# 9. tekmovanje ACM v znanju računalništva za srednješolce

29. marca 2014

## 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.");
   }
}
```

# 9. tekmovanje ACM v znanju računalništva za srednješolce

29. marca 2014

## 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).

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. Dnevnik

V računalniškem sistemu imamo zabeležene dogodke in bi jih radi v strnjeni obliki shranjevali na datoteko. Vsak dogodek (kot na primer prijava ali odjava uporabnika, razne napake) je predstavljen z nizom znakov (kratko besedilo / vrstica), ne daljšim od 100 znakov. Dogodki so zapisani na vhodni datoteki, po en dogodek v vsaki vrstici.

Na voljo imamo podprogram (funkcijo) PreberiDogodek(), ki prebere naslednji dogodek z vhodne datoteke in ga vrne kot niz znakov. Če smo že na koncu vhodne datoteke in novih dogodkov ni več, ta podprogram vrne prazen niz.

Ker se nekateri dogodki včasih zgodijo večkrat zapored (ne da bi se vmes zgodil kakšen drug dogodek), jih lahko zapišemo na izhodno datoteko v skrajšani obliki: ponovitve zadnjega izpisanega dogodka le štejemo in jih ne izpisujemo. Ko se kasneje pojavi nek drugačen dogodek, izpišemo le, da se je zadnji izpisani dogodek ponovil še $n$-krat. V primeru, da gre le za eno dodatno ponovitev ($n = 1$, torej skupaj s prvo izpisano pojavitvijo dve pojavitvi, glej zgled), namesto sporočila o ponovitvi izpišemo kar sam ponovljeni dogodek, saj ne bi s sporočilom o ponovitvi nič prihranili.

Tako bi denimo naslednje zaporedje dogodkov:

```
aaa
bbbbb
ccc
ccc
ccc
```

```
dd
dd
aaa
aaa
aaa
aaa
```

izpisali v strnjeni obliki kot:

```
aaa
bbbbb
ccc
ponovljeno se 2-krat
dd
dd
aaa
ponovljeno se 3-krat
```

**Napiši program**, ki bo bral dogodke z vhodne datoteke in jih, takoj ko bo to mogoče, izpisoval v tukaj opisani strnjeni obliki. (Dogodke lahko bereš po svoje s pomočjo standardnih funkcij za delo z datotekami ali pa uporabiš zgoraj omenjeno funkcijo PreberiDogodek.)

### 2. Proizvodnja čopičev

V Ajdovščini tovarna Wlahna d. o. o. proizvaja krasne veganske bio čopiče, *v celoti* narejene iz lesa. Leseni ročaji so tako ali tako nekaj običajnega, v tej tovarni pa celó konico čopiča izdelajo iz lesa iste vrste, ki ga zmeljejo in predelajo v celulozna vlakna.

V skladišču podjetja imajo $n$ lesenih paličk enake debeline, a različnih dolžin, iz katerih želijo izdelati same enake čopiče. Za posamezen ročaj potrebujejo $r$ centimetrov lesa v enem kosu. Za konico čopiča pa potrebujejo toliko zmletega lesa, kot ga nastane iz $k$ centimetrov ene ali več paličk.

**Opiši postopek** (ali napiši program, če ti je lažje), s katerim bi ugotovil, kolikšno je največje število čopičev, ki jih podjetje s trenutno zalogo lesa lahko proizvede. Števila $n$, $r$ in $k$ so podana in so naravna števila. Prav tako so podane dolžine paličk; lahko si recimo predstavljaš, da nekje obstaja tabela (*array*) $L$, v katerem $i$-ti element opisuje dolžino $i$-te paličke v centimetrih (tudi dolžine paličk so naravna števila).

### 3. Pacifistični generali

Ker imajo vse svetovne vojaške velesile jedrsko orožje, ga moramo nujno imeti tudi pri nas v Sloveniji. Pa je vlada naročila na IJS izdelavo jedrskih konic, od Rusov pa so kupili bojno plovilo VNL-11 Triglav, kjer so smrtonosne rakete zdaj shranjene. Dostopa do tako uničujočega orožja ne sme imeti kdorkoli, ampak ga ima samo peščica najpomembnejših generalov. Ker bi se lahko med generali našel norec, ki bi za zabavo poslal raketo ali dve na katero od sosednjih republik, so se na vojaškem ministrstvu odločili za stroge varnostne ukrepe. Naročili so izdelavo $k$ različic ključev za aktivacijo jedrskih konic. Ključi so oštevilčeni s števili od 1 do $k$. Vsaka različica ključa je bila izdelana v več izvodih. Te ključe so nato razdelili med $n$ generalov, tako da je vsak prejel neko podmnožico (samih različnih) ključev. Označimo z $G_i$ podmnožico ključev, ki jih ima $i$-ti general. Skupina generalov lahko sproži atomsko bombo, če imajo vsi skupaj vsaj po eno kopijo vsakega od ključev.

Zgled: recimo, da imamo $k = 3$ ključe in $n = 3$ generale. Naj bo

$$G_1 = \{1, 2\}, G_2 = \{2, 3\}, G_3 = \{1, 3\}.$$

Prvi general ima torej ključ št. 1 in ključ št. 2. Drugi general ima ključ št. 2 in ključ št. 3. Tretji general ima ključ št. 1 in ključ št. 3. Vsak posamezni general ne more aktivirati jedrske konice, če pa se združita npr. prvi in drugi general, imata skupaj $G_1 \cup G_2 = \{1, 2, 3\}$ vse ključe.

Na ministrstvu so pred kratkim opazili fenomen, na katerega pri snovanju sistema sploh niso pomislili. Med generali je vedno več pacifistov, ki niso pod nobenimi pogoji pripravljeni uporabiti jedrskega orožja. Vlada se zdaj boji, da bi lahko manjša skupinica pacifistov ostalim generalom preprečila uporabo orožja. Rekli bomo, da je sistem $r$-*odporen*, če nobena skupina $r$ (ali manj) generalov ne more ostalim preprečiti aktivacije jerskih konic.

Sistem iz zgleda je 1-odporen, saj nobeden od generalov ne more sam „blokirati“ ostalih dveh.

**Opiši postopek** (ali napiši program, če ti je to lažje), ki bo za dani $r$ in dani sistem preveril, če je ta sistem $r$-odporen. Opiši tudi, kako bi tvoja rešitev hranila in organizirala podatke (množice $G_1, \ldots, G_n$ in podobno).

### 4. Uniforme

Podjetje Wlahna d. o. o. se je odločilo svoje delavce obleči v praktične, trpežne uniforme iz debelega platna, skozi katerega jih ne bodo mogle bosti lesene trske, ki jih je v proizvodni hali podjetja vse polno. Uniforma sestoji iz treh kosov: hlač, jopiča in rokavic. Vsak kos uniforme je dobavljiv v velikostih od 1 do 100.

V podjetje je pravkar prispela nova pošiljka kosov uniform. Ko so jih razkladali s tovornjaka, so sproti popisali vsak kos uniforme (recimo: „hlače velikosti 73“). Zdaj jih zanima, koliko popolnih uniform lahko sestavijo. Popolna uniforma sestoji iz hlač, jopiča in rokavic v enaki velikosti.

**Napiši program**, ki prebere podatke o razpoložljivih kosih uniforme in izpiše največje število popolnih uniform, ki se jih da sestaviti. Podatke lahko bereš s standardnega vhoda ali pa iz datoteke, kar ti je lažje. Podatki imajo naslednjo obliko: v prvi vrstici je zapisano število dostavljenih kosov $n$. Vsaka od naslednjih $n$ vrstic vsebuje dve števili, ločeni s presledkom, in opisuje posamezen kos uniforme. Prvo število (1, 2, ali 3) opisuje tip kosa (hlače, jopič, ali rokavice), drugo število (med 1 in 100) pa velikost.

Primer vhoda:

```
15
3 98
1 45
1 74
1 45
2 98
1 45
2 45
1 74
2 74
2 98
2 74
3 74
3 74
1 98
2 74
```

Pripadajoči izpis:

```
3
```

Komentar: pri teh vhodnih podatkih lahko sestavimo tri popolne uniforme (in sicer dve uniformi velikosti 74 in eno uniformo velikosti 98).

### 5. Davek na ograjo

Slovenski državni urad za odkrivanje novih davkov je ugotovil, da ima vsak Slovenec svoje posestvo omejeno z ograjo in da za to še ni predpisanega nobenega davka. Zato so hitro naročili svojim matematikom, naj pripravijo informativni izračun.

Matematiki so seveda takoj brez škode za splošnost predpostavili, da je Slovenija pravokotna mreža $w \times h$ kvadratkov, kjer vsak kvadratek predstavlja en kvadraten kilometer, in da je vsa zemlja razdeljena med $n$ lastnikov, pri čemer ima vsak kvadratek samo enega lastnika. Nato so oštevilčili vse lastnike zemlje s števili od 0 do $n-1$, vsak kvadratni kilometer zemlje pa so označili s številko lastnika. Torej, kot na skici: polja, označena z 0, pripadajo lastniku 0; polja, označena z 1, pripadajo lastniku 1 in tako naprej.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 2 | 2 | 3 |
| 1 | 1 | 2 | 2 | 4 | 4 | 4 |
| 5 | 5 | 5 | 0 | 4 | 4 | 4 |
| 4 | 6 | 6 | 6 | 6 | 4 | 4 |
| 4 | 4 | 6 | 3 | 6 | 6 | 6 |

Vsa posestva so popolnoma ograjena: ograje stojijo na vseh državnih mejah in povsod tam, kjer kvadratek enega lastnika meji na kvadratek drugega lastnika. Na zgornji sliki so ograje narisane z debelimi črtami.

Davek se zaračuna sorazmerno z dolžino ograje, ki obdaja posestva posameznega lastnika. Ker vsi lastniki trdijo, da ograja spada k njihovemu zemljišču in ne sosednjemu, se jo vsem lastnikom tudi zaračuna — večina ograj se tako šteje dvakrat. **Napiši program**, ki za vsakega lastnika izračuna in izpiše skupno dolžino ograj, od katerih bo moral plačati davek. Obliko izpisa si izberi sam. Predpostavi, da že obstajajo naslednje funkcije, ki vračajo podatke o mreži:

- Visina() vrne $h$, torej višino mreže (celo število, vsaj 1 in največ 250);
- Sirina() vrne $w$, torej širino mreže (celo število, vsaj 1 in največ 250);
- StLastnikov() vrne $n$, torej število različnih lastnikov (največ $w \cdot h$);
- Lastnik(x, y) vrne številko lastnika za celico $(x, y)$; to je celo število od 0 do $n-1$. Pri tem je $x$ številka stolpca (od 0 do $w-1$), $y$ pa številka vrstice (od 0 do $h-1$).

Za primer z gornje slike bi moral tvoj program ugotoviti, da je za lastnika 0 skupna dolžina ograj enaka 8, za lastnika 1 je skupna dolžina 10, za lastnika 2 je skupna dolžina 12, za lastnika 3 je skupna dolžina 8, za lastnika 4 je skupna dolžina 20, za lastnika 5 je skupna dolžina 8, za lastnika 6 pa je skupna dolžina ograj enaka 18.

# 9. tekmovanje ACM v znanju računalništva za srednješolce

29. marca 2014

## 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.

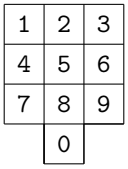
Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi 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).

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. Vnos šifre

Dana je številčna tipkovnica, na kateri so tipke kvadratne oblike in razporejene v pravokotniku podoben lik, kot kaže naslednja slika:




Radi bi si izbrali neko $n$-mestno zaporedje števk, ki ga bomo uporabljali kot šifro oz. geslo. Da ga bo čim lažje tipkati, si želimo, da bi se vsak par zaporednih števk v šifri tipkal z isto tipko ali pa s tipkama, ki sta si na gornji tipkovnici sosedi. Pri tem sosednost pomeni, da imata tipki skupno eno od stranic; na primer, v šifri se lahko takoj za števko `4` pojavi števka `1`, `5` ali `7`, ne pa `2` ali `8`. Nekaj primerov veljavnih 6-mestnih šifer: `414558`, `696969`, `089632`.

**Napiši podprogram** `NastejSifre(n)`, ki kot parameter dobi naravno število $n$ in izpiše vse $n$-mestne šifre, ki ustrezajo opisani omejitvi. Pri tem je vseeno, v kakšnem vrstnem redu jih program izpiše, mora pa izpisati vsako primerno šifro natanko enkrat.

*Lažja različica naloge:* če ti je dosedanja naloga pretežka, lahko rešiš lažjo različico, pri kateri ima $n$ vedno vrednost 6, torej nas zanimajo le 6-mestne šifre. Za rešitev te različice lahko dobiš pri tej nalogi največ 15 točk (od 20 možnih).

### 2. Prenova ceste

V bližini mesteca Cocklebiddy v Zahodni Avstraliji se nahaja najdaljša popolnoma ravna cesta na svetu. Zaradi nedavnih poplav morajo prenoviti 100 km ceste. Za popravilo ceste kandidirata dve podjetji, Kangaroads Ltd. in Wallabyway Inc. Vzdolž cestnega odseka, ki ga je treba popraviti, živi 1 000 000 prebivalcev[1] in vsak od njih ima svoje mnenje o tem, katero podjetje bi moralo popravljati cesto.

Na koncu so se prebivalci odločili za kompromis: cesto lahko popravljata obe podjetji, pri čemer bo vsak še tako majhen košček ceste popravilo tisto podjetje, za katerega glasuje večina od $k$ najbližje stanujočih prebivalcev; $k$ je liho število. (Za namen te naloge je cesta daljica, stanovanja prebivalcev pa so točke na njej.)

**Opiši postopek**, ki izračuna, koliko kilometrov ceste bo popravilo podjetje Kangaroads Ltd. in koliko Wallabyway Inc. Predpostavi, da že obstajajo naslednje spremenljivke: `k`, kot je opisan zgoraj; tabela `x`, kjer je `x[i]` realno število med 0 in 100 in opisuje položaj $i$-tega prebivalca (oddaljenost od zahodnega konca popravljanega odseka ceste, v kilometrih), ter tabela `p`, kjer je `p[i]` enak 0, če $i$-ti prebivalec glasuje za podjetje Kangaroads Ltd., in 1 sicer. Vse koordinate prebivalcev so različne in so podane v naraščajočem vrstnem redu.

[1] Poplave in kakšnih 999 950 prebivalcev je izmišljenih; tisto o najdaljši ravni cesti je pa res, dolga je 90 milj.

### 3. Skrivno sporočilo

Si vohun SOVE, ki dela pod krinko in ravnokar si „opravil“ z agentom zlobne zločinske organizacije OREL, ki je poskušal poslati podatke o času in kraju dostave pomembnega paketa svojim nadrejenim. Pri nakazovanju svoje superiornosti si bil malce pregrob, tako da agenta sedaj ne moreš izprašati, temveč nemočno držiš v rokah njegovo komunikacijsko napravo, kjer se na zaslonu bleščita sporočilo, ki ga je agent želel poslati, in njegova napol zašifrirana kopija. Tvoja želja je, da v njegovem imenu pošlješ svoje sporočilo, ki bo seveda vsebovalo napačen kraj in čas dostave, da boste lahko ne le varno prejeli paket, temveč tudi ujeli še kakšnega agenta.

Agenti ORLA svoja sporočila šifrirajo zelo primitivno, črke abecede le malo zamešajo med seboj in pišejo namesto `a` na primer `r` in podobno. Tvoja naloga je, da preveriš, ali je delno zašifrirano sporočilo veljavno, in čim bolj ugotoviš šifro ter na enak način, kolikor je le mogoče, zašifriraš svoje sporočilo. Zašifirano sporočilo je veljavno, če se enaki črki vedno zašifirirata v enaki črki, poleg tega pa se morata različni črki šifrirati v različni črki, da je sporočilo mogoče odkodirati. (Bolj matematično: kodirna funckcija mora biti bijektivna). Primera neveljavnih kodiranj sta `cat` → `bgb` in `zoo` → `srt`, primer veljavnega kodiranja pa je npr. `please` → `rlagha`.

**Napiši podprogram** (funkcijo) Desifriraj(`p1`, `c1`, `p2`), ki kot parametre dobi tri nize:

- `p1` je originalno (nešifrirano) sporočilo.
- `c1` je delno šifrirano sporočilo, ki smo ga zasegli sovražnemu agentu. Ta niz je enako dolg kot originalno sporočilo. V njem nastopajo že zašifrirane črke; na mestih, ki jih agent še ni utegnil zašifrirati, pa je namesto črke zvezdica (znak `*`).
- `p2` je tvoje sporočilo, ki ga želiš zašifrirati po enakem postopku, kot ga je uporabil sovražni agent.

Vsa sporočila vsebujejo samo male črke angleške abecede in so krajša od 10000 znakov. Tvoj podprogram naj izpiše zašifrirano različico tvojega sporočila `p2` (če za nekatere znake ni mogoče zanesljivo ugotoviti, v kaj bi se morali zašifrirati, namesto njih izpiši zvezdico `*`); če pa dano šifrirano sporočilo ni veljavno, izpiši „`neveljavna šifra`“.

Primer: recimo, da dobimo nize

```
p1 = nexttuesdayfourfiftypminthemainsquare
c1 = r**aa**k***s****e***q*e**o********wl*
p2 = thistuesdayfourfiftypmanddefinetlynotnexttuesdayfourfiftypminthemainsquare
```

Pravilni izpis je potem takšen:

```
aoeka**k*w*s**lsesa*q*wr***ser*a**r*ar**aa**k*w*s**lsesa*q*erao**werk**wl*
```

**4. Potenciranje**

Predstavljajmo si zelo preprost, zbirniku podoben programski jezik. Program je sestavljen iz zaporedja ukazov; pred vsakim ukazom je lahko še oznaka (labela), na katero se lahko sklicujemo pri pogojnih skokih. Dovoljeni ukazi so naslednji:

- ADD x, y — izračuna vsoto $x + y$ in jo shrani v $x$;
- SUB x, y — izračuna razliko $x - y$ in jo shrani v $x$;
- MUL x, y — izračuna zmnožek $x \cdot y$ in ga shrani v $x$;
- DIV x, y — izračuna celi del količnika $x/y$ in ga shrani v $x$;
- MOD x, y — izračuna ostanek po deljenju $x$ z $y$ in ga shrani v $x$;
  Opomba: pri gornjih ukazih mora biti $x$ spremenljivka, $y$ pa je lahko spremenljivka ali celoštevilska konstanta.
- JL x, y, z — pogojni skok: če je $x < y$, skoči na ukaz z oznako (labelo) $z$. Pri tem sta $x$ in $y$ lahko spremenljivki in/ali celoštevilski konstanti. Če pogoj $x < y$ ni izpolnjen, se izvajanje nadaljuje z naslednjim ukazom.

Program lahko uporablja poljubno mnogo spremenljivk. Vse spremenljivke so celoštevilske (kot tip **int** oz. integer, le da lahko za razliko od teh tipov hranijo poljubno velika cela števila) in jih pred uporabo ni treba posebej deklarirati. Ukazi se izvajajo po vrsti, razen če pride do pogojnega skoka (ukaz JL).

V tem programskem jeziku **napiši program**, ki izračuna vrednost $a^b$ in jo shrani v spremenljivko $c$. Pri tem je $b$ naravno število. (Predpostavi, da imata spremenljivki $a$ in $b$ želeni začetni vrednosti že pred začetkom izvajanja tvojega programa, torej se ti ni treba ukvarjati s tem, kako bi ju prebral ali incializiral.) Primer: če je pred začetkom izvajanja tvojega zaporedja ukazov v spremenljivki $a$ vrednost 2, v spremenljivki $b$ pa vrednost 3, mora biti na koncu izvajanja tvojega programa v spremenljivki $c$ vrednost 8. Tvoj program naj pri tem izvede čim manj ukazov za velika števila $a$ in $b$.

Svojo rešitev tudi dobro utemelji in komentiraj, kako in zakaj deluje.

Spomnimo se, da je $b$-ta potenca števila $a$ definirana takole:

$$a^b = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ členov}}$$

in da za potence med drugim velja

$$a^{b+c} = a^b \cdot a^c.$$

*Namig:* najprej razmisli, kako bi učinkovito računal $a^2, a^4, a^8, a^{16}$ itd., nato pa še o tem, kako bi s tem prišel do rešitve za poljuben $b$.

Za ilustracijo je tule primer programa, ki rešuje malo drugačen problem: v spremenljivki vsota izračuna vsoto števil od 1 do $n$ (ob predpostavki, da je $n$ večji od 0):

```
                                Razlaga (ni del programa)
      SUB vsota, vsota          postavi vsota na 0
lab1: ADD vsota, n              prišteje vsoti trenutno vrednost n
      SUB n, 1                  zmanjša n za 1
      JL 0, n, lab1             ponavlja, dokler je n > 0
```

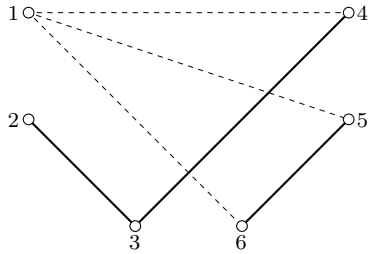
Pa še en primer: spodnji program računa isto vsoto, vendar namesto zanke uporabi formulo $1 + 2 + \ldots + n = n \cdot (n + 1)/2$.

```
                                Razlaga (ni del programa)
SUB vsota, vsota                postavi vsota na 0
ADD vsota, n                    vsota je zdaj enaka n
ADD vsota, 1                    vsota je zdaj enaka n + 1
MUL vsota, n                    vsota je zdaj enaka n(n + 1)
DIV vsota, 2                    vsota je zdaj enaka n(n + 1)/2
```

### 5. Tiskana vezja

Ker se Štefko zanima za elektroniko, je za rojstni dan dobil začetniški komplet za izdelavo tiskanih vezij. Paket vsebuje več plošč, ki že imajo narejene luknjice za priključke. Poleg tega je dobil orodje, s katerim lahko med poljubnima dvema luknjicama (priključkoma) nariše *ravno* (bakreno) povezavo. Luknjice so postavljene na ploščo na tak zvit način, da nobena ravna črta med dvema luknjicama ne prečka katere druge luknjice. Štefko je že pripravil načrte za nekaj genialnih vezij, ko pa jih je hotel narisati, je opazil — ojoj! — da bi se nekatere daljice med seboj sekale, če bi jih zares narisal na ploščo. Ko je vezje načrtoval, je mislil samo na to, kateri pari priključkov morajo biti med seboj povezani.

Ampak ni še vse izgubljeno. Štefko je kmalu opazil, da je plošča dvostranska! Razmišljal je takole: kaj pa, če nekaj povezav narišem na eno, nekaj pa na drugo stran plošče? Potem morda lahko dosežem, da se nobeni dve povezavi ne bosta sekali? Zdaj ga zanima, katera vezja je možno narisati, če lahko uporabi obe strani plošče. Ker je problem za Štefka preveč zapleten, potrebuje tvojo pomoč. **Opiši postopek**, ki ugotovi, ali je mogoče dano vezje narisati na opisani način, ne da bi se kakšni dve povezavi sekali. Pri tem tvoj postopek kot vhodne podatke dobi koordinate vseh luknjic (luknjice oštevilčimo od 1 do $n$ in recimo, da ima $i$-ta luknjica koordinate $(x_i, y_i)$) in seznam parov luknjic, ki jih je treba povezati. Opiši tudi, kako bi v svoji rešitvi predstavil in organiziral te podatke v pomnilniku. Predpostavi, da že obstaja funkcija SeSekata(ax, ay, bx, by, cx, cy, dx, dy), ki vrne **true**, če se daljica od točke $(a_x, a_y)$ do točke $(b_x, b_y)$ seka z daljico od točke $(c_x, c_y)$ do točke $(d_x, d_y)$, sicer pa vrne **false**.




*Primer:* gornja slika kaže vezje, ki bi se ga dalo narisati na opisani način; pri tem debele polne črte pomenijo povezave na eni strani plošče, tanke črtkane črte pa povezave na drugi strani plošče. Če pa bi hoteli temu vezju dodati še povezavo med luknjicama 2 in 5, se ga ne bi več dalo narisati brez križanja povezav.

# 9. tekmovanje ACM v znanju računalništva za srednješolce

29. marca 2014

## 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 imenik `U:\_Osebno`, v katerem lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C# ali java, mi pa jih bomo preverili s 64-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz OpenJDK 7 in s prevajalnikom Mono 4 za C#. Za delo lahko uporabiš `FP` oz. `ppc386` (Free Pascal), `gcc`/`g++` (GNU C/C++ — command line compiler), `javac` (za javo 1.7), Visual Studio in druga orodja.

Na spletni strani `http://rtk/` oz. `http://rtk.std.fmf.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`

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 desetih testnih primerih. Za vsak testni primer se bo izpisalo, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal 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 desetih testnih primerih; pri vsakem od njih dobi 10 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://rtk/`):

- 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();
   }
}
```

# 9. tekmovanje ACM v znanju računalništva za srednješolce

29. marca 2014

## NALOGE ZA TRETJO SKUPINO

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

### 1. Ljudožerci na premici (`ljudozerci.in`, `ljudozerci.out`)

Vzdolž premice stoji $n$ ljudožercev, ki so oštevilčeni od 1 do $n$, vendar ne nujno v kakšnem posebnem vrstnem redu (npr. od leve proti desni ali kaj podobnega). Ljudožerec s številko $i$ se nahaja na koordinati $x_i$ (to je celo število, večje ali enako 0). Te koordinate niso nujno različne; lahko se zgodi, da dva ali več ljudožercev stoji na isti točki.

En za drugim se na premico s padalom spusti še $m$ ljudi. Pri tem $j$-ti od njih naredi naslednje:

- Pristane na neki koordinati $a_j$.
- Pomaha najbližjemu ljudožercu (če je takšnih več, izbere tistega z manjšo koordinato; če je tudi takšnih več, pa tistega z najmanjšo zaporedno številko).
- Ljudožerec mu pomaha nazaj, pride k njemu (padalec ga počaka na točki $a_j$) in ga požre.

**Napiši program**, ki prebere začetne koordinate ljudožercev ter zaporedje prihodov padalcev, nato pa izpiše končne koordinate ljudožercev.

*Vhodna datoteka:* v prvi vrstici sta števili $n$ in $m$, ločeni s presledkom. Sledi $n$ vrstic; $i$-ta od njih vsebuje koordinato $x_i$, na kateri na začetku stoji posamezen ljudožerec. Sledi $m$ vrstic; $j$-ta od njih vsebuje koordinato $a_j$, na katero se spusti $j$-ti padalec. Veljalo bo $1 \leq n \leq 10^5$, $0 \leq m \leq 10^5$ in $0 \leq x_i \leq 10^9$, $0 \leq a_i \leq 10^9$.

*Izhodna datoteka:* izpiši končne koordinate vseh $n$ ljudožercev v naraščajočem vrstnem redu, vsako v svojo vrstico.

Primer vhodne datoteke:

```
6 4
30
10
50
40
30
20
22
35
37
6
```

Pripadajoča izhodna datoteka:

```
6
22
30
37
40
50
```

### 2. Po Indiji z avtobusom (`avtobus.in`, `avtobus.out`)

Čez celotno Indijo poteka v smeri zahod–vzhod dolga cesta. Poljuben kraj ob cesti lahko opišemo s številom kilometrov, ki jih je treba prepotovati od zahodnega konca ceste, da pridemo do tega kraja. Vzdolž ceste vozi $n$ avtobusov, $i$-ti od njih med krajema $a_i$ in $b_i$ v obe smeri. Avtobusi peljejo zelo počasi in imajo ves čas odprta vrata, da se vsaj malo zračijo; tako je možno na poljubni točki izstopiti iz avtobusa ali vstopiti nanj. Praveen želi potovati od mesta $x$ do mesta $y$. Pomagaj mu najti potovalni načrt, s katerim bo uporabil čim manj avtobusov.

*Vhodna datoteka:* v prvi vrstici so po vrsti zapisana cela števila $n$, $x$ in $y$, ločena s po enim presledkom. Vsaka od naslednjih $n$ vrstic vsebuje števili $a_i$ in $b_i$ za posamezen avtobus; to sta celi števili, ločeni z enim presledkom. Veljalo bo $1 \leq n \leq 200\,000$, $0 \leq x < y \leq 10^9$ in (za vsak $i$) $0 \leq a_i < b_i \leq 10^9$.

Vsi primeri so taki, da rešitev obstaja (torej da je res mogoče priti od kraja $x$ do kraja $y$).

V 50% testnih primerov bo veljalo tudi $n \leq 1000$.

*Izhodna datoteka:* izpiši najmanjše število avtobusov, s katerimi je možno prepotovati pot med $x$ in $y$.

Primer vhodne datoteke:

```
6 3 21
10 20
4 11
2 6
11 21
1 4
6 12
```

Pripadajoča izhodna datoteka:

```
3
```

Komentar: lahko se na primer z avtobusom 2–6 peljemo od 3 do 6, nato z avtobusom 6–12 do 11 in nato z avtobusom 11–21 do 21. Možni pa so še tudi drugi poteki potovanja, ki ravno tako pridejo do cilja s tremi avtobusi.

### 3. Luči (`luci.in`, `luci.out`)

Znašel si se v sobi z $L$ lučmi in $S$ stikali. Po daljšem preklapljanju stikal si ugotovil, da je vsako stikalo povezano z nekaj lučmi (lahko tudi z nobeno ali vsemi), pritisk nanj pa spremeni stanje luči, s katerimi je povezano — če je posamezna luč ugasnjena, se prižge in obratno. Ugotovil si tudi, da lahko na vsako luč vplivaš z največ dvema različnima stikaloma. Na vsak način bi rad ugasnil vse luči, zato si se naloge lotil sistematično in zabeležil, katera stikala so povezana s katerimi lučmi. Na koliko načinov lahko ugasneš vse luči? Ker je vsako stikalo smiselno uporabiti največ enkrat, izračunaj število podmnožic stikal, s pritiskom na katera ugasnemo vse luči.

*Vhodna datoteka:* v vhodni datoteki je več testnih primerov. Število primerov $T$ je podano v prvi vrstici, sledijo pa s praznimi vrsticami ločeni opisi posameznih testnih primerov. Vsak testni primer se začne z vrstico, ki vsebuje število luči $L$ in število stikal $S$, ločeni s presledkom. V naslednji vrstici se nahaja $L$ števil 0 ali 1 (ločena so s po enim presledkom). Če je luč $k$ trenutno ugasnjena, bo $k$-to število enako 0, sicer pa 1. Sledi še $S$ vrstic, ki opisujejo povezave stikal z lučmi. Prvo število v $i$-ti vrstici predstavlja število luči $n_i$, s katerimi je povezano stikalo $i$. V nadaljevanju vrstice je podanih še $n_i$ zaporednih številk luči, na katere vpliva to stikalo.

Veljalo bo $1 \leq T \leq 10$, $1 \leq L \leq 300$ in $1 \leq S \leq 300$. V 30 % testnih primerov bo veljalo tudi $L \leq 20$ in $S \leq 20$.

*Izhodna datoteka:* za vsak testni primer izpiši po eno vrstico, vanjo pa izpiši celo število, ki pove, na koliko načinov lahko pri tem testnem primeru s stikali, ki so nam na voljo, ugasnemo vse luči. Ker je to število lahko zelo veliko, izpiši le njegov ostanek pri deljenju z 1 000 000 007 ($10^9 + 7$).

Primer vhodne datoteke:

```
2

7 5
0 1 1 1 1 1 0
2 1 3
2 1 2
2 2 3
0
3 4 5 6

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

Pripadajoča izhodna datoteka:

```
4
0
```

Komentar: možni nabori stikal pri prvem testnem primeru so $\{1, 2, 5\}$, $\{1, 2, 4, 5\}$, $\{3, 5\}$ in $\{3, 4, 5\}$.

**4. Bloki** (`bloki.in`, `bloki.out`)

Mnogi urejevalniki besedil imajo ukaz, ki nas z znaka „{“ premakne na pripadajoči „}“ ali obratno; to pride prav pri urejanju izvorne kode v tistih programskih jezikih, ki z zavitimi oklepaji označujejo začetek in konec bloka oz. skupine stavkov (npr. zanke, funkcije ipd.). Kaj pa, če delamo v jeziku, ki zavitih oklepajev ne uporablja, pač pa pripadnost blokom izraža z zamikanjem vrstic (na primer python)?

Definirajmo *zamik vrstice* kot število presledkov na začetku vrstice. Predpostavi, da se drugih presledkom podobnih znakov (npr. tabulatorjev) ne uporablja. Vrstico, ki vsebuje same presledke, štejemo za prazno.

Blok je skupina več zaporednih vrstic, določena z naslednjimi pravili:

- V neki vrstici se začne blok, če je ta vrstica neprazna in
  - je to prva neprazna vrstica sploh ali pa
  - ima večji zamik kot prejšnja neprazna vrstica (pri tem ni pomembno, koliko praznih vrstic je med njima).
- Ta blok se potem nadaljuje do prve take neprazne vrstice, ki ima manjši zamik kot vrstica, s katero se je blok začel; ta že ne pripada več bloku (tista pred njo pa še, četudi je prazna). Če take vrstice ni, se blok nadaljuje do konca vhodne datoteke.

Ta definicija omogoča, da se bloki gnezdijo eden v drugem in da posamezna vrstica pripada več blokom. Nekaj primerov (bloki so označeni z oglatimi oklepaji na levi strani; presledki so prikazani z znakom ␣, da se jih bolje vidi):

```
[  In␣the␣second
[  ␣␣century␣of␣the
[  ␣␣Christian␣Aera,
[  [  ␣␣␣␣the␣empire␣of
[  ␣␣Rome␣comprehended
[  [  ␣␣␣␣the␣fairest
[  part␣of␣the
[  [  ␣␣earth,␣and␣the
[  [  ␣␣most␣civilized
```

```
[  ␣␣portion␣of
[  [  ␣␣␣␣␣␣mankind.␣The
[  [  ␣␣␣␣␣␣frontiers␣of
[  [  ␣␣␣␣that␣extensive
[  monarchy␣were
[  [  ␣␣guarded␣by
[  [  ␣␣ancient␣renown
```

```
[  ␣␣
[  and␣disciplined
[
[  [  ␣␣valor.␣The
[  [  ␣␣gentle␣but
[  [  ␣␣␣␣
[  powerful
[  [  ␣␣influence␣of
[  [  ␣
[  [  [  ␣␣␣␣laws␣and
[  [  [
[  manners␣had
[  [  ␣␣␣
```

Kot vidimo iz drugega in tretjega primera, se lahko zgodi tudi, da kakšna vrstica ne pripada nobenemu bloku.

**Napiši program**, ki prebere besedilo, poišče v njem vse bloke in za vsak blok izpiše, v kateri vrstici se začne in v kateri se konča.

*Vhodna datoteka:* v prvi vrstici je celo število $n$, ki pove, koliko vrstic je dolgo vhodno besedilo. Veljalo bo $1 \leq n \leq 10^5$. Sledi $n$ vrstic z besedilom, ki ga moraš obdelati. Vsaka vrstica je dolga največ 1000 znakov.

*Izhodna datoteka:* za vsak blok izpiši po eno vrstico, vanjo pa dve celi števili, ločeni s presledkom. Prvo od teh števil naj bo številka prve vrstice tega bloka, drugo pa številka zadnje vrstice tega bloka. Vrstice so oštevilčene od 1 do $n$. Bloke izpiši v naraščajočem vrstnem redu glede na številko začetne vrstice.

Primer vhodne datoteke:

```
10
gradually cemented the union
  of the provinces. Their peaceful
  inhabitants enjoyed and abused the
    advantages of wealth and luxury.
    The image of a free constitution
  was preserved with decent reverence:
the Roman senate appeared to possess
    the sovereign authority, and
    devolved on the emperors all the
      executive powers of government.
```

Pripadajoča izhodna datoteka:

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

**5. Poravnavanje desnega roba** (`poravnavanje.in`, `poravnavanje.out`)

Besedilo, ki se razteza čez več vrstic, je videti lepše, če je njegov desni rob poravnan — z drugimi besedami, v krajših vrsticah presledke malo razširimo, da so na koncu videti vse vrstice enako dolge (razen zadnje vrstice besedila; ta je lahko krajša od ostalih). Po drugi strani pa si ne želimo, da bi v kakšni vrstici nastale prevelike vrzeli med besedami. Zato se pri razbijanju besedila na vrstice včasih splača dati v kakšno vrstico manj besed, kot bi jih sicer lahko šlo vanjo, če nam to omogoči lepše razbiti preostanek besedila.

Recimo, da je naše besedilo dolgo $n$ besed in da so znane tudi širine vseh teh besed, $w_1, w_2, \ldots, w_n$ (pri čemer je $w_i$ širina $i$-te besede). Podana je tudi minimalna zahtevana širina presledka med besedami; recimo ji $s$. Presledki se pojavijo le med besedami, ne pa na začetku ali koncu vrstice. Predpisana je tudi zahtevana širina vrstice po poravnavanju desnega roba; recimo ji $d$.

Definirajmo zdaj *oceno vrstice* takole: recimo, da vrstica vsebuje besede od vključno $i$-te do vključno $j$-te. To je torej skupaj $j - i + 1$ besed, med katerimi mora biti zato $j - i$ presledkov; vsak od teh presledkov mora biti širok vsaj $s$. Skupna dolžina besed in teh minimalnih presledkov je torej $w_i + w_{i+1} + \ldots + w_j + (j-i) \cdot s$; če ta vsota presega $d$, potem take vrstice sploh ne smemo uporabiti, ker bi bila preširoka. Drugače pa vidimo, da bo treba to vrstico razširiti za $d - (w_i + w_{i+1} + \ldots + w_j + (j-i) \cdot s)$, da bo dosegla predpisano širino $d$. Za oceno vrstice vzemimo kvadrat tega števila:

$$\mathit{ocena}(i, j) = (d - (w_i + w_{i+1} + \ldots + w_j + (j-i) \cdot s))^2 .$$

Izjema nastopi na koncu besedila: ker pri zadnji vrstici ne poravnavamo desnega roba, vanjo tudi ne vrivamo dodatnih presledkov, zato vzamemo $\mathit{ocena}(i, n) = 0$.

Dano zaporedje besed želimo razbiti na vrstice tako, da nobena vrstica ne presega širine $d$ in da je vsota njihovih ocen najmanjša možna. **Napiši program**, ki izračuna najmanjšo možno vsoto ocen vrstic, ki jo je mogoče doseči na ta način.

*Vhodna datoteka:* v prvi vrstici so cela števila $n$, $s$ in $d$, ločena s po enim presledkom. Sledi $n$ vrstic, ki po vrsti podajajo cela števila $w_1, w_2, \ldots, w_n$. Velja $1 \leq n \leq 10^6$, $1 \leq s \leq d \leq 1000$ in (za vsak $i$) $1 \leq w_i \leq d$.

*Izhodna datoteka:* vanjo izpiši eno samo celo število, in sicer najmanjšo možno vsoto ocen vrstic, ki jo je mogoče doseči v skladu z zahtevami naloge.

Primer vhodne datoteke:

```
6 2 30
9
9
3
10
10
18
```

Pripadajoča izhodna datoteka:

```
89
```

Komentar primera: najboljše razbitje besed na vrstice je, da vzamemo prve tri besede v prvo vrstico, naslednji dve v drugo in zadnjo besedo v tretjo vrstico. Širina treh besed v prvi vrstici, skupaj z minimalnim razmikom ($s = 2$) med vsakima zaporednima besedama, je $9 + 2 + 9 + 2 + 3 = 25$, torej nam do $d = 30$ manjka še 5, zato je ocena te vrstice enaka $5^2 = 25$. Podobno je drugo vrstica z minimalnim razmikom široka $10 + 2 + 10 = 22$, torej ji do $d$ manjka še 8, zato je ocena te vrstice enaka $8^2 = 64$. Ocena zadnje vrstice je po definiciji 0. Končni rezultat je zato $25 + 64 + 0 = 89$.

# 9. tekmovanje ACM v znanju računalništva za srednješolce

29. marca 2014

## REŠITVE NALOG ZA PRVO SKUPINO

### 1. Dnevnik

Vhodno datoteko berimo po vrsticah in si poleg trenutne vrstice zapomnimo še prejšnjo vrstico in dosedanje število pojavitev te prejšnje vrstice (spremenljivki prejsnja in n). Na začetku izvajanja prejšnje vrstice še ni, zato postavimo n na 0.

Ko preberemo novo vrstico, jo primerjamo s prejšnjo; če sta enaki, ne izpišemo ničesar, pač pa le povečamo števec pojavitev. Če pa sta različni, to pomeni, da se dosedanji blok enakih vrstic (tistih, ki so enake nizu prejsnja) končuje in moramo najprej dokončati pripadajoči izpis zanj (saj smo doslej od tega bloka izpisali le prvo vrstico). Zdaj preverimo števec pojavitev n; če je enak 2, izpišemo vrstico prejsnja še enkrat, če pa je večji od 2, izpišemo niz oblike „ponovljeno še $(n-1)$-krat“. Nato izpišemo novo vrstico, si jo zapomnimo v spremenljivki prejsnja in postavimo števec n na 1.

Poseben primer nastopi na koncu vhodne datoteke, ko namesto nove vrstice dobimo prazen niz. Tega ne izpišemo, pač pa zanko takrat prekinemo. (Pred tem pa ga obravnavamo enako kot običajno vrstico, kar bo zagotovilo, da bomo pravilno zaključili izpis zadnjega bloka nepraznih vrstic.)

Zapišimo dobljeni postopek v pythonu:

```
prejsnja = ""; n = 0
while True:
  nova = PreberiDogodek()
  if n > 0 and prejsnja == nova: n += 1; continue
  if n == 2: print "%s" % prejsnja
  elif n > 2: print "ponovljeno se %d-krat" % (n - 1)
  prejsnja = nova; n = 1
  if nova: print nova
  else: break
```

Rešitev v C-ju je malenkost bolj zapletena, ker je C za delo z nizi malo bolj neroden. Predpostavimo, da PreberiDogodek ob vsakem klicu alocira nov niz na kopici in nam vrne kazalec nanj (tipa **char ***), dealokacija tega pomnilnika pa je naša skrb; na koncu vhodne datoteke pa naj PreberiDogodek vrne 0. (Čisto dobro bi se dalo narediti tudi drugače, na primer tako, da bi podprogram PreberiDogodek shranil niz v neko tabelo, ki bi mu jo podali kot parameter.) Razlika v primerjavi z gornjo pythonovsko rešitvijo je torej predvsem ta, da niz nova po obdelavi posamezne vrstice dealociramo (kličemo standardno funkcijo free), razen če smo si ta niz zapomnili tudi kot prejsnja; v tem primeru pa moramo dealocirati stari niz prejsnja, preden mu priredimo nova. (Na primer, ko je prejsnja v prvi iteraciji zanke še 0, nam ni treba posebej paziti, saj je klic free(0) dovoljen in po definiciji ne naredi ničesar.)

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

extern char *PreberiDogodek();

int main()
{
  char *prejsnja = 0, *trenutna = 0; int n = 0;
  do
  {
    nova = PreberiDogodek();
    if (n > 0 && nova && strcmp(prejsnja, nova) == 0) { n++; free(nova); continue; }
    if (n == 2) printf("%s\n", prejsnja);
```

```
    else if (n > 2) printf("ponovljeno se %d-krat\n", n - 1);
    n = 1; free(prejsnja); prejsnja = nova;
    if (nova) printf("%s\n", nova);
  }
  while (nova != 0);
  return 0;
}
```

### 2. Proizvodnja čopičev

Dolžino $i$-te palice označimo z $L_i$. Če to dolžino delimo z $r$, nam celi del količnika pove, koliko ročajev bi se dalo narediti iz te palice. Če seštejemo to po vseh palicah, dobimo maksimalno število ročajev, ki bi se jih dalo narediti iz razpoložljivih palic. To je tudi zgornja meja za število čopičev, ki bi se jih dalo izdelati, saj za vsak čopič potrebujemo po en ročaj.

Ni pa nujno, da toliko čopičev res lahko naredimo, saj nam mora po izdelavi ročajev ostati še dovolj lesa za konice. Če bi radi na primer naredili $c$ čopičev, potrebujemo poleg ročajev še $c \cdot k$ lesa za izdelavo konic. Če od skupne dolžine palic $L_1 + \ldots + L_n$ odštejemo skupno dolžino ročajev, $c \cdot r$, nam razlika pove, koliko lesa ostane na razpolago za izdelavo konic. Ta razlika mora biti torej vsaj $c \cdot k$, sicer toliko čopičev ne bomo mogli izdelati. Tako imamo neenačbo $L_1 + \ldots + L_n - c \cdot r \geq c \cdot k$, iz česar dobimo $c \leq (L_1 + \ldots + L_n)/(k + r)$.

Naše število čopičev mora ustrezati obema omejitvama (da bo dovolj ročajev in tudi dovolj lesa za konice), zato moramo od obeh doslej dobljenih zgornjih mej vzamemo nižjo.

Zapišimo našo rešitev še v C-ju:

```
int KolikoCopicev(int n, int k, int r, int L[])
{
  int skupDolzina = 0, maxRocajev = 0, maxCopicev, i;
  for (i = 0; i < n; i++) {
    skupDolzina += L[i];
    maxRocajev += L[i] / r; }
  maxCopicev = skupDolzina / (k + r);
  if (maxRocajev < maxCopicev) maxCopicev = maxRocajev;
  return maxCopicev;
}
```

### 3. Generali

Iz definicije v opisu naloge sledi, da lahko skupina generalov prepreči sprožitev bombe, če niti vsi ostali generali skupaj nimajo vseh ključev. Z drugimi besedami, to se zgodi takrat, ko za vsaj en ključ velja, da so vsi izvodi tega ključa v rokah generalov iz naše pacifistične skupine. Sistem je torej $r$-odporen, če ima vsak ključ vsaj $r + 1$ generalov (saj takrat gotovo velja, da kakorkoli izberemo $r$ generalov, ne bomo mogli obvladovati vseh izvodov posameznega ključa; in po drugi strani, če bi obstajal kak ključ v $r$ ali manj izvodih, potem bi se dalo izbrati $r$ generalov tako, da bi obvladovali vse izvode tega ključa in bi lahko ostalim preprečili sprožitev bombe).

Naš postopek mora torej za vsak ključ prešteti, koliko generalov ga ima. V ta namen si lahko pomagamo s tabelo $k$ elementov, ki za vsak ključ povedo, pri koliko generalih smo ga doslej opazili. V zanki se sprehodimo po vseh generalih in pri vsakem od njih po vseh njegovih ključih ter povečujemo števce v tabeli. Na koncu le še preverimo, če so vsi elementi tabele večji od $r$. Zapišimo ta postopek še s psevdokodo:

**algoritem** JeOdporen($r$, $G_1$, ..., $G_n$):
  za vsak ključ $j$ od 1 do $k$ postavi *štPojavitev*[$j$] na 0;
  za vsakega generala $i$ do 1 do $n$:
    za vsak ključ $j$ iz množice $G_i$:
      povečaj *štPojavitev*[$j$] za 1;
  za vsak ključ $j$ od 1 do $k$:
    **if** *štPojavitev*[$j$] $\leq r$ **then return false**;
  **return true**;

Naloga sprašuje tudi, kako bi v računalniku predstavili množice $G_i$. Ena možnost je, da predstavimo vsako tako množico s tabelo $k$ logičnih vrednosti, za vsak ključ po eno, ki nam pove, ali general $i$ ta ključ ima ali ne. (Še bolj varčna različica te predstavitve je tabela $k$ bitov, torej približno $k/8$ bytov.) Slabost takšne predstavitve je, da ko hočemo pregledati ključe, ki jih ima general, moramo iti v zanki po vseh možnih ključih od 1 do $k$ in za vsakega od njih v tabeli preverjati, ali ga ta general ima ali ne. To je še posebej neugodno, če so posamezne množice $G_i$ majhne v primerjavi s številom vseh možnih ključev (torej $k$).

Druga možnost (ki je za naš namen primernejša) pa je, da predstavimo množico $G_i$ s seznamom, v katerem so navedeni le tisti ključi, ki jih general $i$ dejansko ima. Ta seznam je lahko predstavljen s tabelo ali pa kot veriga členov, povezanih s kazalci (*linked list*).

### 4. Uniforme

Pomagamo si lahko s tabelo, ki za vsako kombinacijo velikosti (od 1 do 100) in kosa (od 1 do 3) hrani število prejetih kosov te velikosti. V spodnjem programu imamo v ta namen tabelo zaloga; na začetku postavimo vse njene elemente na 0, nato pa beremo vhodne podatke in po vsaki prebrani vrstici povečamo ustrezni element tabele za 1. (Paziti moramo še na to, da se indeksi v tabelo štejejo od 0 naprej, v vhodnih podatkih pa so števila od 1 naprej.)

Nato lahko za vsako velikost določimo število popolnih uniform, ki jih lahko sestavimo pri tej velikosti. Pogledati moramo, katerega od teh kosov imamo (v tej velikosti) na razpolago v najmanj izvodih; toliko popolnih uniform lahko sestavimo (saj imamo tudi ostala dva kosa v vsaj toliko izvodih), več pa ne (saj bi nam tega kosa za nekatere uniforme zmanjkalo). Ko za neko velikost poznamo število popolnih uniform te velikosti, lahko to število prištejemo spremenljivki rezultat, v kateri se bo tako na koncu nabralo skupno število popolnih uniform, po katerem sprašuje naloga.

```
#include <stdio.h>
#define MaxVelikost 100
#define StKosov 3

int main()
{
  int zaloga[MaxVelikost][StKosov], n, vel, kos, naj, rezultat;

  /* Inicializirajmo tabelo zaloga. */
  for (vel = 0; vel < MaxVelikost; vel++) for (kos = 0; kos < StKosov; kos++)
    zaloga[vel][kos] = 0;

  /* Preberimo vhodne podatke. */
  scanf("%d", &n);
  while (n-- > 0) {
    scanf("%d %d", &kos, &vel);
    zaloga[vel - 1][kos - 1]++; }

  /* Izračunajmo rezultat. */
  for (rezultat = 0, vel = 0; vel < MaxVelikost; vel++) {
    /* Koliko popolnih uniform velikosti vel lahko sestavimo? */
    naj = zaloga[vel][0];
    for (kos = 1; kos < StKosov; kos++)
      if (zaloga[vel][kos] < naj)
        naj = zaloga[vel][kos];
    rezultat += naj; }

  printf("%d\n", rezultat); return 0;
}
```

Gornja rešitev poišče minimum (pri posamezni velikosti) z zanko po kosih; ker pa je vnaprej znano, da so kosi le trije, bi lahko minimum poiskali tudi z dvema pogojnima stavkoma:

```
    naj = zaloga[vel][0];
    if (zaloga[vel][1] < naj) naj = zaloga[vel][1];
    if (zaloga[vel][2] < naj) naj = zaloga[vel][2];
```

### 5. Davek na ograjo

Z dvema gnezdenima zankama (po $x$ in po $y$) se sprehodimo po vseh kvadratkih naše mreže. Pri vsakem si zapomnimo njegovega lastnika (v spremenljivki lastnik), nato pa (s še eno vgnezdeno zanko) preglejmo njegove štiri sosede. Sosedje kvadratka $(x, y)$ imajo koordinate $(x \pm 1, y)$ in $(x, y \pm 1)$; spodnji program jih računa s pomočjo tabel dx in dy. Če ima sosednji kvadratek drugega lastnika kot naš opazovani $(x, y)$, povečajmo dolžino ograj našega lastnika za 1 (v tabeli dolzina, v kateri smo na začetku vse elemente inicializirali na 0). Paziti moramo še na možnost, da sosednji kvadratek leži zunaj mreže; spodnji program v tem primeru postavi sosed = −1, kar bo gotovo različno od lastnika trenutnega kvadratka $(x, y)$, tako da bo program pravilno preštel tudi ograje na zunanjem robu mreže.

Na koncu tega prehoda čez celo mrežo imamo v tabeli dolzina za vsakega lastnika skupno dolžino ograj okoli njegovih kvadratkov in te rezultate moramo le še izpisati.

```
#include <stdio.h>
#define MaxW 250
#define MaxH 250
#define MaxLastnikov (MaxW * MaxH)

int main()
{
  const int dx[] = { -1, 1, 0, 0 }, dy[] = { 0, 0, -1, 1 };
  int dolzina[MaxLastnikov], x, y, xx, yy, smer, lastnik, sosed;
  int h = Visina(), w = Sirina(), n = StLastnikov();

  /* Inicializirajmo tabelo dolžina. */
  for (lastnik = 0; lastnik < n; lastnik++) dolzina[lastnik] = 0;

  /* Preglejmo celo mrežo. */
  for (x = 0; x < w; x++) for (y = 0; y < h; y++) {
    lastnik = Lastnik(x, y);

    /* Preglejmo sosede polja (x, y). */
    for (smer = 0; smer < 4; smer++) {
      xx = x + dx[smer]; yy = y + dy[smer]; /* sosednje polje */

      /* Kdo je lastnik polja (xx, yy)? */
      if (xx < 0 || xx >= w || yy < 0 || yy >= h) sosed = -1;
      else sosed = Lastnik(xx, yy);

      /* Povečajmo števec ograj, če sta lastnika različna. */
      if (lastnik != sosed) dolzina[lastnik]++; }}

  /* Izpišimo rezultate. */
  for (lastnik = 0; lastnik < n; lastnik++)
    printf("Dolžina ograj lastnika %d je %d.\n", lastnik, dolzina[lastnik]);
  return 0;
}
```

## REŠITVE NALOG ZA DRUGO SKUPINO

### 1. Vnos šifre

Naloga je zelo primerna za reševanje z rekurzijo. Če šifro gradimo postopoma in dodajamo števke na konec šifre, imamo na vsakem koraku več možnosti, kako nadaljevati: za naslednjo števko lahko vzamemo katerokoli sosedo prejšnje števke (vključno s prešnjo števko samo). Za vsako od teh možnih nadaljevanj izvedemo rekurziven klic, ki na podoben način zgenerira še preostanek šifre. Robni primer rekurzije nastopi takrat, ko je šifra že dolga $n$ znakov; takrat jo moramo le še izpisati. V spodnji rešitvi za te stvari skrbi podprogram Rekurzija, ki dobi tri parametre: tabelo sifra, v kateri pripravljamo šifro (in jo na koncu izpišemo); zahtevano končno dolžino šifre n; in pa trenutno globino rekurzije i — ta parameter nam pove, da so v nizu sifra že vpisani znaki na indeksih od 0 do i − 1, ne pa še tisti na indeksih od i do n − 1.

Vprašanje je še, kako vemo, katere so možne sosede prejšnje števke. Spodnja rešitev ima za to kar tabelo (sosede), v kateri je za vsako števko od 0 do 9 naveden niz z vsemi njenimi sosednjimi števkami.

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

const char *sosede[] = { "08", "124", "1235", "236", "1457",
                         "24568", "3569", "478", "57890", "689" };

void Rekurzija(char *sifra, int i, int n)
{
  const char *nasl;
  /* Če je šifra zdaj že dolga n znakov, jo le še izpišemo. */
  if (i >= n) printf("%s\n", sifra);
  /* Sicer pa na vse možne načine izberimo naslednji znak in nadaljujmo z rekurzijo. */
  else for (nasl = sosede[sifra[i - 1] - '0']; *nasl; nasl++) {
    sifra[i] = *nasl;
    Rekurzija(sifra, i + 1, n); }
}
```

Glavni podprogram NastejSifre zdaj nima veliko dela: pripraviti mora tabelo sifra in za vsako možno začetno števko (od 0 do 9) sprožiti rekurzivni klic, ki bo izpisal vse šifre, ki se začnejo na to števko.

```
void NastejSifre(int n)
{
  int d;
  char *sifra = (char *) malloc(n + 1);
  sifra[n] = 0; /* Postavimo znak za konec niza. */
  for (d = 0; d < 10; d++) { /* Prva števka je lahko katera koli. */
    sifra[0] = '0' + d;
    Rekurzija(sifra, 1, n); }
  free(sifra);
}
```

### 2. Prenova ceste

Načeloma nas za vsako točko (recimo $t$) zanima, katerih $k$ prebivalcev ji leži najbližje (in za katero podjetje bi ti prebivalci glasovali). Ta soseščina ima naslednjo lepo lastnost: če sta $i$ in $j$ (za $i < j$) med najbližjimi $k$ sosedi naše točke $t$, potem so tudi vsi vmesni prebivalci $(i+1, i+2, \ldots, j-1)$ med najbližjimi $k$ sosedi točke $t$.[2] To pomeni, da skupina $k$ najbližjih sosedov točke $t$ gotovo tvori neko strnjeno podzaporedje (oblike $z, z+1, \ldots, z+k-1$). Na primer, če leži $t$ levo od vseh prebivalcev, so njegovi najbližji sosedje kar najbolj levi prebivalci in imamo $z = 1$; podobno, če leži $t$ desno od vseh prebivalcev, so najbližji sosedje kar najbolj desni prebivalci in imamo $z = n - k + 1$ (če je $n$ število vseh prebivalcev).

Tako torej vidimo, da čeprav je možnih točk $t$ neskončno mnogo (saj je premica zvezna), je možnih različnih soseščin le $n - k + 1$ (ker so odvisne le od tega, pri katerem prebivalcu $z$ se začnejo).

Kako se spreminja $z$ (torej najbolj levi med najbližjimi $k$ sosedi), če se s točko $t$ počasi premikamo od leve proti desni? Na začetku so naši najbližji sosedje prebivalci od 1 do $k$. Do prve spremembe pride, ko prebivalec 1 izpade iz te soseščine in vanjo pride prebivalec $k + 1$ (najbolj levi prebivalec v soseščini ima zdaj številko 2, torej imamo $z = 2$). To se očitno zgodi takrat, ko nam postane $k + 1$ bližje kot 1; torej takrat, ko pridemo ravno na pol poti med tema dvema prebivalcema, pri $t = (x_1 + x_{k+1})/2$. Naslednja sprememba bo nastopila pri $t = (x_2 + x_{k+2})/2$, ko bo iz soseščine izpadel

[2] O tem se lahko prepričamo takole. Če $i$ in $j$ ležita levo od $t$, potem so vsi vmesni prebivalci točki $t$ še bližje kot prebivalec $i$; torej, če je $i$ med najbližjimi $k$ sosedi, so ti vmesni prebivalci tudi. Podobno razmišljamo, če $i$ in $j$ ležita desno od $t$. Ostane še možnost, da $i$ leži levo od $t$, prebivalec $j$ pa desno od $t$; v tem primeru so med sosedi $i+1, \ldots, j-1$ tisti, ki ležijo levo od $t$, gotovo bližje $t$-ju kot sosed $i$, in tisti, ki ležijo desno od $t$, so mu gotovo bližje kot sosed $j$; ker sta $i$ in $j$ oba med najbližjimi $k$ sosedi, so torej ti vmesni prebivalci tudi.

prebivalec 2, vanjo pa bo prišel prebivalec $k+2$. Tako lahko nadaljujemo vse do konca seznama. Sproti lahko vzdržujemo tudi števec, ki pove, koliko izmed najbližjih $k$ sosedov podpira prvo podjetje; tega števca ni težko popraviti, ko prebivalci prihajajo v soseščino in izpadajo iz nje.

Dobljeni postopek je dovolj preprost, da ga lahko zapišemo kar kot podprogram v C-ju. Poleg tabel $x$ in $p$ pričakuje kot parametre še skupno število prebivalcev $n$, velikost soseščine $k$ in skupno dolžino opazovanega odseka ceste $D$. (V besedilu naloge je $n$ fiksiran na $10^6$, dolžina $D$ pa na 100 km.)

```
#include <stdio.h>

void PrenovaCeste(int n, int k, const double x[], const int p[], double D)
{
  double xOd, xDo, dolzina[2] = { 0, 0 };
  int k1 = 0, z;
  for (z = 0; z < k; z++) if (p[z]) k1++;
  for (z = 0; z + k <= n; z++) {
    /* Na katerem intervalu x-koordinat so najbližji sosedje ravno
       prebivalci z, z + 1, ..., z + k - 1? */
    xOd = (z > 0) ? (x[z - 1] + x[z - 1 + k]) / 2 : 0;
    xDo = (z + k < n) ? (x[z] + x[z + k]) / 2 : D;
    /* Med temi k prebivalci jih k1 podpira podjetje 1, ostali pa podjetje 0. */
    dolzina[k1 >= k - k1 ? 1 : 0] += xDo - xOd;
    /* Popravimo števec k1, da bo pripravljen za naslednjo iteracijo. */
    if (z + k < n) k1 = k1 - p[z] + p[z + k]; }
  printf("Kangaroads popravi %g km, Wallabyway pa %g km.\n",
    dolzina[0], dolzina[1]);
}
```

### 3. Skrivno sporočilo

V zanki pregledujemo istoležne znake prvotnega niza p1 in njegove šifrirane različice c1. Pri tem si v tabelo sifra zapisujemo šifre posameznih znakov; pri tistih znakih, za katere šifre še ne poznamo, pa imejmo sifra[c] = '*'. Podobno imamo tudi tabelo original, v kateri za vsak znak piše, kateri znak originalnega besedila se zašifrira vanj; če za nek znak še ne vemo, kaj se zašifrira vanj, pa imamo original[c] = '*'.

Na vsakem koraku preverimo naslednje: če za trenutni znak niza p1 šifre še ne poznamo, jo lahko zdaj odčitamo iz trenutnega znaka niza c1 (če le-ta ni zvezdica); če pa za trenutni znak niza p1 šifro že poznamo, lahko preverimo, če je v c1 zdaj ista šifra. Če tu opazimo neujemanje, izpišemo obvestilo o napaki. Podobno, če za trenutni znak niza c1 še ne poznamo originala, ga lahko zdaj odčitamo iz trenutnega znaka niza p1; če pa nek original za trenutni znak niza c1 že poznamo, moramo preveriti, če se ujema s trenutnim znakom niza p1; če opazimo neujemanje, izpišemo obvestilo o napaki.

Poseben primer nastopi, če lahko na ta način ugotovimo šifre vseh črk abecede razen ene (za lažje preverjanje tega pogoja imamo spremenljivko stNeznanih, v kateri štejemo, za koliko znakov še ne poznamo šifre). V tem primeru lahko sklepamo tudi na šifro tiste ene preostale črke: njena šifra je edina črka, ki še ni šifra nobene druge črke (to je tista, pri kateri je pripadajoči element tabele original še vedno enak '*'). (Ta sklep je upravičen zato, ker besedilo naloge zagotavlja, da je kodirna funkcija bijektivna.)

Nazadnje se le še zapeljemo po nizu p2, kodiramo znake s pomočjo tabele sifra in jih izpisujemo.

```
#include <stdio.h>

void Desifriraj(const char *p1, const char *c1, const char *p2)
{
  char sifra[26], original[26]; int c, neznana, stNeznanih = 26;
  for (c = 0; c < 26; c++) sifra[c] = '*', original[c] = '*';
  for ( ; *p1; ++p1, ++c1) {
```

```
      /* Znaki, ki v c1 niso šifrirani, nam nič ne pomagajo. */
      if (*c1 == '*') continue;

      /* Mogoče smo zdaj šele prvič izvedeli, v kaj se šifrira trenutni znak niza p1
         in kaj se šifrira v trenutni znak niza c1. */
      if (sifra[*p1 - 'a'] == '*') sifra[*p1 - 'a'] = *c1, stNeznanih--;
      if (original[*c1 - 'a'] == '*') original[*c1 - 'a'] = *p1;

      /* Mogoče pa smo za ta znak nekoč prej videli že neko drugo šifro ali za to
         šifro nek drug originalni znak; to pomeni, da je v vhodnih podatkih napaka. */
      if (sifra[*p1 - 'a'] != *c1 || original[*c1 - 'a'] != *p1) {
        printf("neveljavno sporočilo\n"); return; }}

    /* Če smo ugotovili šifre vseh črk abecede razen ene, potem tudi za tisto eno
       lahko sklepamo, v kaj se zašifrira. */
    if (stNeznanih == 1) {

      /* Poglejmo, za katero črko še ne poznamo šifre. */
      for (c = 0; c < 26; c++)
        if (sifra[c] == '*') { neznana = c; break; }

      /* V tabeli „original" je zdaj ena sama črka, za katero še ne poznamo šifre,
         in ta mora biti šifra črke „neznana". */
      for (c = 0; c < 26; c++)
        if (original[c] == '*') { sifra[neznana] = c + 'a'; break; }}

    /* Izpišimo šifrirano obliko niza p2. */
    while (*p2) putchar(sifra[*p2++ - 'a']);
    putchar('\n');
}
```

### 4. Potenciranje

Pomagajmo si z namigom iz besedila naloge. Števil $a^2$, $a^4$, $a^8$ in tako naprej ni težko računati z zaporednim kvadriranjem:

$$a^2 = a \cdot a, \quad a^4 = (a^2) \cdot (a^2), \quad a^8 = (a^4) \cdot (a^4)$$

in tako naprej. Kako pa s temi števili pridemo do poljubne potence $a^b$? Oglejmo si konkreten primer; recimo, da nas zanima $b = 87$. Število $b$ lahko izrazimo kot vsoto nekaj potenc števila 2; v našem primeru je $87 = 64+16+4+2+1 = 2^6+2^4+2^2+2^1+2^0$. (To je pravzaprav isti razmislek, ki ga opravimo pri pretvorbi v dvojiški zapis: na primer, dvojiški zapis števila 87 je 1010111 — enice so ravno pri tistih potencah števila 2, ki jih moramo sešteti, da dobimo 87.) Spomnimo se, kaj velja za produkt vsote: $a^{b+c} = a^b \cdot a^c$. V našem primeru to pomeni, da je

$$a^{87} = a^{64+16+4+2+1} = a^{64} \cdot a^{16} \cdot a^4 \cdot a^2 \cdot a^1.$$

Tako torej vidimo, da lahko $a^{87}$ izračunamo tako, da najprej z zaporednim kvadriranjem izračunamo $a^2, a^4, a^8, a^{16}, a^{32}, a^{64}$ in potem nekatere od teh potenc zmnožimo med sabo.

Zapišimo zdaj ta postopek za splošen $b$:

$c := 1;$
**for** $i := 0, 1, 2, \ldots:$
  **if** je bit $i$ v dvojiškem zapisu števila $b$ prižgan **then**
    $c := c \cdot a^{2^i}$;

Kako lahko z operacijami, ki jih imamo na voljo pri naši nalogi, čim preprosteje preverjamo, ali je bit $i$ v številu $b$ prižgan ali ne? Na primer, če izračunamo $b \bmod 2$, torej ostanek po deljenju $b$ z 2, nam rezultat pove, ali je v $b$-ju prižgan bit 0 (najnižji bit). Celi del količnika po deljenju $b$ z 2 pa je pravzaprav število, ki ga dobimo, če $b$-ju najnižji bit odrežemo. V naši zanki lahko torej po vsaki iteraciji delimo $b$ z 2, tako da nam bo po $i$ iteracijah v bitu 0 pristal tisti bit, ki je bil na začetku na $i$-tem mestu. Zanka se ustavi, ko $b$ pade na 0, saj takrat vemo, da so vsi višje ležeči biti $b$-ja ugasnjeni. Podobno tudi potence $a^{2^i}$ računamo sproti preprosto tako, po vsaki iteraciji število $a$ kvadriramo (pomnožimo s samim sabo). Tako smo prišli do naslednjega postopka:

```
c := 1;
while b > 0:
  if b mod 2 = 1 then
    c := c · a;
  a := a · a;
  b := ⌊b/2⌋; (* celi del količnika po deljenju z 2 *)
```

Tega postopka ni težko zapisati v jeziku, ki ga zahteva besedilo naloge. Namesto stavkov **while** in **if**, ki ju ta jezik nima, si bomo morali pomagati s pogojnimi skoki (ukaz JL):

```
                        Razlaga (ni del programa)
        SUB c, c        postavi c na 0
        ADD c, 1        c je zdaj 1
zanka:  JL b, 1, konec  če je b = 0, končaj
        SUB t, t        postavi t na 0
        ADD t, b        postavi t na b
        MOD t, 2        trenutni bit števila b
        JL t, 1, preskok če je ugasnjen, preskoči naslednji ukaz
        MUL c, a        pomnoži c s trenutno potenco števila a
preskok: MUL a, a       pripravi naslednjo potenco števila a
        DIV b, 2        zamakni b za en bit navzdol
        JL 0, 1, zanka  skoči nazaj na začetek zanke
konec:
```

**5. Vezja**

Primeri, ko se dve povezavi sekata, nam določajo omejitve: takšni povezavi ne smeta biti obe na isti strani plošče; če je ena od njiju na sprednji strani plošče, mora biti druga na zadnji strani in obratno. Lahko torej začnemo pri poljubni povezavi in si izberemo, na kateri strani plošče bi bila; iz tega potem enolično sledi, kje morajo biti tiste povezave, ki se sekajo z našo: biti morajo pač na nasprotni strani plošče; in zato morajo biti tiste, ki se sekajo z *njimi*, spet na prvi strani plošče; in tako naprej. Pri tem si lahko pomagamo z vrsto, v katero dodajamo povezave, ki smo jim že določili stran, nismo pa še pregledali, kaj vse se seka z njimi (in mora biti zaradi tega na nasprotni strani).

Ko tako sledimo omejitvam, bomo prej ali slej bodisi narisali vse povezave bodisi ugotovili, da nas omejitve pripeljejo v protislovje (ker na primer neko povezavo narišemo na eno stran plošče, nato pa ugotovimo, da smo neko drugo povezavo, ki se seka z njo, že prej narisali na isto stran plošče).

Nalogo si lahko predstavljamo tudi kot problem na grafih. Sestavimo graf, ki ima po eno točko za vsako povezavo našega vezja; povezava med dvema točkama pa naj v našem grafu obstaja takrat, ko se pripadajoči povezavi vezja med seboj sekata (in torej ne smeta biti obe na isti strani plošče). Naša naloga ni zdaj nič drugega kot problem barvanja tega grafa z dvema barvama (vsaka barva predstavlja eno od strani plošče) in z običajno omejitvijo, da krajišči povezave ne smeta biti iste barve.

```
naj bo m število povezav in (z_i, k_i) naj bosta luknjici, ki ju povezuje i-ta povezava;
for i := 1 to m do barva[i] := 0;
for i := 1 to m do
  if barva[i] > 0 then continue; (* Točka i je pobarvana že od prej. *)
  barva[i] := 0;
  (* Zdaj smo točki i določili barvo; iz tega pa enolično sledijo tudi
     barve njenih sosed, pa njihovih sosed in tako naprej. *)
  naj bo Q prazna vrsta; dodaj i v Q;
  while Q ni prazna:
    vzemi poljubno u iz vrste Q;
    for v := 1 to m:
      if not SeSekata(x[z_u], y[z_u], x[k_u], y[k_u], x[z_v], y[z_v], x[k_v], y[k_v]) then continue;
      if barva[v] = barva[u] then return false; (* grafa ni mogoče pobarvati *)
      if barva[v] > 0 then continue; (* v je že primerno pobarvan *)
      barva[v] := 3 − barva[u]; dodaj v v Q;
```

**return true**;

Če nam postopek na koncu vrne **true**, lahko s pomočjo tabele *barva* vezje tudi narišemo; tiste povezave, ki imajo $barva[i] = 1$, narišemo na eno stran plošče, ostale pa na drugo stran plošče.

Za namene našega tekmovanja je gornji postopek čisto dovolj dober, dalo pa bi se ga še malo izboljšati: v notranji zanki gremo zdaj z $v$ po vseh povezavah in za vsako preverjamo, ali se seka s povezavo $u$; ker moramo to storiti za vsako povezavo $u$, je časovna zahtevnost tega postopka $O(m^2)$. Obstajajo algoritmi, ki vse pare sekajočih se daljic poiščejo hitreje (če takih parov ni veliko), na primer Bentley-Ottmannov algoritem, ki porabi $O((m + s) \log m)$ časa, če je $s$ število parov sekajočih se daljic.

## REŠITVE NALOG ZA TRETJO SKUPINO

### 1. Ljudožerci na premici

Koordinate ljudožercev bomo hranili v naraščajočem vrstnem redu. (Ker v vhodni datoteki niso urejeni, jih bomo na začetku pred nadaljnjo obdelavo uredili naraščajoče.) Ko pride nov padalec na koordinato $a_j$, lahko v našem urejenem seznamu ljudožercev z bisekcijo poiščemo najbližjega ljudožerca. Naloga pravi, da če na isti točki stoji več ljudožercev, bo šel k našemu padalcu tisti, ki ima najnižjo zaporedno številko, vendar je ta omejitev čisto nebistvena, saj bomo morali na koncu ljudožerce izpisati urejene po koordinatah, ne po zaporedni številki. Zato lahko v primeru, ko recimo $a_j$ pade med $x[i]$ in $x[i + 1]$, brez slabe vesti k padalcu pošljemo enega od teh dveh ljudožercev (tistega, ki je bližje; če sta oba enako daleč od $a_j$, pošljemo levega) in se ne ukvarjamo s tem, ali ima mogoče še več prejšnjih ali naslednjih ljudožercev isto koordinato. Tako moramo po vsakem padalcu popraviti le en element našega seznama koordinat ljudožercev in ta seznam po takem popravku tudi ostane urejen.

Ker bisekcija na tabeli $n$ elementov porabi $O(\log n)$ časa, je časovna zahtevnost celotnega postopka $O(n \log n)$.

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

#define MaxN 100000
int n, x[MaxN];

int PoisciNajblizjega(int a)
{
  int l, d, m;
  if (a <= x[0]) return 0; /* Poseben primer, če je padalec levo od vseh ljudožercev. */
  l = 0; d = n;
  while (d - l > 1)
  {
    /* Na tem mestu velja x[l] < a ≤ x[d]. (Pri d = n si mislimo x[d] = ∞.) */
    m = (l + d) / 2;
    if (x[m] < a) l = m; else d = m;
  }
  /* Tu velja x[d - 1] < a ≤ x[d].
     Poglejmo, kateri od ljudožercev d - 1 in d je bližji padalcu a. */
  return d < n && x[d] - a < a - x[d - 1] ? d : d - 1;
}

/* Primerjalna funkcija za urejanje ljudožercev po koordinati. */
int Primerjaj(const void *a, const void *b) {
  return *(const int *) a - *(const int *) b; }

int main()
{
  int i, j, m, a;
  FILE *f = fopen("ljudozerci.in", "rt");
  fscanf(f, "%d %d", &n, &m);
```

```
  /* Preberimo koordinate ljudožercev in jih uredimo naraščajoče. */
  for (i = 0; i < n; i++) fscanf(f, "%d", &x[i]);
  qsort(x, n, sizeof(x[0]), &Primerjaj);

  /* Obdelajmo padalce. */
  for (j = 0; j < m; j++)
  {
    fscanf(f, "%d", &a); /* Preberimo naslednjega padalca. */
    i = PoisciNajblizjega(a); /* Poiščimo najbližjega ljudožerca... */
    x[i] = a; /* ... in ga postavimo na mesto, kjer je pristal padalec. */
  }
  fclose(f);

  /* Izpišimo končni položaj ljudožercev. */
  f = fopen("ljudozerci.out", "wt");
  for (i = 0; i < n; i++) fprintf(f, "%d\n", x[i]);
  fclose(f); return 0;
}
```

### 2. Po Indiji z avtobusom

Ko razmišljamo, na kateri avtobus stopiti, se splača izbrati (med vsemi, ki peljejo mimo naše trenutne točke) tistega, ki pelje najdlje (torej čigar končna postaja leži najbolj desno). O tem se lahko prepričamo takole: recimo, da smo v točki $s$ in da ima avtobus, izbran na opisani način, končno postajo $t$; in recimo, da boljšo rešitev dobimo, če ne stopimo na ta avtobus, pač pa na nekega drugega, ki ima končno postajo $u < t$. V tisti optimalni rešitvi prej ali slej sestopimo s tega novega avtobusa — očitno nekje na območju $(s, u]$, saj dlje ta avtobus ne pelje. Toda to območje v celoti prevozi tudi avtobus, ki pelje do $t$, torej bi se lahko peljali tudi z njim in še vedno nadaljevali pot enako, kot bi jo pri naši optimalni rešitvi. Tako torej vidimo, da optimalne rešitve gotovo ne bomo spregledali, če bomo pri vsakem koraku stopili na tisti avtobus, ki nas pripelje najdlje. S podobnim razmislekom se lahko prepričamo, da ni pametno sestopiti z avtobusa prej kot na njegovi končni postaji (razen seveda če nas ta avtobus že prej pripelje do našega cilja $y$).

Tako smo dobili naslednji požrešni algoritem:

naj bo $x$ naš začetni položaj, $y$ pa naš cilj;
$štVoženj := 0$;
**while** $x < y$:
  med vsemi avtobusi, ki peljejo mimo točke $x$, izberi tistega z
    najbolj desno končno postajo; recimo mu $i$;
  $x := b_i$; $štVoženj := štVoženj + 1$;

Razmislimo, kako lahko na vsakem koraku učinkovito poiščemo pravi avtobus. Zanimajo nas le tisti, ki se peljejo mimo točke $x$, torej za katere je $a_i \le x \le b_i$. Drugi del tega pogoja, torej $x \le b_i$, je pravzaprav odveč; med vsemi avtobusi z $a_i \le x$ bomo tako ali tako izbrali tistega z največjim $b_i$; če niti ta ne leži desno od $x$, potem sploh noben avtobus ne pelje dlje kot do $x$, torej je problem nerešljiv (naloga pa zagotavlja, da se to ne bo zgodilo). Tako nam ostane pravzaprav pogoj, da med vsemi avtobusi, ki imajo $a_i \le x$, izberemo tistega z največjim $b_i$. V vsaki iteraciji glavne zanke se $x$ malo poveča, zato pogoju $a_i \le x$ ustreza vse več avtobusov (in je tudi $b_i$ lahko vse večji). Vidimo torej, da je koristno avtobuse urediti naraščajoče po $a_i$; tako bomo lahko po vsakem premiku nadaljevali s pregledovanjem tam, kjer smo prej končali (in bomo na novo pregledali le tiste avtobuse, na katere zdaj lahko stopimo, pred zadnjim premikom pa še nismo mogli).

uredi avtobuse naraščajoče po $a_i$;
$štVoženj := 0$; $i := 1$;
**while** $x < y$:
  $b := x$;
  **while** $i \le n$ **and** $a_i \le x$:
    **if** $b_i > b$ **then** $b := b_i$;
    $i := i + 1$;

$x := b$; $štVoženj := štVoženj + 1$;

Ta postopek je prijetno učinkovit, saj ima notranja zanka vsega skupaj (po vseh iteracijah zunanje zanke) le $O(n)$ iteracij, za vsak avtobus po eno. Največ časa, $O(n \log n)$, tako porabimo za urejanje avtobusov na začetku postopka. Zapišimo našo rešitev še v C-ju:

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

#define MaxN 200000

typedef struct { int z, k; } Proga;
Proga proge[MaxN];

/* Primerjalna funkcija za urejanje avtobusov po začetni postaji. */
int Primerjaj(const void *a, const void *b) {
  return ((const Proga *) a)->z - ((const Proga *) b)->z; }

int main()
{
  int i, n, x, y, stVozenj, doKod;
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("avtobus.in", "rt");
  fscanf(f, "%d %d %d", &n, &x, &y);
  for (i = 0; i < n; i++) fscanf(f, "%d %d", &proge[i].z, &proge[i].k);
  fclose(f);
  /* Uredimo avtobuse po začetni postaji. */
  qsort(proge, n, sizeof(proge[0]), &Primerjaj);
  stVozenj = 0; i = 0;
  while (x < y)
  {
    /* Trenutno se nahajamo na postaji x. Med avtobusi, ki vozijo mimo nje,
       poiščimo tistega z najbolj desno končno postajo. */
    doKod = -1;
    for (doKod = -1; i < n && proge[i].z <= x; i++)
      if (proge[i].k > doKod) doKod = proge[i].k;
    stVozenj++; x = doKod;
  }
  /* Izpišimo rezultate. */
  f = fopen("avtobus.out", "wt"); fprintf(f, "%d\n", stVozenj); fclose(f); return 0;
}
```

### 3. Luči

Za vsako stikalo imamo načeloma dve možnosti: lahko ga pritisnemo enkrat ali pa nobenkrat. Vendar pa stikala med seboj niso neodvisna; če si izberemo stanje nekega stikala $u$ (torej ali je pritisnjeno ali ne), potegne to za sabo posledice za luči, ki so povezane na $u$, in s tem tudi za druga stikala, ki so povezana z istimi lučmi kot $u$.

Na primer, če je neka luč $l$ prižgana in je povezana samo s stikalom $u$ (in nobenim drugim), potem $u$ *moramo* pritisniti; če pa je taka luč ugasnjena, potem $u$ *ne smemo* pritisniti. Podobno, če je luč $l$ prižgana in povezana s stikaloma $u$ in $v$, potem moramo pritisniti natanko eno od stikal $u$ in $v$, drugega pa ne; če pa je taka luč ugasnjena, potem moramo pritisniti bodisi obe stikali ali pa nobeno od njiju.

Začnemo lahko pri poljubnem stikalu $u$ in se na primer vprašamo, kaj se zgodi, če to stikalo pritisnemo. Oglejmo si luči, ki so povezane z $u$; s pomočjo omejitev iz prejšnjega odstavka lahko zdaj določimo, v kakšnem stanju morajo biti druga stikala, ki so povezana s temi lučmi; in ker smo zdaj določili stanje teh stikal, lahko v nadaljevanju pregledamo še njihove ostale luči in tako naprej. Pri tem pregledovanju se lahko zgodi, da pridemo v protislovje (npr. za neko stikalo nam ena luč zahteva, da mora biti pritisnjeno, neka druga luč pa, da ne sme biti biti pritisnjeno); tedaj vemo, da naša začetna odločitev, da pritisnemo stikalo $u$, ni sprejemljiva. Če pa se ta postopek ustavi (ko pregleda vse luči

in stikala, dosegljiva iz $u$), ne da bi prišel v protislovje, to pomeni, da so kombinacije, pri katerih je $u$ pritisnjeno, načeloma možne; imajo pa potem vsa druga stikala, ki smo jih dosegli iz $u$, že enolično določeno stanje in si pri njih ne moremo več izbirati, kaj bi naredili z njimi.

V nadaljevanju lahko podoben razmislek ponovimo še za možnost, da $u$-ja ne pritisnemo. Tako ugotovimo, koliko stanj $u$-ja je možnih (lahko sta obe, eno ali pa celo nobeno od njiju).

Videli smo, da je stanje vseh stikal, ki so dosegljiva iz $u$, enolično določeno, čim si izberemo stanje $u$-ja. Pač pa je mogoče, da obstajajo še kakšna druga stikala, ki iz $u$-ja niso dosegljiva; tista pa so neodvisna od $u$ in lahko zdaj isti razmislek ponovimo še pri njih. Na koncu moramo rezultate za takšna neodvisna stikala pomnožiti med sabo (če imamo dve možnosti za stanje stikala $u$, pa dve možnosti za $u'$ in dve možnosti za neko še tretje neodvisno stikalo $u''$, nam dá to vsega skupaj $2 \times 2 \times 2$ možnih kombinacij stanj stikal).

Nalogo si lahko predstavljamo kot problem na grafih; stikala in luči tvorijo točke našega grafa. Gornji razmislek ne pomeni nič drugega kot to, da za vsako povezano komponento tega grafa posebej ugotovimo, ali je možnih stanj te komponente 0, 1 ali 2, in te rezultate potem pomnožimo med sabo.

Poseben primer so še luči, ki niso povezane z nobenim stikalom. Če je taka luč na začetku ugasnjena, nas ne moti; če pa je na začetku prižgana, je ne bomo mogli ugasniti z nobeno kombinacijo stikal, zato bo moral naš postopek vrniti 0.

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

#define MaxL 300
#define MaxS 300
#define M 1000000007

int L, S;
/* Podatki o stikalih: številke luči, s katerimi je povezano stikalo s,
   so v tabeli neighS na indeksih od firstS[s] do firstS[s] + degS[s] − 1]. */
int firstS[MaxS], degS[MaxS], neighS[MaxL * 2];
/* Podatki o lučeh; številke stikal, s katerimi je povezana luč l,
   so v tabeli stikala[l] na indeksih od 0 do degL[l] − 1. */
int prizgana[MaxL], degL[MaxL], stikala[MaxL][2];
/* Naslednje tabele uporabljamo med pregledovanjem grafa.
   stanje[s] pove, ali bi stikalo s pritisnili ali ne (0 ali 1);
   če pa mu stanja še nismo določili, imamo stanje[s] = −1. */
int stanje[MaxS], vrsta[MaxS], glava, rep;
bool obdelano[MaxS];

/* Naslednji podprogram preveri, če smemo stikalo u0 postaviti v stanje s. */
bool Preizkusi(int u0, int s)
{
  int u, v, i, luc;

  /* Postavimo u0 v stanje s in ga dodajmo v vrsto. */
  stanje[u0] = s; obdelano[u0] = true;
  glava = 0; rep = 1; vrsta[glava] = u0;

  /* Preglejmo vse, kar je dosegljivo iz tega stikala. */
  while (glava < rep)
  {
    u = vrsta[glava++]; /* Vzemimo naslednje stikalo iz vrste. */
    /* Stikalu u smo že določili stanje; kaj to pomeni za luči, s katerimi je povezano? */
    for (i = 0; i < degS[u]; i++)
    {
      luc = neighS[firstS[u] + i];

      if (degL[luc] == 1) {
        /* Ta luč je priklopljena samo na u. Preverimo, če bo na koncu ugasnjena. */
        if (stanje[u] ^ prizgana[luc]) return false; }

      else
      {
```

```
            v = stikala[luc][stikala[luc][0] == u ? 1 : 0];
            /* Ta luč je priklopljena na u in še na neko drugo stikalo v.
               Če v-ju še nismo določili stanja, mu ga lahko določimo zdaj. */
            if (stanje[v] < 0)
              stanje[v] = prizgana[luc] ^ stanje[u],
              vrsta[rep++] = v, obdelano[v] = true;
            /* Če pa ima v stanje že od prej, lahko preverimo, če bo luč na koncu res ugasnjena. */
            else if (stanje[v] ^ stanje[u] ^ prizgana[luc]) return false;
          }
        }
      }
      return true;
}

int main()
{
      FILE *fi = fopen("luci.in", "rt");
      FILE *fo = fopen("luci.out", "wt");
      int T, luc, i, stikalo, nMoznosti, nPovezav, rezultat;
      fscanf(fi, "%d", &T);
      while (T-- > 0)
      {
        /* Preberimo naslednji testni primer. */
        fscanf(fi, "%d %d", &L, &S);
        nPovezav = 0;
        for (luc = 0; luc < L; luc++) {
          fscanf(fi, "%d", &prizgana[luc]); degL[luc] = 0; }
        for (stikalo = 0; stikalo < S; stikalo++) {
          fscanf(fi, "%d", &degS[stikalo]);
          stanje[stikalo] = -1; obdelano[stikalo] = false;
          for (i = 0, firstS[stikalo] = nPovezav; i < degS[stikalo]; i++) {
            fscanf(fi, "%d", &luc); luc--;
            neighS[nPovezav++] = luc; stikala[luc][degL[luc]++] = stikalo; }}
        rezultat = 1;
        /* Preverimo, če je kakšna luč prižgana in ni povezana z nobenim stikalom. */
        for (luc = 0; luc < L; luc++)
          if (prizgana[luc] && degL[luc] == 0) { rezultat = 0; break; }
        /* Preglejmo zdaj vsa stikala. */
        for (stikalo = 0; stikalo < S && rezultat > 0; stikalo++)
        {
          if (obdelano[stikalo]) continue;
          /* Poskusimo, ali je dopustno, da tega stikala ne pritisnemo. */
          nMoznosti = 0;
          if (Preizkusi(stikalo, 0)) nMoznosti++;
          /* Pobrišimo stanje stikal, ki smo jih dosegli pri tem poskusu. */
          for (glava = 0; glava < rep; glava++) stanje[vrsta[glava]] = -1;
          /* Poskusimo še, ali je dopustno, da to stikalo pritisnemo. */
          if (Preizkusi(stikalo, 1)) nMoznosti++;
          rezultat = (rezultat * nMoznosti) % M;
        }

        fprintf(fo, "%d\n", rezultat); /* Izpišimo rezultat. */
      }
      fclose(fi); fclose(fo);
      return 0;
}
```

### 4. Bloki

Definicije blokov so odvisne le od zamika posameznih vrstic (števila presledkov na začetku vrstice), ne pa od preostanka vsebine v vrstici. Zato lahko že ob branju vhodnih

podatkov izračunamo zamik vsake vrstice in ga shranimo v neki tabeli; v nadaljevanju bomo delali le s to tabelo, vhodno besedilo pa sproti pozabljali. V spodnjem programu imamo v ta namen tabelo zamik; prazne vrstice (in vrstice, ki vsebujejo same presledke) so predstavljene z zamikom $-1$.

Tabelo zamikov pregledujmo po vrsti, od prve vrstice proti zadnji; spremenljivka i nam pove indeks trenutne vrstice. Koristno je imeti pri roki tudi zamik prejšnje neprazne vrstice (spremenljivka prejZamik), saj ga bomo potrebovali za ugotavljanje, kje se začne blok. Ko pridemo do neprazne vrstice, moramo preveriti naslednje:

- Če je zamik te vrstice večji od zamika prejšnje neprazne vrstice, se tu začne nov blok. Preostanek tega bloka bomo pregledali z rekurzivnim klicem, ob vrnitvi iz njega pa nam bo indeks trenutne vrstice povedal, kje se ta blok konča. Ta podatek si zapomnimo v tabeli blokDo, da bomo lahko na koncu izpisali bloke v izhodno datoteko.

- Če pa je zamnik trenutne vrstice manjši od zamika trenutnega bloka, se ta blok tukaj konča in se moramo vrniti iz trenutnega rekurzivnega klica.

Kljub rekurziji je ta postopek učinkovit, saj se indeks $i$ ves čas le povečuje in po vrnitvi iz rekurzivnega klica nadaljujemo s pregledovanjem tam, kjer je ta rekurzivni klic končal. Časovna zahtevnost tega postopka je le $O(n)$.

Šlo bi tudi brez rekurzije, vendar bi potem morali zamike trenutno odprtih blokov hraniti v nekakšnem seznamu, ki bi imel enako vlogo kot sklad, na katerem se pri rekurziji hranijo parametri posameznih vgnezdenih rekurzivnih klicev.

```
#include <stdio.h>

#define MaxN 100000
#define MaxDolz 1000
int i, n, zamik[MaxN], blokDo[MaxN], prejZamik;

void PoisciBloke(int zamikBloka)
{
  int razlika, blokOd;
  while (i < n)
  {
    /* Prazne vrstice preskočimo. */
    if (zamik[i] < 0) { i++; continue; }
    /* Zapomnimo si razliko glede na prejšnji zamik; nato pa
       zamik trenutne vrstice shranimo v prejZamik, kjer bo prišel
       prav v nadaljevanju. */
    razlika = zamik[i] - prejZamik;
    prejZamik = zamik[i];
    /* Če je zamik manjši kot zamik trenutnega bloka, se blok konča. */
    if (zamik[i] < zamikBloka) break;
    /* Če je zamik vsaj tolikšen kot v prejšnji vrstici, se blok nadaljuje. */
    else if (razlika <= 0) i++;
    /* Če je zamik večji kot v prejšnji vrstici, se začne nov vgnezden blok. */
    else /* if (razlika > 0) */
    {
      blokOd = i; i++;
      PoisciBloke(prejZamik); /* Poglejmo, do kod se razteza ta vgnezdeni blok. */
      blokDo[blokOd] = i - 1; /* Zapomnimo si ta blok v tabeli blokDo. */
    }
  }
}

int main()
{
  int z; char s[MaxDolz + 2];
  /* Preberimo vhodno datoteko. */
```

```
  FILE *f = fopen("bloki.in", "rt");
  fgets(s, sizeof(s), f); sscanf(s, "%d", &n);
  for (i = 0; i < n; i++) {
    fgets(s, sizeof(s), f);
    /* Določimo zamik trenutne vrstice. */
    z = 0; while (s[z] == ' ') z++;
    if (! s[z] || s[z] == '\r' || s[z] == '\n') z = -1;
    zamik[i] = z; blokDo[i] = -1; }
  fclose(f);
  prejZamik = -1; i = 0; PoisciBloke(-1);
  /* Izpišimo rezultate. */
  f = fopen("bloki.out", "wt");
  for (i = 0; i < n; i++) if (blokDo[i] >= 0) fprintf(f, "%d %d\n", i + 1, blokDo[i] + 1);
  fclose(f); return 0;
}
```

### 5. Poravnavanje desnega roba

Vprašanje, kako najbolje razbiti besedilo na vrstice, lahko razdelimo na dva podproblema: (1) izbrati si moramo, koliko besed bi vzeli v prvo vrstico (recimo prvih $k$); (2) potem pa moramo najti še najboljše razbitje preostalega besedila. Vidimo lahko, da je podproblem (2) pravzaprav enak prvotnemu, le da ima malo krajše besedilo: namesto besed $w_1, \ldots, w_n$ gledamo le besede $w_{k+1}, \ldots, w_n$. Ko bi reševali ta podproblem, bi znotraj njega našli podobne podprobleme s še krajšim besedilom.

Označimo torej s $f(i)$ oceno najboljšega razbitja besed $w_i, w_{i+1}, \ldots, w_n$. (Rezultat, po katerem sprašuje naloga, je potem $f(1)$.) Razmislek iz prejšnjega odstavka nam je torej pokazal, da velja

$$f(i) = \min\{\mathit{ocena}(i,j) + f(j+1) : i \le j \le n\}.$$

Z drugimi besedami, če vzamemo v prvo vrstico besede od $w_i$ do $w_j$, nam ostane podproblem z besedami $w_{j+1}, \ldots, w_n$; ocena prve vrstice je potem $\mathit{ocena}(i,j)$, ocena preostanka (če ta preostanek razbijemo na vrstice na najboljši možni način) pa $f(j+1)$. V gornji enačbi smo napisali $i \le j \le n$, vendar smemo iti v resnici z $j$ le tako daleč, dokler še lahko spravimo vse besede od $w_i$ do $w_j$ v eno samo vrstico.

Funkcijo $f$ bi lahko računali z rekurzivnim podprogramom, ki bi klical samega sebe, da bi pri izračunu vrednosti $f(i)$ izračunal vrednosti $f(j+1)$ za razne vrednosti $j$. Pri tem bi večkrat prišlo do rekurzivnih klicev z enako vrednostjo parametra, zato je koristno, če si že izračunane rezultate zapomnimo v neki tabeli, da jih ne bo treba računati po večkrat. Če računamo $f(i)$ po padajoči vrednosti $i$-ja, bomo vedno imeli že izračunane vse rešitve manjših podproblemov, ki jih bomo potrebovali za izračun $f(i)$. Zato rekurzivne funkcije niti ne potrebujemo več in lahko funkcijo računamo sistematično z zanko po $i$. Tako smo dobili naslednji postopek:

$f[n+1] := 0;$
**for** $i := n$ **downto** 1:
  $j := i;\ f[i] := \infty;$
  **while** $j \le n$ **and** besede od $i$ do $j$ gredo lahko v eno vrstico:
    $f[i] := \min\{f[i], \mathit{ocena}(i,j) + f[j+1]\};$
    $j := j + 1;$

Razmislimo še o tem, kako bi učinkovito preverjali, ali gredo besede od $i$ do $j$ še lahko v eno vrstico. Širina take vrstice bi bila $w_i + w_{i+1} + \ldots + w_j + (j-i) \cdot s$; preveriti moramo torej, če je ta vsota $\le d$. Ista vsota bo prišla prav tudi pri izračunu ocene te vrstice. V vsaki iteraciji, ko se $j$ poveča za 1, pridobi vsota $w_i + w_{i+1} + \ldots + w_j$ na koncu en nov člen, torej nove vsote ni težko računati iz prejšnje; v spodnji rešitvi hrani dosedanjo širino vrstice spremenljivka sirina, ki ji v vsaki iteraciji prištejemo dolžino naslednje besede (in presledka $s$).

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

```
#define MaxN 1000000
long long wi[MaxN + 1], f[MaxN + 1];

int main()
{
  int i, j, n; long long s, d, sirina, kand;

  /* Preberimo vhodne podatke. */
  FILE *g = fopen("poravnavanje.in", "rt");
  fscanf(g, "%lld %lld %lld", &n, &s, &d);
  for (i = 0; i < n; i++) fscanf(g, "%lld", &wi[i]);
  fclose(g);

  f[n] = 0;
  for (i = n - 1; i >= 0; i--)
  {
    sirina = 0;
    for (j = i; j < n; j++)
    {
      sirina += wi[j]; if (j > i) sirina += s;

      /* Ali gredo lahko besede od i do j še vse v eno vrstico? */
      if (sirina > d) break;

      /* Izračunajmo oceno najboljšega razbitja, ki se začne z vrstico i..j. */
      if (j == n - 1) kand = 0;
      else kand = (d - sirina) * (d - sirina) + f[j + 1];

      /* Če je to najboljše razbitje doslej, si ga zapomnimo. */
      if (j == i || kand < f[i]) f[i] = kand;
    }
  }

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

Za izračun ocen smo uporabili 64-bitne spremenljivke, ker je besed veliko in bi se lahko zgodilo, da bi bila skupna ocena celotnega razbitja večja od $2^{31}$.

Časovna zahtevnost tega postopka je načeloma $O(n \cdot m)$, če je $m$ največje število besed, ki gredo v eno vrstico. V našem primeru gre lahko $n$ do $10^6$, število besed v eni vrstici pa je lahko največ okrog 500 (ker imamo $d \leq 1000$, vsaka beseda je široka najmanj 1 in med besedami je še presledek širine najmanj 1). Za namene naše naloge je ta rešitev čisto dovolj dobra, omenimo pa lahko, da obstajajo za reševanje tega problema tudi učinkovitejši postopki, ki dosežejo časovno zahtevnost $O(n \log n)$ ali celo le $O(n)$.[3]

Viri nalog: pacifistični generali, tiskana vezja — Nino Bašić; luči — Tomaž Hočevar; potenciranje, poravnavanje desnega roba — Matjaž Leonardis; dnevnik — Mark Martinec; davek na ograjo, skrivno sporočilo — Jure Slak; proizvodnja čopičev, uniforme, prenova ceste, ljudožerci na premici, po Indiji z avtobusom — Mitja Trampuš; vnos šifre, bloki — Janez Brank. Hvala Tomažu Hočevarju za pomoč pri implementaciji rešitev nalog za 3. skupino. Primer pri nalogi „bloki“ je iz 1. poglavja Gibbonove *Zgodovine zatona in propada Rimskega cesarstva*.

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

[3] Glej npr. `http://xxyxyz.org/line-breaking/` in tam navedeno literaturo, še posebej: D. S. Hirschberg, L. L. Larmore: *The least weight subsequence problem*, SIAM J. on Computing, 16(4):628–38, April 1987; A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, R. Wilber: *Geometric applications of a matrix-searching algorithm*, Algorithmica 2(1–4):195–208, November 1987; R. Wilber: *The concave least-weight subsequence problem revisited*, J. of Algorithms 9(3):418–25, September 1988.