* Vrnimo spremenljivki jeNaj in naj[g] nazaj v prvotno stanje. */
  jeNaj = prejJeNaj; naj[g] = prejNajBit;

  /* Zdaj poskusimo nadaljevati tako, da bo imel xor vseh treh planetov
     na mestu g ugasnjen bit. */
  /* Takšno nadaljevanje pa nima smisla, če že poznamo neko drugo rešitev,
     ki se na višjih bitih ujema z našo, na trenutnem mestu pa ima prižgan bit. */
  if (! jeNaj && naj[g] == 1) return false;
  naj[g] = 0; /* Naše zavezništvo bo imelo tu ugasnjen bit. */
  /* Izvedimo zdaj oba rekurzivna klica. */
  if (Rekurzija(U->otrok[0], V->otrok[pb], g + 1, jeNaj)) kajNasli = true, jeNaj = false;
  if (u != v || pb == 0)
    if (Rekurzija(U->otrok[1], V->otrok[pb ^ 1], g + 1, jeNaj)) kajNasli = true;
  if (kajNasli) return true;
  /* Če nismo našli nove najboljše rešitve, povrnimo vrednost naj[g]
     v prvotno stanje. */
  naj[g] = prejNajBit; return false;
}

int main()
{
  int i, j, tren, b;

  /* Preberimo n in m. */
  FILE *f = fopen("xor.in", "rt");
  fscanf(f, "%d %d\n", &n, &m);

  /* Pripravimo tabelo za vozlišča drevesa in ustvarimo koren. */
  vozl = (Vozlisce *) malloc(sizeof(Vozlisce) * m * n);
  koren = 0; nVozlisc = 1;
  vozl[koren].otrok[0] = -1; vozl[koren].otrok[1] = -1; vozl[koren].listov = n;

  /* Preberimo vseh n nizov in jih dodajmo v drevo. */
  for (i = 0; i < n; i++) {
```

```
    fgets(nizi[i], sizeof(nizi[i]), f);
    tren = koren;
    for (j = 0; j < m; j++) {
      b = nizi[i][j] -= '0';
      /* Če še ni otroka, v katerem moramo nadaljevati pot po drevesu, ga zdaj ustvarimo. */
      if (vozl[tren].otrok[b] < 0) {
        vozl[nVozlisc].otrok[0] = -1; vozl[nVozlisc].otrok[1] = -1;
        vozl[nVozlisc].listov = 0; vozl[tren].otrok[b] = nVozlisc++; }
      /* Premaknimo se v otroka, ki ustreza trenutnemu bitu našega niza. */
      tren = vozl[tren].otrok[b]; vozl[tren].listov++; }}
  fclose(f);
  /* Za vsak možen izbor prvega planeta izberimo ostala dva tako,
     da bosta dala skupaj s prvim čim večji xor vseh treh nizov. */
  for (j = 0; j < m; j++) naj[j] = 0;
  for (prvi = 0; prvi < n; prvi++)
  {
    /* Pobrišimo planet „prvi“ iz drevesa. */
    for (j = 0, tren = koren; j < m; j++) {
      tren = vozl[tren].otrok[nizi[prvi][j]]; vozl[tren].listov--; }
    /* Z rekurzijo poiščimo najboljše zavezništvo tega planeta s še dvema drugima. */
    Rekurzija(koren, koren, 0, false);
    /* Dodajmo planet „prvi“ nazaj v drevo. */
    for (j = 0, tren = koren; j < m; j++) {
      tren = vozl[tren].otrok[nizi[prvi][j]]; vozl[tren].listov++; }
  }
  /* Izpišimo rezultate. */
  f = fopen("xor.out", "wt");
  for (j = 0; j < m; j++) fputc('0' + naj[j], f);
  fclose(f); free(vozl); return 0;
}
```

Drevo lahko uporabimo tudi še drugače. Z dvema gnezdenima zankama si na vse možne načine izberimo prva dva planeta, recimo p in a; potem pa se vprašajmo: kateri b moramo vzeti, da bo skupaj s tema p in a dal najmočnejše zavezništvo $p \operatorname{xor} a \operatorname{xor} b$? Takega b-ja z drevesom ni težko poiskati: začnemo v korenu drevesa, nato pa se v vsakem koraku spustimo v tistega otroka, pri katerem se bo trenutni bit b-ja razlikoval od istoležnega bita v $p \operatorname{xor} a$ (tako da bo ta bit v $p \operatorname{xor} a \operatorname{xor} b$ prižgan). Šele če takega otroka ni, uporabimo tistega drugega, pri katerem se bo trenutni bit b-ja ujemal z istoležnim bitom v $p \operatorname{xor} a$. Tudi ta postopek ima časovno zahtevnost $O(n^2 m)$, le za nek konstantni faktor je slabši od prejšnjega.

4. Asteroidi

Preprosta, vendar neučinkovita rešitev je, da predstavimo igralno površino z dvodimenzionalno tabelo velikosti $w \times h$. V njej označimo, katere celice (kvadratki) so proste, katere pa zasedajo asteroidi (to nam vzame $O(wh)$ časa, ker se asteroidi ne prekrivajo in ne štrlijo iz mreže, tako da je vsota njihovih ploščin $\leq w \times h$). V tej tabeli nato z iskanjem v širino poiščemo najkrajšo pot od začetnega položaja $(0, y_0)$ do desnega roba, torej do poljubne celice oblike $(w-1, y)$. Ta postopek torej porabi $O(wh)$ časa in tudi $O(wh)$ pomnilnika, tako da je primeren le za manjše testne primere (na našem tekmovanju bi z njim rešili polovico testnih primerov).

Porabo pomnilnika lahko sicer malo zmanjšamo, če ne predstavimo cele mreže naenkrat; ker se lahko naše vesoljsko plovilo premika le v desno, ne pa v levo, to pomeni, da ko iščemo najkrajše poti do celic v stolpcu x, se nam ni treba ukvarjati s tem, kaj se dogaja v mreži desno od njega (torej v stolpcih od $x+1$ naprej), pa tudi ne s tem, kaj se dogaja levo od stolpca $x-1$, saj plovilo ne more kar skočiti za več kot eno enoto v desno. Ni nam torej treba imeti v pomnilniku cele tabele velikosti $w \times h$, ampak le po dva stolpca naenkrat. Poraba pomnilnika se tako zmanjša na $O(h)$. Pri vsakem stolpcu moramo iti po vseh asteroidih, da vidimo, kateri so prisotni v tem stolpcu; časovna

zahtevnost je tako $O(w(h+n))$, kar je pravzaprav še vedno $O(wh)$, saj je pri večjih testnih primerih n veliko manjši od h. Za takšne primere je ta postopek še vedno veliko prepočasen.

Primerjajmo v mislih dva sosednja stolpca, recimo x in $x+1$. Razlikujeta se lahko le v primeru, če se kakšen asteroid konča v stolpcu x in/ali če se kakšen asteroid začne v stolpcu $x+1$. Vsak asteroid torej lahko povzroči takšno spremembo pri največ dveh x-ih (na svojem levem robu in na svojem desnem robu). Sprememb je torej največ $2n$, vseh stolpcev pa je w. Pri naši nalogi gre lahko w do 10^6, število asteroidov n pa je največ 300. Neizogibno je torej, da nastopijo v naši mreži velike skupine zaporednih enakih stolpcev. Če sta dva sosednja stolpca enaka, to pomeni, da lahko v obeh opravimo popolnoma enake premike v smeri gor in dol; vse, kar lahko naredimo v enem, lahko naredimo tudi v drugem in obratno. Zato se lahko dogovorimo, da bomo v taki skupini več zaporednih enakih stolpcev izvedli premike gor ali dol le v zadnjem od njih, v prejšnjih stolpcih pa se bomo premikali samo v desno.

Našo prejšnjo rešitev lahko torej izboljšamo tako, da ne gledamo vseh stolpcev, ampak le tiste, pri katerih nastopi kakšna sprememba. Namesto časovne zahtevnosti $O(wh)$ imamo zdaj le še $O(nh)$.

Podoben razmislek pride prav tudi pri vrsticah. Recimo, da naše plovilo nekje na svoji poti naredi nekaj premikov desno v vrstici y, nato pa se premakne navzdol v vrstico $y+1$. Če se v vrstici $y+1$ ne začne noben asteroid (torej če to ni najvišja vrstica kakšnega asteroida), so vsa polja, ki so bila prosta v y, prosta tudi v $y+1$, torej bi lahko iz vrstice y takoj (pred premiki desno) naredili še premik dol in se nato premikali v desno po vrsici $y+1$ namesto po vrstici y. Podoben razmislek lahko naredimo tudi, če premikom desno sledi premik navzgor (in če se v vrstici $y-1$ ne konča noben asteroid). Tako torej vidimo, da se smemo omejiti na take poti, pri kateri vsi premiki desno potekajo po takih vrsticah, ki ležijo tik nad ali pa tik pod kakšnim asteroidom. (Poseben primer je le še možnost, da plovilo celotno pot opravi kar v svoji začetni vrstici y_0, če mu pri tem ni v napoto nobeden od asteroidov.)

Zdaj smo torej od vseh h vrstic obdržali največ $2n+1$ vrstic (vrstico y_0 in tiste tik pod in tik nad asteroidi). Da bomo lažje preverjali, kje nas kakšen asteroid ovira pri navpičnih premikih, je koristno od vsakega asteroida obdržati tudi eno notranjo vrstico (torej tako, v kateri ta asteroid vsaj delno leži). Tako nam ostane največ $3n+1$ vrstic — lahko jih je manj, če se več asteroidov začne ali konča v isti vrstici ipd.; spodnji program ima za to spremenljivko nys. Vse ostale vrstice pa lahko ignoriramo. Spodnji program koordinate vrstic kar preštevilči iz prvotnega območja $0, \ldots, h-1$ v $0, \ldots, \mathsf{nys}-1$. Prvotne koordinate vrstic moramo uporabljati le takrat, ko računamo, koliko navpičnih premikov je treba za premik iz ene vrstice v drugo.

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

int Primerjaj(const void *a, const void *b) {
  return *(const int *) a - *(const int *) b; }

#define MaxN 300

int w, h, n, y0, X1[MaxN], X2[MaxN], Y1[MaxN], Y2[MaxN];
int xs[2 * MaxN + 2], ys[3 * MaxN + 1], nxs, nys;
long long naj[3 * MaxN + 1]; bool zasedeno[3 * MaxN + 1];

int main()
{
  int i, ix, iy; long long kand;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("asteroidi.in", "rt");
  fscanf(f, "%d %d %d %d", &h, &w, &y0, &n);
  for (i = 0, nxs = 0; i < n; i++)
  {
    fscanf(f, "%d %d %d %d", &X1[i], &Y1[i], &X2[i], &Y2[i]);
    if (X1[i] > 0) xs[nxs++] = X1[i] - 1;
    xs[nxs++] = X2[i]; ys[nys++] = Y1[i] - 1; ys[nys++] = Y1[i]; ys[nys++] = Y2[i] + 1;
  }
```

```
  fclose(f);
  xs[nxs++] = 0; xs[nxs++] = w - 1; ys[nys++] = y0;
  /* Uredimo seznama zanimivih vrstic in stolpcev. */
  qsort(xs, nxs, sizeof(xs[0]), &Primerjaj);
  qsort(ys, nys, sizeof(ys[0]), &Primerjaj);
  /* Iz seznama zanimivih vrstic pobrišimo morebitne duplikate. */
  for (iy = 0, i = 1; i < nys; i++) if (iy == 0 || ys[iy - 1] != ys[i]) ys[iy++] = ys[i];
  nys = iy;
  /* Preštevilčimo y-koordinate asteroidov z območja 0...h-1 na 0...nys-1. */
  for (i = 0; i < n; i++) {
    iy = 0; while (ys[iy] < Y1[i]) iy++;
    Y1[i] = iy; while (ys[iy] <= Y2[i]) iy++;
    Y2[i] = iy; }
  /* Asteroid i zdaj pokriva vrstice od vključno ys[y1] do vključno ys[y2] - 1. */
  /* Na levem robu je najprej dosegljivo le polje v vrstici y0. */
  for (iy = 0; iy < nys; iy++) naj[iy] = (ys[iy] == y0) ? 0 : -1;
  /* Preglejmo zanimive stolpce od leve proti desni. */
  for (ix = 0; ix < nxs; ix++)
  {
    if (ix > 0 && xs[ix - 1] == xs[ix]) continue; /* Preskočimo duplikate v xs[]. */
    /* Poglejmo, katera polja v tem stolpcu zasedajo asteroidi.
       Tam premik desno iz xs[ix - 1] v xs[ix] ni mogoč. */
    for (iy = 0; iy < nys; iy++) zasedeno[iy] = (ys[iy] < 0 || ys[iy] >= h);
    for (i = 0; i < n; i++) if (X1[i] <= xs[ix] && xs[ix] <= X2[i])
      for (iy = Y1[i]; iy < Y2[i]; iy++) zasedeno[iy] = true, naj[iy] = -1;
    /* Upoštevajmo premike dol. */
    for (iy = 1; iy < nys; iy++) if (naj[iy - 1] >= 0 && ! zasedeno[iy]) {
      kand = naj[iy - 1] + ys[iy] - ys[iy - 1];
      if (naj[iy] < 0 || kand < naj[iy]) naj[iy] = kand; }
    /* Upoštevajmo premike gor. */
    for (iy = nys - 2; iy >= 0; iy--) if (naj[iy + 1] >= 0 && ! zasedeno[iy]) {
      kand = naj[iy + 1] + ys[iy + 1] - ys[iy];
      if (naj[iy] < 0 || kand < naj[iy]) naj[iy] = kand; }
  }
  /* Izpišimo najboljši rezultat. */
  for (kand = -1, iy = 0; iy < nys; iy++)
    if (naj[iy] >= 0) if (kand < 0 || naj[iy] < kand) kand = naj[iy];
  f = fopen("asteroidi.out", "wt"); fprintf(f, "%lld\n", kand); fclose(f); return 0;
}
```

5. Brisanje niza

Nalogo lahko rešujemo z dinamičnim programiranjem. Označimo i-ti znak našega niza s s_i, torej imamo niz $s = s_1 s_2 \ldots s_n$. Naloga sprašuje po najmanjšem številu brisanj, s katerim pobrišemo celoten niz, koristno pa jo je malo posplošiti in računati potrebno število brisanj tudi za s-jeve podnize. Naj bo torej $f(i, j)$ najmanjše število brisanj, s katerim je mogoče popolnoma pobrisati niz $s_i s_{i+1} \ldots s_j$. Pri izračunu $f(i, j)$ lahko razmišljamo takole: eno od teh brisanj bo moralo pobrisati tudi znak s_i. Mogoče ne bo pobrisalo nobenega drugega; tedaj bodo morala ostala brisanja v celoti pobrisati niz $s_{i+1} \ldots s_j$, za to pa je potrebnih vsaj $f(i + 1, j)$ brisanj. Skupaj z brisanjem znaka s_i imamo torej $1 + f(i + 1, j)$ brisanj, kar je eden od kandidatov za $f(i, j)$.

Druga možnost pa je, da tisto brisanje, ki pobriše s_i, pobriše tudi še kakšen kasnejši znak (ki mora biti seveda enak znaku s_i). Naj bo s_k (za nek k z območja $i < k \leq j$) prvi naslednji znak, ki ga pobriše isto brisanje kot znak s_i. Preden postane tako brisanje sploh mogoče, smo morali v celoti pobrisati podniz med znakoma s_i in s_k, torej podniz $s_{i+1} \ldots s_{k-1}$; za to potrebujemo $f(i + 1, k - 1)$ brisanj. Po tistem nam od niza $s_i s_{i+1} \ldots s_{k-1} s_k \ldots s_j$ ostane le $s_i s_k \ldots s_j$; ker sta znaka s_i in s_k enaka, bomo lahko s_i vsekakor pobrisali z istim brisanjem, ki bo pobrisalo s_k; torej je potrebno število brisanj za niz $s_i s_k \ldots s_j$ enako kot za niz $s_k \ldots s_j$, to pa je $f(k, j)$ brisanj. Tako imamo torej

skupaj $f(i+1, k-1) + f(k, j)$ brisanj, kar je še eden od kandidatov za $f(i, j)$.

Med vsemi tako dobljenimi kandidati za $f(i, j)$ vzamemo najmanjšega:

$$f(i,j) = \min\{\ 1 + f(i+1, j-1),\\ \min\{f(i+1, k-1) + f(k, j-1) : i < k \le j,\ s_i = s_k\}\ \}.$$

Robni primeri nastopijo pri $j < i$, ko imamo v resnici prazen podniz in brisanj sploh ne potrebujemo; takrat je torej $f(i, j) = 0$. Opazimo lahko, da za izračun $f(i, j)$ potrebujemo vrednosti funkcije f za krajše podnize znotraj $s_i \ldots s_j$, torej je koristno vrednosti funkcije f računati od krajših podnizov proti daljšim in si jih shranjevati v neko tabelo, odkoder jih potem lahko prebiramo, kadarkoli jih spet potrebujemo. Tako pridemo do naslednje rešitve s časovno zahtevnostjo $O(n^3)$:

```
#include <stdio.h>

#define MaxN 1000

/* f[i][j] bo najmanjše potrebno število brisanj, s katerim
   lahko popolnoma pobrišemo niz s[i...j-1]. */
char s[MaxN + 3];
int f[MaxN + 1][MaxN + 1];

int main()
{
  int i, j, d, n, naj, kand;

  /* Preberimo vhodni niz. */
  FILE *g = fopen("brisanje.in", "rt");
  fscanf(g, "%d\n", &n);
  fgets(s, sizeof(s), g); fclose(g);

  /* Pri podnizih dolžine 0 ni treba nobenih brisanj. */
  for (i = 0; i <= n; i++) f[i][i] = 0;

  /* Daljše podnize obdelajmo po naraščajoči dolžini. */
  for (d = 1; d <= n; d++) for (i = 0; i + d <= n; i++)
  {
    /* Kako najceneje pobrisati s[i...i+d-1]?
       Ena možnost je, da pobrišemo s[i] samega zase in nato brišemo s[i+1...i+d-1]. */
    naj = 1 + f[i + 1][i + d];
    for (j = i + 1; j < i + d; j++) if (s[j] == s[i])
    {
      /* Lahko pa poskušamo s[i...i+d] pobrisati tako, da tisto brisanje,
         ki pobriše s[i], pobriše tudi s[j] in nobenega od vmesnih znakov.
         Vmes moramo torej s[i+1...j-1] pobrisati v celoti, cena brisanja tega, kar
         ostane — to je s[i] + s[j...i+d-1] — pa je enaka kot cena brisanja
         s[j...i+d-1] samega po sebi, ker sta znaka s[i] in s[j] enaka in bomo lahko
         s[i] pobrisali z istim brisanjem, ki pobriše tudi s[j]. */
      kand = f[i+1][j] + f[j][i + d];
      if (kand < naj) naj = kand;
    }
    f[i][i + d] = naj;
  }

  /* Izpišimo rezultat. */
  g = fopen("brisanje.out", "wt"); fprintf(g, "%d\n", f[0][n]); fclose(g); return 0;
}
```

Za kratke nize, kakršni so bili pri našem tekmovanju pri 40 % testnih primerov, pa bi deloval dovolj hitro tudi kakšen preprost rekurzivni postopek, ki poskuša na vse možne načine izbrisati nek podniz strnjenih znakov iz niza s in nato izvede rekurzivni klic za niz, ki ostane po tem brisanju (če še ni prazen).

Viri nalog: tipkanje, h-indeks — Tomaž Hočevar; sorodstvo — Boris Horvat; virus — Vid Kocijan; asteroidi — Jurij Kodre; dominosa — Mitja Lasič; letala — Matjaž Leonardis; zoom — Matija Lokar; izštevanka, suhi dnevi — Mark Martinec; analiza enot, za žužke gre — Jure Slak; galaktična zavezništva — Patrik Zajec; zaklepajski izrazi, brisanje niza — Janez Brank.

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