10. tekmovanje ACM v znanju računalništva za srednješolce

21. marca 2015

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.");
   }
}
```

10. tekmovanje ACM v znanju računalništva za srednješolce

21. marca 2015

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. Delni izid

Na košarkarskih tekmah ekipe običajno zelo nihajo v nivoju igre. Tako pride do zanimivih delnih izidov v prid eni ali drugi ekipi, ki jih novinarji vestno analizirajo in sporočajo. *Delni izid* je razlika med točkami, ki sta jih v nekem časovnem intervalu dosegli ekipi.

Opiši postopek (ali napiši program, če ti je lažje), ki izračuna največji delni izid (ne glede na to, katera od obeh ekip je pri tem delnem izidu dosegla več točk, katera pa manj). Tvoj postopek kot vhodne podatke dobi zaporedje celih števil, ki predstavljajo posamezne koše, dosežene v tekmi. Urejeni so po času (od začetka tekme proti koncu); koši, ki jih je dosegla ena ekipa, so predstavljeni s pozitivnimi števili, tisti, ki jih je dosegla druga ekipa, pa z negativnimi. Z branjem vhodnih podatkov se ti ni treba ukvarjati, pač pa predpostavi, da jih tvoj postopek oz. program dobi v neki primerni spremenljivki ali podatkovni strukturi.

Primer: če dobimo vhodno zaporedje

$$-2, -2, +2, +2, +3, +2, -2, +2, -2, -3, +2, -2, -2, -2, +2,$$

je največji možni delni izid 9. Dosežen je celo večkrat: pri podzaporedju $\langle +2, +2, +3, +2\rangle$ je razlika 9 v prid prve ekipe, pri podzaporedju $\langle -2, -3, +2, -2, -2, -2\rangle$ je razlika 9 v prid druge ekipe, obstaja pa še nekaj drugih podzaporedij, ki tudi dajo delni izid 9.

2. Kompresija

Merilna postaja meri višino Blejskega jezera in podatke po mobilni povezavi sporoča v meteorološki center. Meritve so pozitivna cela števila in se le počasi spreminjajo. Tipičen primer izmerjenih podatkov:

$$310, 310, 310, 310, 310, 311, 311, 311, 313, 313, 311.$$

Ker bi radi varčevali pri količini zakupljenega podatkovnega prometa, smo se odločili, da bomo vsak drugi ponovljeni podatek izpustili. Namesto zgornje serije meritev bi tako radi v center sporočili le:

$$310, 310, 310, 311, 311, 313, 311.$$

Kadar je zaporednih podatkov z isto vrednostjo liho mnogo, število sporočenih podatkov zaokrožimo navzgor. V zgornjem primeru se to zgodi pri meritvah 310 in 311.

Napiši program, ki se bo vrtel v neskončni zanki, bral podatke z merilne postaje in jih sporočal v center. Na voljo ima dva podprograma oz. funkciji:

- **int** Preberi();
  Funkcija počaka, da strojna oprema opravi meritev, ter vrne izmerjeno vrednost. Ta je vedno večja od 0.
- **void** Sporoci(**int** meritev);
  Podprogram sporoči meritev v center.

Program napiši tako, da bo izmerjene vrednosti kar najhitreje poslal dalje. (To pomeni, da ne boš dobil(a) vseh točk, če bo program čakal, da se bo vrednost meritve spremenila, ter šele nato sporočil polovico zapomnjenih vrednosti.)

Še deklaracije v drugih jezikih:

```
{ v pascalu }
function Preberi: integer;
procedure Sporoci(meritev: integer);
```

```
# v pythonu
def Preberi(): ...      # vrne int
def Sporoci(meritev): ...
```

3. znajdi.se

Razvili smo nov iskalnik *znajdi.se*, ki nam izpiše, kako pridemo iz točke A v točko B. Problem je, da nam iskalnik korake izpiše v pomešanem vrstnem redu. Tako se opis poti iz kraja G v kraj S glasi:

```
G S
- mimo O nadaljujemo skozi tri križišča, dokler ne pridemo do A
- pri B zavijemo desno, dokler ne dospemo do S
- iz A zavijemo v krožno križišče, od tam nadaljujemo naravnost do D
- na H zavijemo desno proti O
- od D sledimo modrim oznakam, dokler ne zagledamo B
- nato pri J peljemo naprej 1,2 km proti H
- iz G zavijemo levo proti J
```

Na srečo veljajo naslednje omejitve, ki nam bodo prišle prav pri urejanju navodil v pravi vrstni red:

- vsak korak poti (od enega kraja do naslednjega) je v svoji vrstici (ki je dolga največ 100 znakov) in v njej se ime kraja, pri katerem se ta korak začne, pojavi prej kot ime kraja, pri katerem se ta korak konča; to pa sta tudi edina dva kraja, ki sta v tej vrstici omenjena;
- ime vsakega kraja je ena sama velika črka angleške abecede (od `A` do `Z` — kot vidimo tudi v gornjem primeru);
- drugače se v navodilih velike črke ne pojavljajo (vsi drugi znaki so male črke, številke, ločila in presledki);
- imena vseh krajev so med seboj različna in v nobenega ne gremo več kot enkrat;
- pot se gotovo ne konča v istem kraju, v katerem se začne;
- vsaka vrstica (razen prve, ki vsebuje začetni in končni kraj) predstavlja en korak poti (v vhodnih podatkih torej ni kakšnih odvečnih vrstic, ki ne bi bile del iskane poti).

Napiši program, ki bo prebral začetni in končni kraj ter zmešana navodila. Tvoja naloga je, da izpišeš kraje na poti v pravilnem vrstnem redu. Če torej tvoj program prebere zgornja navodila in podatek, da želiš priti iz G v S, mora izpisati:

```
GJHOADBS
```

Tvoj program lahko podatke bere s standardnega vhoda ali pa iz datoteke `pot.txt`, karkoli ti je lažje.

4. Dva od petih

Leta 1958 je podjetje IBM dalo na trg računalnik IBM 7070, ki je nadomestil predhodne modele računalnikov z elektronkami, saj je bil izdelan s tranzistorji in zato zmogljivejši in energijsko manj potraten.

Ker je bil model IBM 7070 namenjen poslovnim obdelavam, so bila števila v njem shranjena kot deset desetiških števk, vsaka števka pa zapisana s petimi biti, skupaj torej 50 bitov (in dodatni predznak, s katerim pa se v tej nalogi ne bomo ukvarjali).

Čeprav bi za zapis števke med 0 in 9 zadoščali štirje biti, so se izdelovalci zavedali možnosti napak pri hranjenju in obdelavi podatkov, zato so se odločili za pet bitov in izmed vseh možnih 32 kombinacij izbrali deset takih, pri katerih velja, da ima vsaka veljavna desetiška števka natanko dva bita od petih postavljena na 1, ostali trije pa morajo biti 0. Če se je med obdelavo kje pojavila nedovoljena kombinacija bitov, je bila javljena napaka.

Tako kodiranje omogoča, da zaznamo vsako posamično napako (sprememba enega bita iz 1 v 0 ali obratno), lahko pa celo več napak, če so vse iste vrste (vse iz 0 v 1 ali pa vse iz 1 v 0).

Tole je tabela, po kateri se pri IBM 7070 števila med 0 in 9 zakodirajo v petbitno kodo:

1	11000
2	10100
3	10010
4	01010
5	00110
6	10001
7	01001
8	00101
9	00011
0	01100

Napiši program, ki bo prebral eno vrstico z vhodne datoteke ali standardnega vhoda (kar ti je lažje), v kateri se nahaja zapis enega desetmestnega kodiranega števila. Vrstica vsebuje 50 takih znakov, ki so enice ali ničle (poleg njih so lahko v vrstici še drugi znaki, na primer presledki, vendar vse take druge znake zanemarimo; vsega skupaj pa je vrstica dolga največ 100 znakov), in predstavlja deset desetiških števk. Program naj izpiše prebrano število s števkami med „0“ in „9“, morebitne neveljavne števke v številu pa naj izpiše kot zvezdice.

Primer vhodnih podatkov:

```
01a100 0110000110, 10100  10110  x 010 10 00110;;;;10001 00000 00011
```

Pri tem primeru je rezultat:

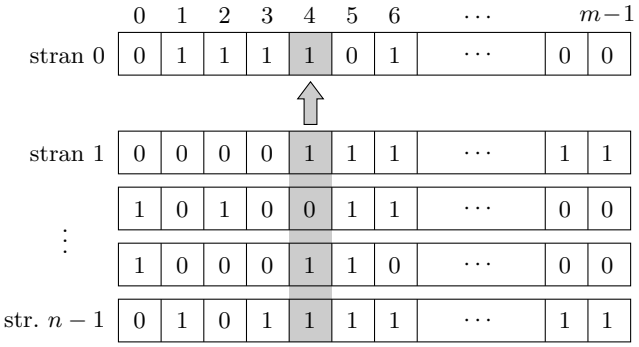
```
0052*456*9
```

5. Kontrolne vsote

V računalniku imamo pomnilnik razdeljen na n strani, vsako pa sestavlja m pomnilniških celic; posamezna celica hrani en bit podatkov (torej vrednost 0 ali 1). Na voljo sta dve funkciji, s katerima dostopamo do podatkov v tem pomnilniku:

- **int** Beri(**int** stran, **int** naslov) — prebere podatek iz celice z danim naslovom (naslov je število od 0 do $m-1$) iz dane strani (številke strani so od 0 do $n-1$); vrne vrednost 0 ali 1, če pa podatka ni bilo mogoče prebrati, vrne -1.
- **void** Pisi(**int** stran, **int** naslov, **int** novaVrednost) — zapiše novo vrednost (ki mora biti 0 ali 1) v dano stran na dani naslov. Če pri pisanju pride do napake, se funkcija vseeno vrne, kot da se ni zgodilo nič neobičajnega.

Ker se posamezne pomnilniške celice včasih okvarijo, smo se odločili eno od strani nameniti za *kontrolne vsote*. Recimo, da bo to stran številka 0. Kontrolne vsote so definirane takole: v posamezni celici strani 0 mora biti vrednost 1, če je v istoležnih celicah na ostalih straneh liho mnogo enic, sicer pa vrednost 0. (Istoležne celice so tiste, ki imajo znotraj strani enak naslov.)



Primer za $n = 5$. S sivo barvo je označen stolpec celic na naslovu 4; puščica ponazarja, da je kontrolna vsota (na strani 0) določena z vsebino ostalih celic na tem naslovu. V tem konkretnem primeru so na teh ostalih celicah tri enice, kar je liho število, zato je kontrolna vsota enaka 1.

Napiši funkciji Beri2 in Pisi2, ki ju bo lahko uporabnik našega pomnilnika poklical za branje in pisanje podatkov, pri čemer bosta skrbeli za kontrolne vsote (za dejansko branje in pisanje podatkov v pomnilnik naj kličeta funkciji Beri in Pisi):

- **int** Beri2(**int** stran, **int** naslov) — številka strani je zdaj le od 1 do $n-1$, predpostavimo torej lahko, da uporabnik ve, da iz strani 0 ne sme brati, ker je ta stran rezervirana za kontrolne vsote. Funkcija Beri2 naj prebere vrednost iz celice naslov strani stran, če pa to branje spodleti, naj pravo vrednost te celice rekonstruira (če je to mogoče) iz istoležnih celic na ostalih $n-1$ straneh. Prebrano (oz. rekonstruirano) vrednost naj funkcija Beri2 vrne kot rezultat, če pa vrednosti ni mogla niti prebrati niti rekonstruirati, naj vrne -1.
- **void** Pisi2(**int** stran, **int** naslov, **int** novaVrednost) — zapiše naj novo vrednost (ki je 0 ali 1) v dano stran (od 1 do $n-1$) na dani naslov (od 0 do $m-1$) in ustrezno popravi kontrolno vsoto v istoležni celici na strani 0.

Opiši, kako se tvoja rešitev obnaša ob primeru raznih napak pri branju ali pisanju, za katere v tej nalogi obnašanje ni natančno predpisano. Predpostavi, da so na začetku delovanja našega programa v vseh celicah na vseh straneh shranjene ničle (kar med drugim pomeni, da so tudi vse kontrolne enote na strani 0 pravilne).

Še deklaracije v drugih jezikih:

```
{ v pascalu }
function Beri(stran, naslov: integer): integer; { in podobno za Beri2 }
procedure Pisi(stran, naslov, novaVrednost: integer); { in podobno za Pisi2 }

# v pythonu
def Beri(stran, naslov): ... # in podobno za Beri2
def Pisi(stran, naslov, novaVrednost): ... # in podobno za Pisi2
```

10. tekmovanje ACM v znanju računalništva za srednješolce

21. marca 2015

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.

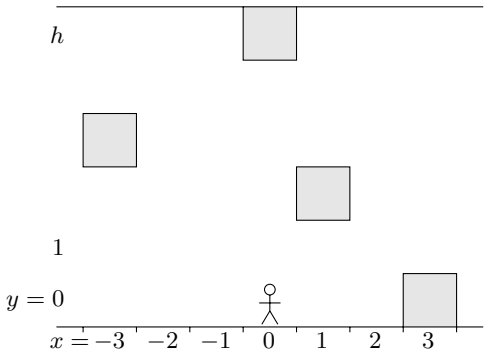
Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi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

Si figurica v arkadni igrici, kjer s stropa padajo kocke. Vsako sekundo se nekje na stropu pojavi kocka, ki s stalno hitrostjo en kvadratek na sekundo pada proti tlom. Ti se lahko vsako sekundo premakneš en kvadratek levo ali desno. Tvoj cilj je, da se izogneš vsem kockam in preživiš; če te kakšna kocka uspe speštati pod sabo v bitne črepinje, pa si igro izgubil. Ko kocka pade na tla, tam ostane in ti blokira pot, tako da ne moreš mimo.



(*Naloga se nadaljuje na naslednji strani.*)

Skupno število kock označimo z n (lahko jih je veliko, $n \leq 10^7$); za vsak čas t od 0 do $n-1$ vemo, da kocka, ki začne padati ob času t, pada na x-koordinati x_t. Vse kocke začnejo padati z višine h (kocka, ki se pojavi ob času t na višini h, torej pristane na tleh ob času $t+h$; ob tem času ti torej ne moreš več varno stati na polju x_t ali se nanj premakniti). Tvoj začetni položaj (ob času $t=0$) je $x=0$.

Števila $n, h, x_0, x_1, \ldots, x_{n-1}$ so podana. **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki na podlagi teh podatkov ugotovi, ali lahko preživiš ali ne. Če lahko preživiš, opiši tudi potrebne premike. **Utemelji**, zakaj tvoj postopek deluje.

2. Strahopetni Hektor

Pojdimo v čas trojanske vojne, ko Ahajci napadajo Trojance, ki čepijo za svojim obzidjem. Zid je zgrajen iz n dolgih kamnitih blokov, visokih 1 enoto, ki so položeni vodoravno drug na drugega, tako da se ne porušijo. Na vsaki višini imamo le en blok; za blok na višini i (za $i = 1, 2, \ldots, n$) je znano, da se začne na x-koordinati z_i in je dolg d_i enot.

Ti si na strani Ahajcev in želiš z ogromnim katapultom porušiti trojansko obzidje; na voljo imaš več krogel, s katerimi lahko ustreliš v zid in razbiješ najbolj levo enoto izbranega bloka. Če uspeš popolnoma uničiti en blok, se zid podre in Hektor, poveljnik trojanske strani, se nemudoma vda (to se zgodi celo, če je uničen vrhnji blok). Vendar je dovolj razbiti kakšen blok tudi le toliko, da se del zidu nad njim prevrne. To se zgodi, ko njegovo težišče ni več podprto s spodnjim blokom. Tudi če se v tem primeru zid le nagne, Hektor in njegova vojska izgubijo vse upanje na zmago in se predajo.

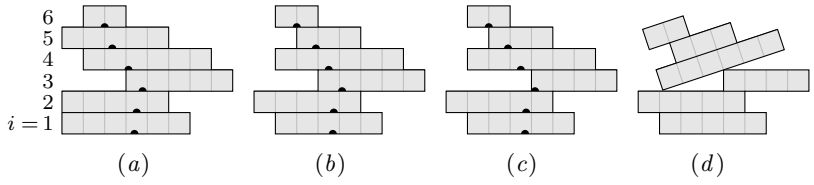
Opiši postopek (ali napiši program, če ti je lažje), ki bo pomagal Ahajcem simulirati potek rušenja takega obzidja, da se bodo lažje pripravili na boj. Postopek na začetku dobi opis zidu (vrednosti n, $z_1, \ldots, z_n$ in $d_1, \ldots, d_n$), nato pa naj v zanki bere podatke o izstreljenih kroglah (za vsako kroglo je podano, v kateri blok je bila izstreljena — to je število od 1 do n, pri čemer 1 pomeni najnižji blok, n pa najvišjega) in po vsaki prebrani krogli takoj *sproti* (še preden prebere naslednjo kroglo) izpiše, ali zid zdaj še stoji ali je že podrt.

Fizikalni poduk: x-koordinata težišča je definirana kot povprečje x-koordinat posameznih enot zidu. Vse enote zidu so enako težke. Če se težišče nahaja točno nad robom podpore, se telo še ne prevrne.

Primer: recimo, da imamo zid višine $n = 6$ z naslednjimi začetnimi podatki:

i	1	2	3	4	5	6
z_i	0	0	3	1	0	1
d_i	6	5	5	6	5	2

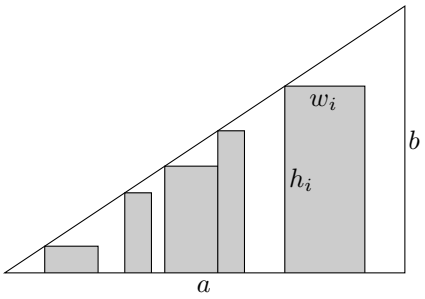
Ahajci bi lahko zid (ne nujno optimalno) napadali takole:



(a) Začetno stanje zidu. Črna pika na spodnjem robu posameznega bloka označuje položaj težišča tistega dela zidu, ki ga sestavljajo ta blok in vsi nad njim. — (b) Stanje zidu po treh strelih, enem na višini 1 in dveh na višini 5. Zid je še vedno stabilen. — (c) Če nato izvedemo še en strel na višini 3, zid ni več stabilen: blok 3 ne podpira več težišča blokov 4, 5, 6. Ti bi se zato nagnili in prevrnili, kot kaže slika (d).

3. Polaganje plošč

Imamo pravokotni trikotnik, širok a enot in visok b enot. Dobili smo tudi več pravokotnih plošč in poznamo njihove velikosti: i-ta plošča je široka w_i enot in visoka h_i enot. Plošče bi radi zložili v trikotnik tako, da bo spodnja stranica plošče ležala na spodnjem robu trikotnika (ki ima dolžino a; glej sliko spodaj), zgornje levo oglišče plošče pa bo ležalo na hipotenuzi trikotnika. Plošč ne smemo vrteti tako, da bi stranica w_i prišla po višini, h_i pa po širini. Poleg tega se plošče ne smejo prekrivati ali štrleti ven iz trikotnika, lahko pa se dotikajo. Naslednja slika kaže primer takega trikotnika, v katerega smo položili pet plošč:



Opiši postopek (ali napiši program ali podprogram, če ti je lažje), ki v okviru teh omejitev poišče največje možno število plošč, ki jih je mogoče položiti v trikotnik. Kot vhodne podatke tvoj postopek dobi velikost trikotnika (a in b), število plošč n in njihove dimenzije $w_1, h_1, w_2, h_2, \ldots, w_n, h_n$. **Utemelji**, zakaj je število plošč, ki ga najde tvoja rešitev, res največje možno.

4. Kodiranje

Recimo, da bi radi števke od 0 do 9 predstavili s 5-bitnimi kodami, torej z zaporedji petih ničel in enic. Takšnih peteric je kar $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$, mi pa moramo med temi 32 petericami izbrati deset peteric, ki jih bomo uporabili kot kode. Izbranim desetim petericam bomo rekli, da so *veljavne*, za ostalih 22 peteric pa recimo, da so *neveljavne*.

Če deset veljavnih peteric izberemo dovolj pazljivo, ima lahko tako dobljeni nabor kod nekatere lepe lastnosti, zaradi katerih je bolj odporen na napake pri branju, pisanju ali prenašanju podatkov. Primeri takšnih lepih lastnosti so:

(a) Če poljubni veljavni peterici spremenimo en bit (iz ničle v enico ali obratno), ali je tako spremenjena peterica zagotovo neveljavna? (Če to drži, potem vemo, da bomo zagotovo lahko zaznali napako pri prenosu podatkov, če se spremeni samo en bit.)

(b) Če poljubni veljavni peterici spremenimo eno ali več ničel v enice, ali je tako spremenjena peterica zagotovo neveljavna?

(c) Če v poljubni veljavni peterici enkrat med seboj zamenjamo dva sosednja različna bita (torej en par 01 spremenimo v 10 ali obratno), ali je tako spremenjena peterica zagotovo neveljavna?

Nekaj konkretnih primerov naborov 10 peteric:

Nabor	Katere lastnosti ima
00011, 00101, 00110, 01001, 01010, 01100, 10001, 10010, 10100, 11000	(a) in (b)
00000, 00011, 00110, 01001, 01100, 01111, 10010, 10101, 11000, 11011	(a) in (c)
00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111, 01000, 01001	nobene

Prvi od teh treh naborov na primer nima lastnosti (c): 00011 je veljavna peterica in če v njej zamenjamo tretji in četrti bit, dobimo 00101, ki je tudi veljavna peterica.

Napiši program ali podprogram, ki za dani nabor 10 različnih peteric preveri, ali ima lastnost (c) ali ne. Z branjem vhodnih podatkov se ti ni treba ukvarjati; predpostaviš lahko, da so peterice že shranjene v neki globalni spremenljivki. Peterice lahko obravnavaš kot nize 5 znakov (`'0'` in `'1'`) ali pa kot cela števila (od 0 do 31, pri čemer posamezni biti predstavljajo posamezne števke v peterici), kar ti je lažje.

5. Golovec

Na predoru Golovec so ob vhodu in izhodu iz predora namestili merilnike, ki ob vstopu ali izstopu iz predora zabeležijo registrsko številko avtomobila in čas vstopa v sekundah od začetka dneva. S pomočjo teh podatkov lahko, ko avtomobil zapelje iz predora, izračunamo njegovo povprečno hitrost pri vožnji skozi predor in odkrijemo tiste, ki so vozili prehitro.

Napiši podprogram (oz. funkcijo), ki ga bo sistem poklical vsakič, ko bo kak avtomobil zapeljal v predor ali iz njega. Tvoj podprogram naj ob izstopu avtomobila iz predora izračuna njegovo povprečno hitrost in če ta presega 22 m/s, naj izpiše registrsko številko tega avtomobila. Za shranjevanje podatkov si lahko deklariraš tudi globalne spremenljivke in opišeš, kako jih je treba inicializirati. Tvoj podprogram naj bo take oblike:

```
procedure Golovec(regStevilka: string; cas: integer);          { v pascalu }
void Golovec(char* regStevilka, int cas);                      /* v C/C++ */
void Golovec(string regStevilka, int cas);                     // v C++
public static void Golovec(String regStevilka, int cas);       // v javi
public static void Golovec(string regStevilka, int cas);       // v C#
def Golovec(regStevilka, cas): ...                             # v pythonu
```

Registrska številka je dolga največ 7 znakov. V predoru je največ 300 vozil hkrati in vsa imajo različne registrske številke. Vsako vozilo, ki kdaj zapelje v predor, prej ali slej zapelje tudi iz njega; za vsako tako vozilo bo sistem poklical naš podprogram natanko dvakrat: prvič, ko vozilo zapelje v predor, in drugič, ko zapelje iz njega. Vozila iz predora ne izstopajo nujno v enakem vrstnem redu, v kakršnem so vstopila vanj. Predor je dolg 622 m, ponoči pa v njem vedno potekajo nujna vzdrževalna dela, tako da naj te ne skrbi, da bi kakšen avtomobil vozil skozi predor ob polnoči.

10. tekmovanje ACM v znanju računalništva za srednješolce

21. marca 2015

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Delovno okolje

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

Na vsakem računalniku imaš na voljo disk `D:\`, v katerem lahko kreiraš svoje datoteke in imenike. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali VB.NET, mi pa jih bomo preverili s 64-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz OpenJDK 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, Eclipse in druga orodja.

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

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

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

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

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

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

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. (Izjemi sta prva naloga, kjer je testnih primerov 25 in za pravilen odgovor pri posameznem testnem primeru dobiš 4 točke, in peta naloga, kjer je testnih primerov 20 in za pravilen odgovor pri posameznem testnem primeru dobiš 5 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://tekmovanje.fri1.uni-lj.si/`):

- V pascalu:

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

- V C-ju:

```
#include <stdio.h>

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

- V C++:

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

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

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

- V javi:

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

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

- V C#:

```
using System.IO;

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

- V Visual Basic.NETu:

```
Imports System.IO

Module Poskus

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

End Module
```

10. tekmovanje ACM v znanju računalništva za srednješolce

21. marca 2015

NALOGE ZA TRETJO SKUPINO

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

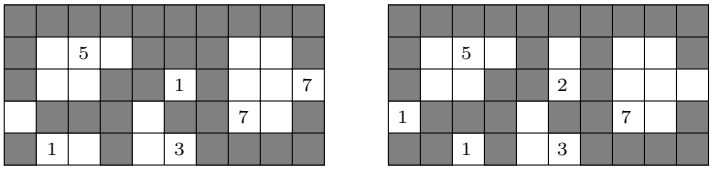
1. Nurikabe (`nurikabe.in`, `nurikabe.out`)

Nurikabe je igra, ki poteka na karirasti mreži. Posamezno polje mreže je lahko črno ali belo; na nekaterih belih poljih lahko stojijo tudi števila (od 1 do 9). Skupino črnih polj imenujemo *morje*, skupino belih polj pa *otok*. Polji, ki imata skupno stranico in sta iste barve, pripadata istemu morju (če sta črni) oz. istemu otoku (če sta beli).

Za otok rečemo, da je *pravilno označen*, če vsebuje natanko eno polje s številko in če je ta številka ravno enaka številu polj, ki tvorijo ta otok.

Med drugim lahko vidimo, da iz teh pravil sledi, da noben otok z več kot 9 polji ne more biti pravilno označen; in da imata dva različna otoka ali dve različni morji lahko skupno kakšno oglišče, ne pa tudi kakšne stranice.

Cilj igre nurikabe je izpolniti mrežo tako, da je morje eno samo, da so vsi otoki pravilno označeni in da v morju ni nobenega kvadrata 2×2 črnih polj. **Napiši program**, ki bo prebral opis mreže in pomagal preverjati te pogoje.



Primer: na levi mreži sta dve morji (črno polje v spodnjem levem kotu tvori majceno morje samo zase, vsa ostala črna polja pa tvorijo še drugo veliko morje) in 6 otokov, pri čemer trije niso pravilno označeni (otok velikosti 1 v najbolj levem stolpcu je brez številke; otok velikosti 2 v spodnji vrstici je pomotoma označen s številko 1; otok velikosti 7 na desni strani mreže pa je pomotoma označen dvakrat); in v mreži se pojavljata dva črna kvadrata velikosti 2×2 (ki se tudi malo prekrivata — ležita v prvih dveh vrsticah, od 5. do 7. stolpca). Desna mreža pa je izpolnjena po vseh pravilih.

Vhodna datoteka: v prvi vrstici sta dve celi števili, w in h, ločeni s presledkom. Pri tem je w širina mreže, h pa njena višina; veljalo bo $1 \leq w \leq 1000$ in $1 \leq h \leq 1000$. (V 60 % testnih primerov bo veljalo tudi $w \leq 20$ in $h \leq 20$.) Sledi h vrstic, vsaka od njih pa vsebuje w znakov, ki podajajo opis mreže. Pri tem so črna polja predstavljena z znaki `#`, bela polja brez številk s pikami `.`, bela polja s številko pa so predstavljena z znaki od `1` do `9`.

Izhodna datoteka: v prvo vrstico izpiši število morij; v drugo vrstico izpiši število otokov; v tretjo vrstico izpiši število pravilno označenih otokov; v četrto vrstico izpiši število črnih kvadratov velikosti 2×2 polj (štejejo tudi kvadrati, ki se prekrivajo oz. tvorijo še večja črna območja).

Primer vhodne datoteke:

```
10 5
##########
#.5.###..#
#..##1#..7
.###.##7.#
#1.#.3####
```

Pripadajoča izhodna datoteka:

```
2
6
3
2
```

(Opomba: to je leva mreža z gornje slike.)

2. Analiza signala (`signal.in`, `signal.out`)

Več oddajnikov oddaja signale — zaporedja ničel in enic, pri čemer se enice pojavljajo z neko konstantno periodo (ki pa je lahko pri različnih oddajnikih različna). Na primer, nek oddajnik oddaja s periodo 2:

$$1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots$$

Nek drug oddajnik pa s periodo 3:

$$1, 0, 0, 1, 0, 0, 1, 0, 0, 1, \ldots$$

Oddajniki so med seboj sinhronizirani tako, da na začetku opazovanja vsi oddajo enico (kot vidimo tudi v zgornjih dveh primerih).

Mi imamo detektor, ki zazna te signale. Pravzaprav ne more zaznati signalov posameznih oddajnikov, pač pa le njihovo vsoto. Pri zgornjih dveh oddajnikih bi na primer naš detektor zaznal takšno zaporedje:

$$2, 0, 1, 1, 1, 0, 2, 0, 1, 1, \ldots$$

Napiši program, ki analizira takšno zaporedje in ugotovi, koliko je vseh oddajnikov in kakšne so njihove periode.

Vhodna datoteka: v prvi vrstici je eno samo celo število n, ki pove dolžino našega zaznanega zaporedja; veljalo bo $1 \leq n \leq 100\,000$. (V 60 % testnih primerov bo veljalo tudi $n \leq 10\,000$.) V drugi vrstici je n nenegativnih celih števil, ki tvorijo prvih n členov zaporedja, ki ga je zaznal naš detektor; vsako od teh števil je manjše ali enako 10^9.

Zagotovljeno je, da v vhodnih podatkih ne bo napak; vhodno zaporedje je torej vedno takšno, do kakršnega bi res lahko prišlo z nekim primernim naborom oddajnikov in njihovih period.

Izhodna datoteka: v prvo vrstico izpiši dve celi števili, ločeni z enim presledkom; prvo od njiju naj bo število oddajnikov, drugo pa število oddajnikov, za katere z analizo prebranega zaporedja ni mogoče določiti periode. Sledi naj 0 ali več vrstic, po ena za vsako periodo, ki jo ima vsaj kakšen oddajnik. Vsaka od teh vrstic naj vsebuje dve pozitivni celi števili, ločeni z enim presledkom; prvo od njiju naj bo perioda, drugo pa število oddajnikov s to periodo. Urejene naj bodo naraščajoče po periodi.

Primer vhodne datoteke:

```
6
4 0 1 2 1 0
```

Pripadajoča izhodna datoteka:

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

3. Mafijski semenj (`semenj.in`, `semenj.out`)

Leto za letom v mesecu marcu na javnosti neznanem kraju poteka tradicionalni mafijski semenj, na katerem mafijci trgujejo z orožjem, ponarejenimi listinami, belimi praški in ostalim tovrstnim blagom. Letos se je zbralo n mafijcev, ki pripadajo m različnim mafijskim združbam.

Ravno letos pa je prišlo do neljubega incidenta. Iz daljave se sliši zavijanje policijskih siren in brnenje helikopterja. Kot vse kaže, jih je nekdo izdal. V paniki je vsak mafijec potegnil pištoli (kot se za resne mafijce spodobi, ima vsak pri sebi po dve pištoli) in ju usmeril v neka dva udeleženca sejma. Ker je bila panika nepopisna, se je lahko zgodilo tudi to, da je kateri od mafijcev obe pištoli usmeril v isto osebo ali celo samemu sebi v glavo.

Po začetnem preplahu so se mafijci hitro zbrali. Vsak je s pogledom premeril vse ostale in ugotovil, kdo vse je uperil pištolo v njega. Tisti mafijci, v katere ni usmerjena nobena pištola oz. tisti, v katere so usmerjene samo pištole mafijcev, ki pripadajo isti združbi, lahko pobegnejo. Tako so začeli drug za drugim bežati. Na koncu bo ostalo nekaj takih, ki ne bodo mogli pobegniti. Tem grozi, da bodo mnogo let preživeli na hladnem.

Napiši program, ki bo prebral podatke o tem, katerim mafijskim združbam pripadajo posamezni udeleženci in v koga merijo s pištolami, nato pa bo določil tiste mafijce, ki bodo šli na hladno.

Vhodna datoteka: v prvi vrstici sta števili n in m, ločeni z enim presledkom. Nato sledi n vrstic; i-ta med njimi vsebuje števila z_i, l_i in d_i, ločena s po enim presledkom. Pri tem je z_i številka združbe, ki ji pripada i-ti mafijec, l_i in d_i pa sta oznaki mafijcev, v katera i-ti meri s svojima pištolama. V vhodnih podatkih so mafijci označeni s števili $1, 2, \ldots, n$. Mafijske združbe so označene s števili $1, 2, \ldots, m$. Veljalo bo $1 \leq m \leq n \leq 10^6$, $1 \leq l_i \leq n$, $1 \leq d_i \leq n$ in $1 \leq z_i \leq m$. (Pri 50 % testnih primerov bo veljalo tudi $n \leq 1000$.)

Izhodna datoteka: izpiši oznake vseh mafijcev, ki bodo šli na hladno. Vsako oznako izpiši v svojo vrstico. Urejene naj bodo v naraščajočem vrstnem redu.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
2
4
5
6
```

4. Trgovanje z zrni (`zrna.in`, `zrna.out`)

V neki deželi je trgovanje z žitnimi zrni glavna dejavnost. Trgovanje poteka v glavnem pristanišču, kjer se vsako jutro zberejo večji in manjši lokalni kmetje ter trgovci z žitom. Cilj vsakega kmeta je, da trgovcu proda celoten pridelek, ki ga je pripeljal v pristanišče, in od tega nikakor ne odstopa. Vsak trgovec pride s svojo ladjo, ki lahko nosi določeno število zrn in ki bi se ob večji obremenitvi (četudi za eno samo zrno) nemudoma potopila.

Cilj vsakega trgovca je, da svojo ladjo čim bolj napolni; ne pozabimo pa, da hoče vsak kmet svoj pridelek prodati v celoti. **Napiši program**, ki za vsakega izmed trgovcev izračuna količino žitnih zrn, ki je najbližja nosilnosti njegove ladje (nikakor pa je ne presega) in bi jo lahko kupil ob predpostavki, da še nihče izmed kmetov ni prodal svojega pridelka.

Vhodna datoteka: v prvi vrstici sta dve celi števili, n in m, ločeni s presledkom; pri tem je n število kmetov (zanj velja $1 \leq n \leq 40$), m pa število trgovcev (zanj velja $1 \leq m \leq 400$). V drugi vrstici je n pozitivnih celih števil, ločenih s po enim presledkom; ta števila za vsakega od kmetov povedo, koliko zrn ima naprodaj. Sledi m vrstic, za vsakega trgovca po ena; vsaka od teh vrstic vsebuje po eno pozitivno celo število, ki pove nosilnost ladje tega trgovca (v zrnih). Noben kmet nima naprodaj več kot $3 \cdot 10^{12}$ zrn in noben trgovec ne želi kupiti več kot $2 \cdot 10^{13}$ zrn. (Nasvet: za računanje z zrni je koristno uporabiti kakšnega od 64-bitnih celoštevilskih tipov, na primer **long long** v C/C++, int64 v pascalu, **long** v C# in javi, Long v Visual Basic.NETu.)

Izhodna datoteka: vanjo izpiši m vrstic, za vsakega trgovca po eno (v enakem vrstnem redu, v kakršnem se ti trgovci pojavljajo v vhodni datoteki); v vsako od teh vrstic izpiši po eno samo celo število, namreč največje število zrn, ki bi jih ta trgovec lahko kupil ob upoštevanju omejitev naloge.

Primer vhodne datoteke:

```
5 7
2 3 5 11 44
1
6
42
49
8
12
22
```

Pripadajoča izhodna datoteka:

```
0
5
21
49
8
11
21
```

5. Razcep niza (`razcep.in`, `razcep.out`)

Pri tej nalogi se ukvarjamo z nizi, ki jih sestavljajo same ničle in enice. Število ničel v nizu s označimo z $N(s)$, število enic v nizu s pa označimo z $E(s)$. Zdaj lahko definiramo *oceno niza* kot $f(s) = \min\{N(s), E(s)\}$. To je torej število tistih števk (ničel oz. enic), ki jih je v tem nizu manj. Nekaj primerov: $f(\texttt{00110}) = 2$ (ker sta v tem nizu 2 enici in 3 ničle), $f(\texttt{111}) = 0$, $f(\texttt{1010}) = 2$.

Napiši program, ki dani niz razbije na natanko k nepraznih podnizov tako, da bo vsota ocen teh podnizov najmanjša možna.

Vhodna datoteka: v prvi vrstici sta dve celi števili, n in k, ločeni z enim presledkom; pri tem je n dolžina niza, ki bi ga radi razbili, k pa je število podnizov, na katere bi ga radi razbili. Veljalo bo $1 \leq k \leq n \leq 1000$ (v 40 % testnih primerov bo veljalo tudi $n \leq 20$). V drugi vrstici je niz, ki bi ga radi razbili; to je zaporedje n znakov, vsak od njih pa je bodisi „`0`“ bodisi „`1`“.

Izhodna datoteka: vanjo izpiši eno samo celo število, namreč najmanjšo možno vsoto ocen podnizov, ki jo lahko dosežemo, če niz iz vhodne datoteke primerno razbijemo na k nepraznih podnizov.

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:
`9 3` `110100100`	`2`

Komentar: niz `110100100` lahko razbijemo na tri podnize kot `1101` + `001` + `00`. Vsota ocen teh podnizov je $f(\texttt{1101}) + f(\texttt{001}) + f(\texttt{00}) = 1 + 1 + 0 = 2$. Pokazati je mogoče, da manjše skupne ocene pri razbijanju tega niza na tri podnize ni mogoče doseči.

10. tekmovanje ACM v znanju računalništva za srednješolce

21. marca 2015

REŠITVE NALOG ZA PRVO SKUPINO

1. Delni izid

Označimo vhodno zaporedje z $a_1, a_2, \ldots, a_n$ — torej je bilo na tekmi doseženih n košev in število a_i pove, koliko točk (in za katero ekipo) je bilo doseženih pri i-tem košu. Delni izid zdaj ni nič drugega kot vsota nekega podzaporedja tega zaporedja, torej vsota oblike $a_i + a_{i+1} + a_{i+2} + \ldots + a_{j-1} + a_j$ za neka i in j. Če je delni izid v prid prve ekipe, je ta vsota pozitivna, če je v prid druge ekipe, pa je negativna. Nas bo pravzaprav zanimala le absolutna vrednost te vsote, saj naloga sprašuje po največjem možnem delnem izidu ne glede na to, kateri ekipi je v prid.

Zelo preprosta rešitev je torej ta, da gremo v zankah po vseh možnih začetnih indeksih i, vseh možnih končnih indeksih j in pri vsakem paru (i, j) izračunamo delni izid; med tako dobljenimi delnimi izidi pa si zapomnimo največjega:

```
naj := 0;
for i := 1 to n:
  for j := i to n:
    vsota := 0;
    for k := i to j:
      vsota := vsota + a_k;
    if |vsota| > naj then naj := |vsota|;
return naj;
```

Časovna zahtevnost tega postopka je kar $O(n^3)$, saj imamo tri gnezdene zanke, ki imajo po $O(n)$ iteracij. Spomnimo se lahko, da če pri istem i povečamo j za 1, je vsota zelo podobna kot prej, le na koncu pridobi en dodatni člen. Torej vsote ni treba računati vsakič od začetka, ampak jo lahko postopoma dopolnjujemo, ko povečujemo j:

```
naj := 0;
for i := 1 to n:
  vsota := 0;
  for j := i to n:
    vsota := vsota + a_j;
    (* Zdaj je spremenljivka „vsota“ enaka a_i + a_{i+1} + ... + a_j. *)
    if |vsota| > naj then naj := |vsota|;
return naj;
```

Časovna zahtevnost tega postopka je le še $O(n^2)$, saj imamo pri vsakem paru (i, j) le še konstantno mnogo dela, da popravimo vsoto in si jo (če je treba) zapomnimo v *naj*.

Še boljšo rešitev pa dobimo takole: delne vsote, ki smo jih dobili pri $i = 1$, si je koristno zapomniti; naj bo torej $s_j = a_1 + a_2 + \ldots + a_{j-1} + a_j$. Pri $j = 0$ si mislimo še $s_j = 0$. Če bi imeli vse te vsote shranjene v neki tabeli, bi lahko zelo poceni izračunali poljubno vsoto oblike $a_i + a_{i+1} + \ldots + a_{j-1} + a_j$ — ta vsota je preprosto enaka $s_j - s_{i-1}$. Naloga torej pravzaprav sprašuje, kakšna je največja možna $|s_j - s_{i-1}|$. Če zapišemo vse delne vsote kot zaporedje $s_0, s_1, s_2, \ldots, s_{n-1}, s_n$, je jasno, da bomo največjo razliko (po absolutni vrednosti) dosegli takrat, če za s_{i-1} in s_j vzamemo največji in najmanjši člen tega zaporedja. To, ali je s_{i-1} najmanjši in s_j največji ali obratno, je pravzaprav nepomembno, saj bomo na koncu tako ali tako vzeli absolutno vrednost razlike med njima. Naš postopek mora torej le poiskati največjo in najmanjšo vrednost v zaporedju delnih vsot in vrniti razliko med njima:

```
min := 0; max := 0; vsota := 0;
for i := 1 to n:
  vsota := vsota + a_i;
  (* Zdaj je spremenljivka „vsota“ enaka s_i = a_1 + a_2 + ... + a_i. *)
  if vsota > max then max := vsota;
  if vsota < min then min := vsota;
return |max − min|;
```

Tu imamo torej le še eno zanko z n iteracijami, pri vsaki od njih pa konstantno mnogo dela, tako da je časovna zahtevnost tega postopka le še $O(n)$ in bi lahko učinkovito obdelal tudi zelo dolga vhodna zaporedja.

2. Kompresija

Naloga pravi, da moramo meritve sporočati naprej čim bolj sproti. Ko se odločamo o tem, ali bi trenutno meritev sporočili ali ne, lahko ločimo naslednje tri možnosti:

- če je trenutna meritev različna od prejšnje, jo vsekakor moramo sporočiti;
- če je trenutna meritev enaka prejšnji in smo prejšnjo že sporočili, potem trenutne rešitve ne smemo sporočiti;
- če pa je trenutna meritev enaka prejšnji in prejšnje nismo sporočili, potem trenutno rešitev moramo sporočiti.

Zadnji dve točki poskrbita, da od vsake skupine več zaporednih enakih meritev sporočimo prvo, tretjo, peto, sedmo in tako dalje; iz tega tudi sledi, da če je taka meritev lihe dolžine, bomo število sporočenih meritev pravilno zaokrožili navzgor, tako kot zahteva naloga (od petih meritev izpišemo tri, od sedmih meritev izpišemo štiri ipd.).

Koristno je torej, če naša rešitev poleg trenutne meritve hrani še prejšnjo meritev in še podatek o tem, ali smo prejšnjo meritev sporočili ali ne. Za to bi lahko uporabili dve spremenljivki, gre pa tudi z eno samo; v spodnjem programu je to spremenljivka prejsnja. Če prejšnje rešitve nismo sporočili, postavimo prejsnja na 0; pri naslednji meritvi (ki je zagotovo večja od 0 — to zagotavlja besedilo naloge) se nam bo torej zazdelo, da je drugačna od prejšnje, zato jo bomo zagotovo sporočili naprej. Zdaj sicer od prej omenjenih treh možnosti ne moremo razlikovati med možnostma 1 in 3, vendar to za nas niti ni pomembno, saj moramo v obeh primerih narediti isto stvar: sporočiti trenutno meritev naprej.

```
#include <stdbool.h>
int main()
{
  int meritev, prejsnja = 0;
  while (true)
  {
    meritev = Preberi();
    if (meritev == prejsnja)
      /* Trenutna meritev je enaka prejšnji in tisto smo že sporočili,
         zato trenutne ne smemo. To pa pomeni, da bomo naslednjo zagotovo
         morali sporočiti, zato postavimo spremenljivko prejsnja na 0. */
      prejsnja = 0;
    else {
      /* Trenutna meritev je različna od prejšnje (ali pa prejšnje nismo
         sporočili naprej), zato jo moramo sporočiti. */
      Sporoci(meritev);
      /* Zapomnimo si trenutno meritev v spremenljivki prejsnja; če ji bo
         naslednja meritev enaka, te naslednje ne bomo sporočili naprej. */
      prejsnja = meritev; }
  }
}
```

3. znajdi.se

Pri tej nalogi so imena krajev dolga le 1 črko in ta črka je velika črka angleške abecede, torej je možnih le 26 različnih krajev. Zato si lahko privoščimo tabelo, v kateri za vsak možen kraj od A do Z piše, kateri je njegov neposredni naslednik na naši poti. Spodnji program ima v ta namen tabelo nasl.

Na začetku preberimo začetni in končni kraj poti in si ju zapomnimo v spremenljivkah zac in kon; nato lahko postopoma beremo preostanek navodil in si v tabelo nasl shranjujemo podatke o poteku poti. Vse znake, ki niso velike črke, lahko kar preskočimo, saj za nas niso zanimivi. Velike črke pa nastopajo v parih, pri čemer prva pove začetni kraj na enem od korakov poti, druga pa končni kraj na tem koraku. Ko preberemo še drugo, si lahko v nasl zapomnimo, da je ona neposredna naslednica tiste prve. Prvo si med tem zapomnimo v spremenljivki od; ko pa preberemo še drugo veliko črko, postavimo od na -1, kar bo znak, da v nadaljevanju spet čakamo na prvo veliko črko v naslednjem paru (v naslednji vrstici navodil).

Na koncu se moramo le še z zanki sprehoditi po poti od začetnega kraja do končnega in jih sproti izpisovati; pri tem si pomagamo s tabelo nasl, da vemo, kako nadaljevati. Končni kraj prepoznamo po tem, da ima v tabeli nasl vrednost -1 (ker pač nima naslednika, saj je tam konec poti).

```
#include <stdio.h>

int main()
{
  int nasl[26], od = -1, c;
  char zac, kon;

  /* Preberimo ime začetnega in končnega kraja. */
  fscanf(stdin, "%c %c\n", &zac, &kon);
  zac -= 'A'; kon -= 'A';
  nasl[kon] = -1;

  /* Preberimo preostanek navodil znak po znak. */
  while ((c = fgetc(stdin)) != EOF)
  {
    /* Znake, ki niso velike črke, preskočimo. */
    if (c < 'A' || c > 'Z') continue;
    /* Če je od == -1, to pomeni, da je naslednja velika črka,
       ki jo bomo prebrali, ime začetnega kraja v trenutni vrstici. */
    if (od == -1) od = c - 'A';
    else {
      /* Sicer pa je naslednja prebrana velika črka ime končnega kraja v trenutni
         vrstici. Zdaj si lahko ta korak zapomnimo v tabeli nasl. */
      nasl[od] = c - 'A';
      /* Naslednja velika črka bo spet začetni kraj, zato postavimo „od“ spet na -1. */
      od = -1; }
  }

  /* Izpišimo potek poti. */
  while (zac != -1)
  {
    printf("%c", zac + 'A'); /* Izpišimo trenutni kraj... */
    zac = nasl[zac]; /* ...in se premaknimo na naslednjega. */
  }
  return 0;
}
```

4. Dva od petih

Vhodne podatke lahko beremo znak po znak; prebrane ničle in enice si sproti zapomnimo v neki spremenljivki (vse znake, ki niso 0 ali 1, lahko sproti preskočimo). Ko se nabere pet takih znakov, moramo ugotoviti, katero številko predstavlja ta peterica, in jo izpisati; če pa se izkaže, da peterica ne predstavlja nobene od desetih številk, izpišemo *.

Podrobnosti tega postopka je mogoče izvesti na več načinov. Peterice lahko predstavimo z nizi oz. tabelami znakov; vseh deset veljavnih peteric lahko hranimo v tabeli in gremo po njej z zanko, da ugotovimo, kateri od njih (če sploh kateri) je enaka naša pravkar prebrana peterica. Še ena možnost pa je, da na peterice gledamo kot na cela števila, zapisana v dvojiškem sestavu. Iz vsake peterice tako nastane celo število od 0 (= dvojiško 00000) do 31 (= dvojiško 11111). Zdaj si lahko privoščimo tabelo, ki za vseh 32 možnih peteric pove, kateri znak moramo pri tisti peterici izpisati; pri desetih so to števke od 0 do 9, pri 22 neveljavnih petericah pa izpišemo zvezdico *.

Tako dobimo naslednjo rešitev: trenutno peterico hranimo v spremenljivki x, število prebranih bitov pa v b; ko le-ta doseže 5, vemo, da smo prebrali že celo peterico in moramo izpisati pripadajoči znak, ki ga dobimo iz tabele izpis. Slednjo smo pripravili s pomočjo tabele v besedilu naloge; na primer, tam piše, da ima števka 6 kodo 10001, kar je dvojiški zapis števila 17, zato mora biti izpis[17] enak '6' ipd.

```
#include <stdio.h>
int main()
{
                /* 01234567890123456789012345678901 */
  char izpis[] = "***9*85**74*0****63*2***1*******";
  int c, x = 0, b = 0;

  /* Berimo vhodne podatke znak po znak. */
  while ((c = fgetc(stdin)) != EOF && c != '\n')
  {
    /* Znake, ki niso 0 ali 1, preskočimo. */
    if (c != '0' && c != '1') continue;
    /* Dodajmo pravkar prebrani bit v x. */
    x <<= 1; if (c == '1') x |= 1;
    /* Po vsakem petem prebranem bitu dekodirajmo prebrano peterico
       in izpišimo ustrezni znak. */
    if (++b == 5) {
      fputc(izpis[x], stdout);
      /* Pripravimo se na branje naslednje peterice. */
      x = 0; b = 0; }
  }

  return 0;
}
```

5. Kontrolne vsote

Razmislimo najprej o funkciji Beri2, ki je lažji del naloge. Za začetek lahko poskusimo zahtevani bit prebrati kar s funkcijo Beri; če se to branje posreči, vrnemo prebrano vrednost in smo končali. Če pa to branje spodleti, moramo v zanki prebrati istoležne celice na vse ostalih straneh in iz njih rekonstruirati vrednost, ki nas zanima. Če spodleti tudi kakšno od teh branj, iskane vrednosti ne moremo rekonstruirati in nam ne preostane kaj dosti drugega, kot da vrnemo -1.

Kako naj iz vrednosti na ostalih straneh rekonstruiramo tisto, ki nam je ni uspelo prebrati? Naj bo x_i (za $i = 0, \ldots, n-1$) vrednost celice na strani i in na naslovu, ki nas zanima (naša funkcija Beri2 ga je dobila v parametru naslov). Naloga pravi, da so kontrolne vsote definirane takole: če je $x_1 + x_2 + \ldots + x_{n-1}$ sodo število, je $x_0 = 0$, sicer pa je $x_0 = 1$. Isto pravilo pa lahko zelo elegantno zapišemo tudi takole: x_0 ima tako vrednost (izmed 0 in 1), da je $x_0 + x_1 + x_2 + \ldots + x_{n-1}$ v vsakem primeru sodo število.

Recimo zdaj, da iščemo x_s za neko stran s (pri čemer je $1 \leq s < n$). Lahko torej izračunamo vsoto $x_0 + \ldots + x_{s-1} + x_{s+1} + \ldots + x_{n-1}$ (recimo ji y); v kateri manjka le naša iskana vrednost x_s. Vemo, da mora biti $y + x_s$ sodo število; torej, če je y lih, mora biti $x_s = 1$, sicer pa mora biti $x_s = 0$. Vidimo torej, da iskani x_s ni nič drugega kot ostanek po deljenju y z 2; ali pa še drugače: x_s je enak najnižjemu bitu števila y.

Zelo elegantna možnost pa je, da namesto seštevanja uporabimo operacijo xor (v C/C++ in sorodnih jezikih jo dobimo z operatorjem ^. Spomnimo se, kako deluje xor na dveh bitih: če nek bit xor-amo z 0, se ne spremeni, če pa ga xor-amo z 1, se obrne

(iz 0 v 1 ali obratno). Če torej začnemo z bitom 0 in ga po vrsti xor-amo s števili $x_0, x_1, \ldots, x_{s-1}, x_{s+1}, \ldots, x_{n-1}$, bo na koncu prižgan, če je bilo med temi števili liho mnogo enic, sicer pa bo na koncu ugasnjen. Tako torej vidimo, da bo rezultat tega xor-anja na koncu ravno enak vrednosti x_s, ki jo iščemo. Zapišimo dobljeno rešitev v C-ju:

```
int Beri2(int stran, int naslov)
{
  int i, x, y;

  /* Poskusimo prebrati zahtevano celico. */
  x = Beri(stran, naslov);

  /* Če se je branje posrečilo, vrnimo njeno vrednost. */
  if (x >= 0) return x;

  /* Sicer preberimo istoležne celice na vseh ostalih straneh;
     xor njihovih vrednosti je ravno vrednost celice, ki nas zanima. */
  y = 0;
  for (i = 0; i < n; i++) if (i != stran)
  {
    x = Beri(i, naslov);

    /* Če spodleti branje kakšne od istoležnih celic, ne bomo mogli
       rekonstruirati vrednosti iskane celice. */
    if (x < 0) return -1;
    y ^= x;
  }
  return y;
}
```

Funkcija Pisi2 mora zapisati novo vrednost v zahtevano celico (kar lahko stori preprosto tako, da kliče funkcijo Pisi) in popraviti kontrolno vsoto na istoležni celici strani 0. Po vsem, kar smo doslej videli o kontrolnih vsotah, lahko razmišljamo takole: če se je trenutna celica s tem pisanjem spremenila (iz 0 v 1 ali obratno), se je s tem spremenila tudi parnost vsote $x_1 + x_2 + \ldots + x_{n-1}$, zato se mora spremeniti tudi kontrolna vsota x_0: če je bila le-ta prej 0, mora zdaj postati 1 in obratno. Če pa se trenutna celica pri pisanju ni spremenila (ker je bila novaVrednost enaka dosedanji vrednosti te celice), se tudi kontrolna vsota ne sme spremeniti.

Zametek funkcije Pisi2 je torej nekaj takšnega:

```
void Pisi2(int stran, int naslov, int novaVrednost)
{
  int staraVrednost, staraVsota;

  /* Preberimo staro vrednost celice. */
  staraVrednost = Beri(stran, naslov);

  /* Zapišimo novo vrednost. */
  Pisi(stran, naslov, novaVrednost);

  /* Če se vrednost ni spremenila, tudi kontrolne vsote ni treba spreminjati. */
  if (staraVrednost == novaVrednost) return;

  /* Sicer preberimo staro kontrolno vsoto. */
  staraVsota = Beri(0, naslov);

  /* In zapišimo novo kontrolno vsoto. */
  Pisi(0, naslov, 1 - staraVsota);
}
```

Toda v tej rešitvi dvakrat kličemo Beri; kaj se zgodi, če pri kakšnem od teh branj pride do napake? Če pride do napake pri branju Beri(stran, naslov), še ni treba takoj obupati, saj lahko poskusimo staro vrednost naše celice rekonstruirati iz kontrolne vsote in istoležnih celic na drugih straneh. To je ista stvar, ki jo že počne naša funkcija Beri2, zato lahko za branje uporabimo kar njo. Če spodleti tudi njej, potem vemo, da stare vrednosti naše celice ne moremo rekonstruirati (ker je okvarjena poleg nje še vsaj ena druga istoležna celica), zato se nam tudi s popravljanjem kontrolne vsote ni treba ukvarjati (če imamo

dve okvari na istem naslovu, nam tudi kontrolna vsota ne more več pomagati, saj z njo pri kasnejših branjih ne bomo mogli rekonstruirati vrednosti okvarjenih celic).

Če pa pride do napake pri branju Beri(0, naslov), to pomeni, da je okvarjena celica s kontrolno vsoto. Načeloma bi lahko tudi tu klicali Beri2, ki bi v takem primeru uspešno rekonstruiral staro kontrolno vsoto iz vrednosti istoležnih celic na straneh od 1 do $n-1$. Toda če je celica, v kateri bi morali hraniti kontrolno vsoto, okvarjena, si lahko mislimo, da ni posebne koristi od tega, da poskušamo računati novo kontrolno vsoto in jo vpisovati tja. Zato spodnja rešitev staro kontrolno vsoto vseeno bere kar z Beri namesto Beri2 in če to branje spodleti, kontrolne vsote ne poskuša popravljati.

```
void Pisi2(int stran, int naslov, int novaVrednost)
{
  int staraVrednost, staraVsota;

  /* Preberimo staro vrednost celice. */
  staraVrednost = Beri2(stran, naslov);

  /* Zapišimo novo vrednost. */
  Pisi(stran, naslov, novaVrednost);

  /* Če stare vrednosti nismo mogli rekonstruirati, tudi kontrolne vsote
     ne moremo popraviti; če pa je stara vrednost enaka novi, potem
     kontrolne vsote ni treba spreminjati. V obeh primerih se lahko takoj vrnemo. */
  if (staraVrednost < 0 || staraVrednost == novaVrednost) return;

  /* Preberimo staro kontrolno vsoto. */
  staraVsota = Beri(0, naslov);

  /* Če je branje uspelo, zapišimo novo kontrolno vsoto. */
  if (staraVsota >= 0) Pisi(0, naslov, 1 - staraVsota);
}
```

Mimogrede lahko še omenimo, da bi si lahko tudi pri računanju nove kontrolne vsote pomagali z operacijo xor. Recimo, da istoležne celice na vseh straneh spet označimo z $x_0, \ldots, x_{n-1}$, pri čemer je x_0 kontrolna vsota, torej je $x_0 = x_1 \operatorname{xor} \ldots \operatorname{xor} x_{n-1}$. Če se v izrazu na desni člen x_s spremeni v $\hat{x_s}$, se leva stran spremeni v $x_0 \operatorname{xor} x_s \operatorname{xor} \hat{x_s}$. O tem se lahko prepričamo, če upoštevamo lastnosti operacije xor: je komutativna in za poljuben a velja $a \operatorname{xor} a = 0$ in $a \operatorname{xor} 0 = a$.

REŠITVE NALOG ZA DRUGO SKUPINO

1. It's raining cubes

Nobene koristi ni od tega, da bi se premikali malo levo in malo desno; recimo namreč, da se najprej premikamo malo desno, nato pa v nekem trenutku naredimo korak v levo, z x na $x-1$. Vprašanje je, zakaj nismo že kar prej, ko smo že bili na $x-1$, tam tudi počakali, namesto da smo šli naprej na x (in zdaj z x nazaj na $x-1$). Edini možen razlog je, da je na $x-1$ medtem padla kocka in smo se ji s premikom na x izognili, da nas ni ubila; toda če je tako, je ta kocka še zdaj tam na $x-1$ in se tja zdaj sploh ne moremo premakniti. Ta razlog torej ne pride v poštev. Podobno bi se lahko prepričali, da če smo se najprej premikali v levo, nima smisla potem narediti koraka v desno.

Podobno tudi vidimo, da v resnici nima smisla čakati na nekem polju x in kasneje nadaljevati poti. Zakaj bi čakali tam (in stali pri miru, namesto da nadaljujemo z gibanjem v isto smer kot doslej)? Mar zato, ker vidimo, da bo vsak hip padla kocka na $x+1$, pa ne bi radi, da nas ubije? Če je tako, bo ta kocka potem tako ali tako ostala na $x+1$ in nam preprečila, da bi se še kdaj premaknili tja; torej je vseeno, če kar končamo naš sprehod in trajno ostanemo na x.

Tako torej vidimo, da se lahko omejimo na takšne vzorce gibanja, pri katerih se najprej nekaj časa premikamo (ves čas v isto smer, brez postankov), nato pa na nekem polju obstanemo in se odtlej ne premikamo več.

Katera polja pa lahko na ta način dosežemo? Naloga pravi, da ob času i začne na x-koordinati x_i in na višini h padati kocka, ki pristane na tleh ob času $i+h$. Ker se mi začnemo premikati ob času 0 na x-koordinati 0 in se na vsakem koraku premaknemo le

za 1 mesto levo ali desno, do koordinate x_i ne moremo priti prej kot ob času $|x_i|$. Torej, če je $|x_i| \geq i+h$, nam bo ta kocka preprečila, da bi prišli do polja x_i in sploh do vsakega polja, ki je v tisti smeri oddaljeno od našega začetnega položaja ($x = 0$) v isti smeri za vsaj $|x_i|$. Glede vseh ostalih polj pa velja, da nas ta kocka pri dostopu do njih ne bo nič ovirala.

Če zdaj za vsako kocko opravimo takšen razmislek, nam ostane nek interval možnih x-koordinat, ki so nam načeloma dosegljive; recimo $L < x < D$ za neka L in D. Če za vsaj eno koordinato na tem intervalu velja, da nanj nikoli ne pade nobena kocka, se lahko na začetku premaknemo nanj in tam počakamo do konca igre; če pa takega mesta ni, potem vemo, da ne bomo mogli preživeti.

Zapišimo dobljeni postopek še s psevdokodo:

$L := -\infty$; $D := \infty$;
for $i := 1$ **to** n:
  if $|x_i| < i + h$ **then continue**;
  if $x_i \geq 0$ **and** $x_i < D$ **then** $D := x_i$;
  if $x_i \leq 0$ **and** $x_i > L$ **then** $L := x_i$;
if obstaja tak x, ki je različen od vseh x_i in leži na $L < x < D$ **then** ($\star$)
  lahko preživimo: v prvih $|x|$ korakih se premaknemo na x in tam obstanemo;
else
  ne moremo preživeti;

Razmislimo še o tem, kako preveriti pogoj v vrstici ($\star$). Ena možnost je, da je interval neomejen; na primer, če je $L = -\infty$, lahko pridemo poljubno daleč proti levi, torej gremo lahko na primer na $x = \min_i x_i - 1$, kjer bomo levo od vseh kock, in tam počakamo konec igre. Podobno, če je $D = \infty$, lahko gremo na $x = \max_i x_i + 1$.

Še en poseben primer je, če je interval prazen (kar prepoznamo po tem, da je $D \leq L$); takrat lahko takoj zaključimo, da primernega x ni in da ne bomo preživeli.

Drugače pa moramo najti nekakšno vrzel med dvema kockama (ali pa med kocko in enim od krajišč L in D), torej območje, na katerega ne pade nobena kocka. Lahko si na primer koordinate vseh kock, ki padejo na območju $L < x_i < D$, zapišemo v neko tabelo; dodajmo vanjo še števili L in D; tabelo uredimo; v tako urejeni tabeli zdaj iščemo dve zaporedni števili, ki se razlikujeta za več kot 1. Če najdemo kakšen tak primer, to pomeni, da je tam med dvema kockama (ali pa med neko kocko in L ali D) vrzel, kamor se lahko premaknemo in tam počakamo konec igre.

Časovna zahtevnost tega postopka je $O(n \log n)$, zaradi urejanja; gre pa tudi brez urejanja. Vse kocke x_i, ki ležijo na $L < x_i < D$, zložimo v razpršeno tabelo; v tej razpršeni tabeli lahko zdaj v času $O(1)$ za poljubno x-koordinato preverimo, ali na njej leži kakšna kocka ali ne. Preverimo to za vsak $x = x_i + 1$ (za tiste x_i, ki ležijo na $L < x_i < D - 1$) in še za $x = L + 1$. Če je vsaj ena od teh koordinat prosta (na njej ne leži nobena kocka), se lahko premaknemo tja in preživimo, sicer pa je naš položaj brezupen. Tako imamo $O(n)$ poizvedb v razpršeno tabelo, tako da je časovna zahtevnost tega postopka le še $O(n)$.

2. Strahopetni Hektor

Za začetek se dogovorimo, kako bomo uporabljali koordinatni sistem. Če neka enota leži na x-koordinati a (za neko celo število a), kaj točno to pomeni? Ena možna interpretacija je, da je to x-koordinata središča enote; ta enota torej pokriva x-koordinate od $a - 1/2$ do $a + 1/2$; težišče bloka z začetkom z_i in dolžino d_i je tedaj $z_i + (d_i - 1)/2$. Druga interpretacija je, da a pomeni x-koordinato levega roba enote; ta enota torej pokriva x-koordinate od a do $a + 1$; težišče bloka z začetkom z_i in dolžino d_i je tedaj $z_i + d_i/2$. Načeloma je vseeno, katero interpretacijo si izberemo, važno je le, da se je potem dosledno držimo pri vseh izpeljavah in izračunih. V naši rešitvi bomo uporabljali prvo interpretacijo.

Označimo s t_i položaj (x-koordinato) težišča tistega dela zidu, ki ga tvorijo bloki na višinah od vključno i do vključno n. Težišče je povprečje x-koordinat vseh enot v teh blokih, torej ga lahko zapišemo kot $t_i = s_i/b_i$, pri čemer je s_i vsota x-koordinat vseh enot v teh blokih, b_i pa je število teh enot. Vse te stvari lahko računamo preprosto in učinkovito, če gremo od vrha zidu navzdol. Pri $i = n + 1$, kar je že nad našim

zidom, si mislimo $s_{n+1} = b_{n+1} = 0$. Recimo zdaj, da za nek i že poznamo s_{i+1} in b_{i+1}; kako bi izračunali s_i in b_i? Vse enote, ki so prišle v poštev za s_{i+1} in b_{i+1}, moramo šteti tudi v s_i in b_i, dodati pa jim moramo še vse enote bloka i. Teh je d_i, torej je $b_i = b_{i+1} + d_i$; in njihove koordinate so $z_i, z_i + 1, \ldots, z_i + d_i - 1$, torej je $s_i = s_{i+1} + z_i + (z_i + 1) + \ldots + (z_i + d_i - 1) = s_{i+1} + d_i \cdot (z_i + (d_i - 1)/2)$. Naš postopek gre lahko med inicializacijo v zanki po blokih od zgoraj navzdol in izračuna vse s_i, b_i in tudi težišča t_i.

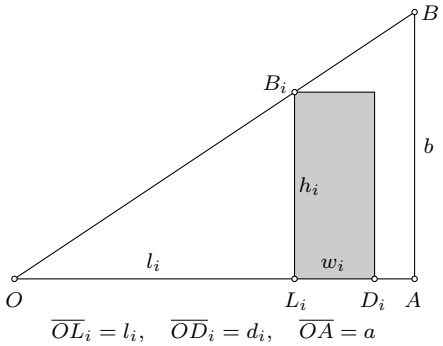
Naslednje vprašanje je, kako te podatke vzdrževati, ko krogle podirajo zid in se njegova oblika spreminja. Recimo, da zadene krogla blok na višini v. V njem torej poruši najbolj levo enoto, zato se z_v poveča za 1; in blok je zdaj za eno enoto krajši kot prej, zato se d_v zmanjša za 1. Ta blok je prispeval k vsotam s_i in b_i za vse i od 1 do v, zato jih moramo zdaj ustrezno zmanjšati: vsak tak s_i se zmanjša za (staro vrednost) z_v (to je bila namreč koordinata pravkar porušene enote), vsak b_i pa za 1. Ko tako popravimo vse s_i in b_i, lahko na novo izračunamo tudi težišča t_i.

Ko pa poznamo vsa težišča, težko preveriti, če zid še stoji. Blok i sestavljajo enote s koordinatami od vključno z_i do vključno $z_i + d_i - 1$, torej ta blok pokriva interval x-koordinat $[z_i - 1/2, z_i + d_i - 1/2]$. Na tem intervalu mora ležati težišče višje ležečega dela zidu (torej dela, ki ga tvorijo bloki od $i + 1$ do n), sicer se bo ta del nagnil in prevrnil. Pogoj za stabilnost je torej, da pri vsakem i (od 1 do $n - 1$) velja $z_i - 1/2 \leq t_{i+1} \leq z_i + d_i - 1/2$ (težišča t_1 nam ni treba preverjati, saj blok 1 leži na tleh in je torej težišče v vsakem primeru podprto), poleg tega pa mora seveda še pri vsakem i (od 1 do n) veljati $d_i > 0$ (da ni kakšen blok popolnoma uničen).

Časovna zahtevnost našega postopka je: $O(n)$ za inicializacijo in nato še $O(n)$ po vsaki izstreljeni krogli (toliko časa potrebujemo tako za izračun novih s_i, b_i, t_i kot za preverjanje, če zid še stoji). Ni pa nujno, da hranimo vrednosti s_i, b_i in jih popravljamo po vsaki krogli. Lahko bi le popravili z_v in d_v ter nato čisto na novo izračunali vse s_i, b_i in t_i po enakem postopku kot med inicializacijo. Časovna zahtevnost takšne rešitve bi bila še vedno le $O(n)$ za vsak strel.

3. Polaganje plošč

Naloga pravi, da mora zgornji levi kot plošče ležati na hipotenuzi našega trikotnika; s tem je položaj posamezne plošče čisto enolično določen — glede tega, kam jo položiti, nimamo nobene izbire, izbiramo lahko le to, ali jo sploh položimo ali ne. Kot vidimo iz spodnje slike, moramo ploščo i položiti tako, da njen levi rob leži na x-koordinati $l_i := a \cdot h_i / b$; njen desni rob pa torej leži na x-koordinati $d_i := x_i + w_i$. (Če se pri kakšni plošči izkaže, da je $d_i > a$, jo lahko takoj zavržemo, saj bi taka plošča nujno štrlela iz našega trikotnika, česar pa naloga ne dovoli.)



Veliki trikotnik $\triangle OAB$, v katerega smo položili sivo ploščo, je podoben manjšemu trikotniku $\triangle OL_iB_i$, ki ga tvori levo oglišče O skupaj z levim robom naše plošče. Ker sta si podobna, so razmerja v dolžinah stranic enaka:

$$\frac{l_i}{a} = \frac{h_i}{b},$$

iz česar dobimo $l_i = a \cdot h_i / b$.

Recimo, da polagamo plošče od leve proti desni. Če se odločimo najprej uporabiti ploščo i, to pomeni, da ne bomo mogli uporabiti nobene take plošče j, ki bi imela $x_j < d_i$, saj bi se taka plošča prekrivala s ploščo i (ali pa celo ležala levo od nje, kar bi kršilo našo odločitev, da bomo plošče polagali od leve proti desni). Torej je smiselno za začetek uporabiti tisto ploščo, ki ima najmanjšo vrednost d_i, saj nas bo ta najmanj omejevala pri izbiri nadaljnjih plošč. Če bi namesto z to ploščo začeli z neko drugo, recimo k, ki ima $d_k > d_i$, in potem nadaljevali z neko ploščo j (ki ima torej $x_j > d_k$), bi lahko namesto plošče k uporabili ploščo i, pa bi razpored še vedno ostal veljaven (če je $x_j > d_k$ in $d_k > d_i$, je tudi $x_j > d_i$, tako da sme plošča i stati pred ploščo j) in enako dober

(število plošč se nič ne spremeni). Torej res ni nič narobe, če začnemo naš razpored s ploščo, ki ima najmanjšo vrednost d_i — tako gotovo ne bomo spregledali najboljšega možnega razporeda.

Ko smo si na ta način izbrali prvo ploščo (recimo i), lahko v mislih zavržemo vse tiste plošče j, ki imajo $x_j < d_i$, nato pa nadaljujemo s podobnim razmislekom kot prej: med preostalimi ploščami izberemo tisto z najmanjšim d_j; nato zavržemo vse plošče, ki imajo $x_k < d_j$; tako nadaljujemo, dokler nam ne zmanjka plošč.

Zapišimo dobljeni postopek še s psevdokodo:

```
za vsako ploščo i:
  l_i := a · h_i/b;    d_i := l_i + w_i;
d := 0; (* d predstavlja desni rob zadnje doslej položene plošče *)
uredi plošče naraščajoče po d_i;
za vsako ploščo i v tem vrstnem redu:
  if l_i < d then
    continue; (* plošča i se prekriva z doslej položenimi *)
  položi ploščo i;
  d := d_i;
```

Časovna zahtevnost tega postopka je $O(n \log n)$, zaradi urejanja; vsi drugi koraki nam vzamejo le $O(n)$ časa.

4. Kodiranje

Recimo, da imamo naših deset peteric predstavljenih kar s tabelo 10×5 znakov `'0'` in `'1'` (v spodnji rešitvi je to globalna spremenljivka kode). Z dvema gnezdenima zankama se lahko sprehodimo po vseh možnih parih peteric in za vsak par preverimo, če bi se eno od njiju dalo predelati v drugo z eno zamenjavo dveh sosednjih bitov. Če najdemo kakšen tak par, lahko zaključimo, da naš nabor peteric iskane lastnosti (c) nima, sicer pa jo ima.

Razmisliti moramo še o tem, kako za dani dve peterici preveriti, ali je mogoče eno predelati v drugo z eno samo zamenjavo dveh sosednjih bitov. V zanki lahko primerjamo istoležne bite obeh peteric; ko opazimo neujemanje, preverimo, ali bi se ga dalo odpraviti z zamenjavo trenutnega in naslednjega bita. Če to ne gre, lahko takoj zaključimo, da sta si peterici preveč različni. Drugače pa si zapomnimo, da smo zamenjavo izvedli, in nadaljujmo s primerjanjem istoležnih bitov; če odtlej opazimo še kakršno koli drugo neujemanje, lahko tudi zaključimo, da sta si peterici preveč različni. Če pa pridemo do konca, ne da bi opazili še kakšno neujemanje, potem vemo, da bi se dalo eno peterico predelati v drugo z zamenjavo dveh sosednjih bitov (in torej nabor kot celota nima iskane lastnosti).

```
#include <stdbool.h>
enum { n = 10, d = 5 };
char kode[n][d];

bool ImaLastnostC()
{
  int x, y, i; bool dovoljRazlicni, zamenjava;

  /* Preglejmo vse pare različnih peteric in za vsak par preverimo,
     če sta si peterici dovolj različni. */
  for (x = 0; x < n; x++) for (y = 0; y < n; y++) if (x != y)
  {
    dovoljRazlicni = false; zamenjava = false;
    /* Primerjajmo istoležne bite in iščimo neujemanja. */
    for (i = 0; i < d; i++)
    {
      if (kode[x][i] == kode[y][i]) continue;
      /* Če smo zamenjavo že izvedli in opazimo še kakšno neujemanje,
         potem sta si peterici dovolj različni. */
      if (zamenjava) { dovoljRazlicni = true; break; }
```

```
        /* Če zamenjave še nismo izvedli, preverimo, ali bi lahko trenutno
           neujemanje odpravili z zamenjavo bitov i in i + 1. */
        if (i + 1 < d && kode[x][i] == kode[y][i + 1] &&
            kode[x][i + 1] == kode[y][i]) { i++; zamenjava = true; }
        /* Če se neujemanja ne da odpraviti z zamenjavo, sta si peterici dovolj različni. */
        else { dovoljRazlicni = true; break; }
      }
      if (! dovoljRazlicni) return false;
    }
    return true;
}
```

5. Golovec

Za vsako vozilo v predoru v neki podatkovni strukturi zapomnimo njegovo registrsko številko in čas prihoda v predor. Ko sistem pokliče naš podprogram, moramo preveriti, ali je vozilo z dano številko trenutno v predoru ali ne; če ga še ni v predoru, si ga le zapomnimo v naši podatkovni strukturi; če pa je že v predoru, lahko zdaj izračunamo njegovo povprečno hitrost (čas prihoda v predor imamo v naši podatkovni strukturi, čas izhoda iz predora pa smo pravkar dobili kot parameter funkcije Golovec) in če je previsoka, izpišemo njegovo registrsko številko. V vsakem primeru pa moramo nato podatke o tem vozilu pobrisati iz naše podatkovne strukture, saj ga zdaj ni več v predoru. (Če bo kasneje prišel spet kak klic za to vozilo, bo to pomenilo, da je isto vozilo ponovno zapeljalo v predor.)

Vprašanje je, kakšno podatkovno strukturo bi uporabili. Ker je vozil malo (naloga pravi, da jih je naenkrat v predoru največ 300), smo v naši spodnji rešitvi uporabili kar navadno tabelo (pravzaprav dve tabeli, eno za registrske številke in eno za čase prihoda). Tabela ima 300 elementov, dejansko število vozil v predoru (in s tem v tabeli) pa hranimo v globalni spremenljivki stVozil. Ta vozila so v naši tabeli shranjena na indeksih od 0 do stVozil − 1, vendar brez kakšnega posebnega vrstnega reda, tako da moramo pri preverjanju, ali je neko vozilo že v tabeli, iti kar v zanki po vseh elementih tabele in za vsakega primerjati njegovo registrsko številko s številko trenutnega vozila. Nova vozila dodajamo na konec tabele (na indeks stVozil), pri brisanju pa podatke za zadnje vozilo v tabeli (tisto na indeksu stVozil − 1) preprosto skopiramo v celico, iz katere smo vozilo pobrisali, in zmanjsamo stVozil za 1.

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

enum { dolzinaPredora = 622, maxHitrost = 22, maxVozil = 300, dolzinaStevilke = 7 };
int stVozil = 0, casPrihoda[maxVozil];
char regStevilke[maxVozil][dolzinaStevilke + 1];

void Golovec(char *stevilka, int cas)
{
  int i, casVoznje;

  /* Poglejmo, če že imamo podatek o vozilu s to številko. */
  i = 0; while (i < stVozil && 0 != strcmp(regStevilke[i], stevilka)) i++;
  /* Če vozila s to številko še nimamo, je očitno pravkar zapeljalo
     v predor, zato si ga le zapomnimo. */
  if (i >= stVozil) {
    strcpy(regStevilke[stVozil], stevilka);
    casPrihoda[stVozil++] = cas; return; }
  /* Sicer pa je pravkar zapeljalo iz predora; preverimo,
     če je njegova povprečna hitrost previsoka. */
  casVoznje = cas - casPrihoda[i];
  if (maxHitrost * casVoznje < dolzinaPredora)
    /* S povprečno hitrostjo maxHitrost se v tem času ne bi dalo prevoziti
       celega predora, torej je ta avtomobil vozil prehitro. */
    printf("%s\n", stevilka);
```

```
    /* Zdaj lahko to vozilo pobrišemo iz naše tabele. */
    strcpy(regStevilke[i], regStevilke[stVozil − 1]);
    casPrihoda[i] = casPrihoda[stVozil − 1];
    −−stVozil;
}
```

Pri takšni tabeli je dodajanje in brisanje vozila poceni, saj vzame le $O(1)$ časa, drago pa je iskanje, ki traja kar $O(n)$ časa, če je n število vozil v predoru. Možnih je še več drugih podatkovnih struktur: namesto tabele lahko uporabimo verigo, povezano s kazalci (*linked list*); vozila lahko hranimo urejena po registrski številki, kar bi nam omogočilo iskanje (z bisekcijo) v času $O(\log n)$, vendar bi za dodajanje in brisanje porabili po $O(n)$ časa, tako da s tem ne bi nič pridobili (saj vsako vozilo enkrat dodamo, enkrat pobrišemo in dvakrat iščemo). Boljša možnost je, da bi vozila hranili v drevesu (na primer AVL-drevesu ali rdeče-črnem drevesu), kjer bi tako iskanje kot dodajanje in brisanje trajala le po $O(\log n)$ časa. Še lepše pa bi bilo hraniti vozila v razpršeni tabeli (*hash table*), kjer vzamejo te operacije le po $O(1)$ časa.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Nurikabe

Podatke o mreži si preberimo kar v dvodimenzionalno tabelo (v spodnjem programu je to mreza), v kateri bo vsako polje predstavljal en znak (tipa **char**).

Števila črnih kvadratov velikosti 2×2 ni težko določiti; pojdimo v zanki po vseh poljih mreže, razen tistih v najbolj spodnji vrstici in najbolj desnem stolpcu, in za vsako preverimo, če tisto polje skupaj s svojim spodnjim, desnim in spodnjim desnim sosedom tvori tak črn kvadrat (če so vsa štiri polja črna, povečamo števec kvadratov za 1).

Malo več dela bo s štetjem otokov in morij. Postavimo se v poljubno polje mreže, recimo kar tisto v zgornjem levem kotu; iz pripadajočega znaka v tabeli mreza lahko ugotovimo, ali je tam morje ali otok; zdaj pa bi radi odkrili še vsa ostala polja, ki pripadajo temu morju ali otoku. To lahko naredimo z iskanjem v širino: začetno polje dodajmo v vrsto, nato pa na vsakem koraku vzemimo po eno polje iz vrste in dodajmo v vrsto vse tiste njegove sosede, ki pripadajo istemu morju ali otoku. Tako bomo prej ali slej obiskali vsa polja trenutnega morja ali otoka. Pri tem moramo paziti, da ne dodamo istega polja v vrsto po večkrat; zato bomo, ko polje prvič dodamo v vrsto, postavili njemu pripadajoč element v tabeli mreza na `'x'`, da bomo kasneje vedeli, da ga ne smemo več dodajati v vrsto.

Vrsto je mogoče implementirati na različne načine, najenostavnejši pa je kar s tabelo (v spodnjem programu je to spremenljivka vrsta) in dvema števcema, glava in rep. Elemente vrste tako hranimo na indeksih od glava do rep − 1; elemente pobiramo iz vrste pri glavi, nove elemente pa dodajamo pri repu.

Na koncu tega postopka smo torej pregledali celotno morje ali otok in označili vsa njegova polja z `'x'`. Če gre za otok (in ne morje), lahko spotoma tudi gledamo, če je na kakšnem njegovem polju številka; to številko si zapomnimo v spremenljivki oznaka. Če številke še nismo našli, naj ima oznaka vrednost 0; če pa smo našli več kot eno številko, postavimo oznaka na -1. Na koncu vemo, da je otok pravilno označen le, če je oznaka enaka številu polj na otoku; to število pa imamo v spremenljivki rep, saj se ta ob vsakem dodajanju polja v vrsto poveča za 1, torej na koncu pove ravno skupno število vseh polj na otoku.

Zdaj vemo, ali imamo morje ali otok in ali je otok pravilno označen, tako da lahko primerno povečamo ustrezne števce (nMorij, nOtokov in nOznacenih).

Ko smo tako obdelali prvi otok ali morje, lahko poiščemo naslednje še neobdelano polje (torej tako, ki v tabeli še nima znaka `'x'`) in na enak način obdelamo tudi njegov otok ali morje; tako nadaljujemo, dokler ni obdelana cela mreža. Na koncu moramo le še izpisati rezultate.

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

```
#define MaxW 1000
#define MaxH 1000
const int DX[] = { -1, 1, 0, 0 }, DY[] = { 0, 0, -1, 1 };
char mreza[MaxH][MaxW + 2];
int vrsta[MaxW * MaxH];

int main()
{
  int w, h, x, y, x1, y1, x2, y2, u, d, oznaka, glava, rep; bool otok; char c;
  int nMorij = 0, nOtokov = 0, nOznacenih = 0, nKvadratov = 0;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("nurikabe.in", "rt");
  fscanf(f, "%d %d\n", &w, &h);
  for (y = 0; y < h; y++) fgets(mreza[y], w + 2, f);
  fclose(f);

  /* Preštejmo črne kvadrate 2 * 2. */
  for (y = 0; y < h - 1; y++) for (x = 0; x < w - 1; x++)
    if (mreza[y][x] == '#' && mreza[y][x + 1] == '#' &&
    mreza[y + 1][x] == '#' && mreza[y + 1][x + 1] == '#') nKvadratov++;

  /* Poiščimo otoke in morja. */
  for (y = 0; y < h; y++) for (x = 0; x < w; x++)
  {
    /* Če je na trenutnem polju mreže znak 'x', to pomeni, da to polje
       pripada nekemu morju ali otoku, ki smo ga že obdelali, zato ga
       lahko zdaj preskočimo. */
    c = mreza[y][x]; if (c == 'x') continue;

    /* Sicer pa bomo njegov otok ali morje obdelali zdaj. Dodajmo ga v vrsto. */
    glava = 0; rep = 0; vrsta[rep++] = y * w + x; mreza[y][x] = 'x';

    /* Poglejmo, ali gre za otok in če da, ali ima že tudi oznako. */
    otok = (c != '#');
    oznaka = (c >= '1' && c <= '9') ? (c - '0') : 0;

    /* Preglejmo preostanek tega morja ali otoka. */
    while (glava < rep)
    {
      /* Vzemimo naslednje polje iz vrste in preglejmo njegove sosede. */
      u = vrsta[glava++]; x1 = u % w; y1 = u / w;
      for (d = 0; d < 4; d++) {
        x2 = x1 + DX[d]; y2 = y1 + DY[d];
        if (x2 < 0 || y2 < 0 || x2 >= w || y2 >= h) continue;

        /* Ali ta sosed sploh pripada istemu morju oz. otoku? */
        c = mreza[y2][x2];
        if (c == 'x' || (otok ? c == '#' : c != '#')) continue;

        /* Če je tu oznaka otoka, si jo zapomnimo. */
        if (c >= '1' && c <= '9') oznaka = (oznaka == 0) ? (c - '0') : -1;

        /* Dodajmo tega soseda v vrsto. */
        vrsta[rep++] = y2 * w + x2; mreza[y2][x2] = 'x'; }
    }

    /* Primerno povečajmo števce otokov in morij. */
    if (otok) { nOtokov++; if (oznaka == rep) nOznacenih++; }
    else nMorij++;
  }

  /* Izpišimo rezultate. */
  f = fopen("nurikabe.out", "wt");
  fprintf(f, "%d\n%d\n%d\n%d\n", nMorij, nOtokov, nOznacenih, nKvadratov);
  fclose(f); return 0;
}
```

2. Analiza signala

Označimo naš vhodni signal z $a_0, a_1, \ldots, a_{n-1}$. Ker so oddajniki sinhronizirani tako, da na začetku vsi oddajo enico, mi pa zaznamo vsoto vseh oddanih signalov, to pomeni, da je a_0 ravno število vseh oddajnikov.

Zdaj pa lahko razmišljamo takole: poiščimo najmanjši tak $t > 0$, pri katerem je $a_t > 0$. Ob času t oddajo signal le tisti oddajniki, katerih perioda je delitelj števila t. Če bi nek tak oddajnik imel periodo $p < t$, bi oddal enico že tudi ob času p, torej bi bil $a_p > 0$; mi pa smo t izbrali tako, da so med a_0 in a_t v zaporedju same ničle, torej takega oddajnika ni. Torej so enice, ki so se seštele v a_t, prispevali le oddajniki s periodo točno t. Zdaj torej vemo, da imamo točno a_t oddajnikov s periodo t; to si zapomnimo v neki tabeli (spodnji program ima v ta namen tabelo `np`). Ti oddajniki seveda ne oddajo enice le ob času t, ampak tudi ob času 0, $2t$, $3t$ in tako naprej. Ob vseh teh večkratnikih t-ja zmanjšajmo vrednost našega signala za število oddajnikov s periodo t. Tako nam ostane v naši tabeli a le vsota tistih signalov, ki so jih oddali oddajniki s periodo, večjo od t.

S tem postopkom lahko zdaj nadaljujemo in odkrivamo še nove oddajnike z vse večjimi periodami, dokler vse vrednosti $a_1, \ldots, a_{n-1}$ ne padejo na 0. Na tej točki se lahko zgodi, da je a_0 še vedno večji od 0. Če pride do tega, vemo, da obstajajo nekateri oddajniki s periodo, večjo ali enako n; od njih smo zaznali le enico ob času 0, kasnejših enic pa ne, ker jih nismo poslušali dovolj dolgo, da bi zaznali naslednjo enico. Zanje torej ne moremo ugotoviti, kakšne točno so njihove periode, nam pa sedanja vrednost a_0 pove vsaj to, koliko je takih oddajnikov. To pa je tudi ena od stvari, po katerih sprašuje naloga.

```
#include <stdio.h>
#define MaxN 100000
int a[MaxN], np[MaxN];

int main()
{
  int nOddajnikov, n, i, j;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("signal.in", "rt");
  fscanf(f, "%d", &n);
  for (i = 0; i < n; i++) fscanf(f, "%d", &a[i]);
  fclose(f);

  /* Ker vsi oddajniki na začetku oddajo enico, je prvi element
     prebranega signala ravno skupno število oddajnikov. */
  nOddajnikov = a[0];

  /* Preglejmo vse možne periode od 1 do n - 1. */
  for (i = 1; i < n; i++) {

    /* Na tej točki vsebuje tabela a samo še vsoto signalov vseh tistih oddajnikov,
       ki imajo periodo vsaj i. Poleg tega so a[1], ..., a[i - 1] že vsi enaki 0.
       Če je torej v a[i] neka neničelna vrednost, jo morajo povzročati oddajniki s periodo i
       (in ne kakšno krajšo). Zapomnimo si, koliko jih je. */
    np[i] = a[i];

    /* Odštejmo iz skupnega signala a tisto, kar prispevajo ti oddajniki.
       Torej moramo odšteti število teh oddajnikov pri vseh večkratnikih i-ja. */
    if (np[i] > 0) for (j = 0; j < n; j += i) a[j] -= np[i]; }

  /* Izpišimo rezultate. Za vsak i imamo np[i] oddajnikov s periodo i. Kar na koncu še
     ostane v a[0], so oddajniki s periodo, večjo ali enako n (ki v našem signalu
     prispevajo le enico v a[0] in nikjer drugje, ker je naša meritev prekratka). */
  f = fopen("signal.out", "wt");
  fprintf(f, "%d %d\n", nOddajnikov, a[0]);
  for (i = 1; i < n; i++)
    if (np[i] > 0) fprintf(f, "%d %d\n", i, np[i]);
  fclose(f); return 0;
}
```

3. Mafijski semenj

V nalogi se skriva problem topološkega urejanja grafa (v katerem je za vsakega mafijca po ena točka, usmerjena povezava pa obstaja tam, kjer en mafijec meri s pištolo na drugega, ki ne pripada isti združbi), vendar si lahko rešitev predstavljamo tudi kot preprosto simulacijo dogajanja, ki ga opisuje besedilo naloge.

Vhodne podatke preberimo v tri tabele, eno za z_i, eno za l_i in eno za d_i. Nato za vsakega mafijca izračunamo, koliko članov drugih združb meri vanj; to shranimo v tabeli stopnja (v grafu je *vhodna stopnja* točke definirana kot število povezav, ki kažejo v to točko).

Zdaj vemo, da lahko mafijci s stopnjo 0 takoj odidejo s prizorišča; zaradi njihovega odhoda se potem lahko zmanjša stopnja nekaterih drugih mafijcev; če kakšnemu od njih pade stopnja na 0, odide tudi on in tako naprej. Koristno je torej vzdrževati nekakšen seznam mafijcev, za katere že vemo, da bodo odšli, nismo pa še pregledali, na koga so merili in kako se zaradi njihovega odhoda spremenijo stopnje. V spodnjem programu imamo v ta namen tabelo toDo, v kateri hranimo mafijce, ki jih bo treba še obdelati (na indeksih od 0 do nToDo − 1).

Na začetku dodamo v seznam tiste, ki imajo že na začetku stopnjo 0. Nato na vsakem koraku vzamemo enega mafijca (recimo mu u) iz seznama; za vsakega v, v katerega je u uperil kakšno od svojih pištol, pogledamo, če pripada drugi združbi kot u, in če je tako, zmanjšamo v-jevo stopnjo za 1 (ker vemo, da bo u sčasoma odšel, zato bo takrat v v-ja uperjena ena pištola manj). Če kakšnemu v-ju zaradi tega pade stopnja na 0, dodamo v seznam toDo še njega.

Ta postopek se ustavi, ko se seznam toDo izprazni. Mafijci, ki imajo še zdaj stopnjo, večjo od 0, pa so tisti, ki bodo obtičali na sejmišču in jih moramo zdaj izpisati.

```
#include <stdio.h>
#define MaxN 1000000
int n, m, zi[MaxN], li[MaxN], di[MaxN];
int stopnja[MaxN], toDo[MaxN], nToDo;

int main()
{
  int u, v;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("semenj.in", "rt");
  fscanf(f, "%d %d", &n, &m);
  for (u = 0; u < n; u++) {
    fscanf(f, "%d %d %d", &zi[u], &li[u], &di[u]);
    li[u]--; di[u]--; stopnja[u] = 0; }
  fclose(f);

  /* Določimo vhodne stopnje vseh točk. */
  for (u = 0; u < n; u++) {
    if (zi[u] != zi[li[u]]) stopnja[li[u]]++;
    if (zi[u] != zi[di[u]]) stopnja[di[u]]++; }
  /* Točke z vhodno stopnjo 0 dodajmo v seznam toDo. */
  for (u = 0, nToDo = 0; u < n; u++)
    if (stopnja[u] == 0) toDo[nToDo++] = u;
  /* Topološko uredimo graf. */
  while (nToDo > 0) {
    u = toDo[--nToDo];
    v = li[u]; if (zi[u] != zi[v]) if (--stopnja[v] == 0) toDo[nToDo++] = v;
    v = di[u]; if (zi[u] != zi[v]) if (--stopnja[v] == 0) toDo[nToDo++] = v; }

  /* Izpišimo rezultate: točke, ki niso nikoli prišle v seznam toDo. */
  f = fopen("semenj.out", "wt");
  for (u = 0; u < n; u++) if (stopnja[u] > 0) fprintf(f, "%d\n", u + 1);
  fclose(f); return 0;
}
```

4. Trgovanje z zrni

Označimo z $a_1, \ldots, a_n$ število zrn, ki jih imajo naprodaj posamezni kmetje. Recimo, da bi nek trgovec rad kupil t zrn. Radi bi torej nekaj (nič ali več) števil izmed $a_1, \ldots, a_n$ sešteli tako, da bi bila vsota čim bližje t (vendar ne večja od t). Pri vsakem a_i imamo dve možnosti: lahko ga vzamemo v vsoto ali pa ne; tako je skupaj $2 \cdot 2 \cdot \ldots \cdot 2 = 2^n$ možnih vsot. Ker gre lahko n do 40, si ne moremo privoščiti, da bi izračunali vse te vsote in pogledali, katera med njimi je najbližja t, saj bi nam to vzelo preveč časa.

Koristna ideja je, da razdelimo kmete na dve skupini. V prvi bodo kmetje $a_1, \ldots, a_k$ (za nek primerno izbran k), v drugi pa preostali kmetje, torej $a_{k+1}, \ldots, a_n$. Za vsako skupino izračunajmo vse možne vsote in jih uredimo naraščajoče; tako imamo seznam 2^k vsot za prvo skupino in seznam 2^{n-k} vsot za drugo skupino. Če je vseh kmetov na primer $n = 40$, si lahko izberemo $k = 20$ in imamo torej dva seznama s po 2^{20} vsotami. To je malo več kot milijon v vsakem seznamu, kar je čisto obvladljivo; pri manjših n pa so seznami še krajši.

Označimo vsote na prvem seznamu z $v_1, \ldots, v_s$ in podobno tiste na drugem seznamu z $\hat{v}_1, \ldots, \hat{v}_{\hat{s}}$. Načeloma sta dolžini seznamov $s = 2^k$ in $\hat{s} = 2^{n-k}$, vendar sta lahko tudi krajša, če iz seznamov pobrišemo duplikate (če lahko enako vsoto dobimo na več načinov, si jo zapomnimo le enkrat). Zdaj bi torej radi z vsoto oblike $v_i + \hat{v}_j$ (za neka indeksa i in j) prišli čim bližje t (ne smemo pa ga preseči).

Tega se lahko lotimo na različne načine, pri čemer si pomagamo z dejstvom, da sta seznama urejena. Ena možnost je, da za vsak v_i s prvega seznama izvedemo bisekcijo po drugem seznamu, v katerem na ta način poiščemo največji tak $\hat{v}_j$, ki je še $\leq t - v_i$. Taka bisekcija nam pri vsakem v_i vzame $O(\log \hat{s}) = O(n - k)$ časa, tako da je časovna zahtevnost skupaj $O(2^k(n - k))$.

Druga možnost pa je neke vrste zlivanje seznamov. Načeloma nas pri vsakem elementu prvega seznama (recimo i) zanima, kateri je največji tak element drugega seznama (recimo mu $j(i)$), pri katerem vsota $v_i + \hat{v}_{j(i)}$ še ne preseže t. Za $j(i)$ torej velja, da če vzamemo poljuben kasnejši element, recimo $u > j(i)$, je vsota $v_i + \hat{v}_u$ gotovo prevelika (večja od t). Ker je tudi prvo zaporedje urejeno naraščajoče, je potem za $u > j(i)$ tudi vsota $v_{i+1} + \hat{v}_u$ večja od t (saj je v_{i+1} večji od v_i). Iz tega vidimo, da bo $j(i+1) \leq j(i)$ — vsi členi drugega zaporedja, ki dajo skupaj z v_i preveliko vsoto, dajo skupaj z v_{i+1} še bolj preveliko vsoto. Ko torej povečamo i za 1, se $j(i)$ lahko le zmanjša ali ostane enak, ne more pa se povečati. Po vsakem povečanju i-ja moramo v zanki zmanjševati j, dokler vsota $v_i + \hat{v}_j$ ne posane $\leq t$.

Časovna zahtevnost tega zlivanja je $O(s + \hat{s}) = O(2^k + 2^{n-k})$, saj se po prvem seznamu premikamo ves čas le navzgor (i se povečuje), po drugem pa ves čas le navzdol (j se zmanjšuje). Če sta seznama približno enako dolga (na primer pri $k \approx n/2$), je to hitreje od bisekcije (ima pa tudi to prednost, da pri zlivanju prevladujejo zaporedni dostopi do pomnilnika, pri bisekciji pa naključni dostopi, kar slabše izkoristi procesorjev predpomnilnik). Bisekcija bi bila lahko hitrejša, če vzamemo dovolj majhen k, vendar v tem primeru potrebujemo toliko več pomnilnika za drugi seznam. Pri omejitvah, ki veljajo na našem tekmovanju v tretji skupini, je rešitev z zlivanjem boljša.

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

#define MaxN 40
#define MaxM 400
#define MaxNakup 20000000000000LL

typedef long long znesek_t;

/* Ta funkcija vpiše v tabelo „vsote“ vse možne vsote članov iz tabele „cleni“
   (teh članov je nClenov) in vrne število teh vsot. */
int PripraviVsote(znesek_t *cleni, int nClenov, znesek_t *vsote)
{
  int nVsot = 0, i, j, vsota;
  vsote[nVsot++] = 0;
  for (i = 0; i < nClenov; i++)
    /* Na tem mestu imamo v vsote[0..nVsot - 1] že vse možne vsote prvih i članov.
       Dodajmo v tabelo še eno kopijo teh vsot, povečanih za cleni[i]; tako bomo
```

```
        dobili vse možne vsote prvih i + 1 členov. */
      for (j = nVsot − 1; j >= 0; j−−) {
        vsota = vsote[j] + cleni[i];
        if (vsota > MaxNakup) continue; /* prevelike vsote sproti zavržemo */
        vsote[nVsot++] = vsota; }
    return nVsot;
}

/* Primerjalna funkcija za urejanje s qsort() iz standardne knjižnice. */
int Primerjaj(const void *a, const void *b) {
    znesek_t A = *(const znesek_t *) a, B = *(const znesek_t *) b;
    return A > B ? 1 : A < B ? −1 : 0; }

int main(int argc, char** argv)
{
    int i, j1, j2, m, n, k, nVsot1, nVsot2;
    znesek_t kmetje[MaxN], *vsote1, *vsote2, nakup, naj;

    /* Preberimo seznam kmetov. */
    FILE *f = fopen("zrna.in", "rt"), g = fopen("zrna.out", "wt");
    fscanf(f, "%d %d", &n, &m);
    for (i = 0; i < n; i++) fscanf(f, "%lld", &kmetje[i]);

    /* Razdelimo jih na dve skupini (prvih k in ostalih n − k)
       in za vsako pripravimo urejen seznam vseh možnih vsot. */
    k = n / 2;
    vsote1 = (znesek_t *) malloc(sizeof(znesek_t) << k);
    vsote2 = (znesek_t *) malloc(sizeof(znesek_t) << (n − k));
    nVsot1 = PripraviVsote(kmetje, k, vsote1);
    nVsot2 = PripraviVsote(kmetje + k, n − k, vsote2);
    qsort(vsote1, nVsot1, sizeof(vsote1[0]), &Primerjaj);
    qsort(vsote2, nVsot2, sizeof(vsote2[0]), &Primerjaj);

    /* Obdelajmo vse trgovce. */
    for (i = 0; i < m; i++) {
      fscanf(f, "%lld", &nakup);
      naj = 0; /* Najboljši doslej najdeni rezultat. */
      j2 = nVsot2 − 1; /* Položaj v zaporedju „vsote2“. */
      /* Pojdimo po vseh možnih vsotah prvih k kmetov. */
      for (j1 = 0; j1 < nVsot1 && j2 >= 0; j1++)
      {
        /* Na tem mestu vemo, da so vse vsote oblike
           vsote1[j1] + vsote2[j] za j > j2 že prevelike (večje od iskanega nakupa).
           Mogoče je celo vsota za j = j2 prevelika — če je treba, zmanjšajmo j2. */
        while (j2 >= 0 && vsote1[j1] + vsote2[j2] > nakup) j2−−;
        /* Zdaj je v j2 največji tak indeks, pri katerem je vsota vsot1[j1] + vsote2[j2]
           še ≤ nakup. Če je to največja doslej dosežena vsota, si jo zapomnimo v „naj“. */
        if (j2 >= 0 && vsote1[j1] + vsote2[j2] > naj) naj = vsote1[j1] + vsote2[j2];
      }
      /* Izpišimo rezultat. */
      fprintf(g, "%lld\n", naj);
    }

    /* Pospravimo za sabo. */
    fclose(f); fclose(g); free(vsote1); free(vsote2); return 0;
}
```

Omenimo še dve drobni izboljšavi te rešitve. Ena je, da kmete na začetku uredimo padajoče, tako da v prvo skupino pridejo tisti z največjimi a_i. Tako bodo vsote na prvem seznamu čim večje in lahko upamo, da se bo pri mnogih t že kmalu med zlivanjem zgodilo, da bo v_i že sam po sebi presegel t in se bo lahko zlivanje takoj končalo (saj če je že v_i večji od t, bo tudi vsaka vsota oblike $v_i + \hat{v}_j$ večja od t).

Druga izboljšava pa se nanaša na pripravo urejenega seznama vsot. V gornji rešitvi smo najprej pripravili neurejen seznam (funkcija PripraviVsote) in ga nato uredili (s funk-

cijo qsort iz standardne knjižnice). Če gledamo na primer skupino k kmetov, bomo najprej porabili $O(2^k)$ časa za pripravo seznam vsot in nato $O(k \cdot 2^k)$ za urejanje. Z nekaj pazljivosti lahko do urejenega sezanam pridemo že v času $O(2^k)$. Recimo, da že imamo urejen seznam vseh možnih 2^{k-1} vsot za prvih $k-1$ kmetov. V mislih si pripravimo še eno kopijo tega seznama, v kateri vsako vsoto povečamo za a_k.[1] Zdaj imamo dva urejena seznama, ki oba skupaj vsebujeta ravno vse možne vsote k kmetov; vse, kar moramo še narediti, je, da ju zlijemo v en sam seznam. Zlivanje poteka tako, da začnemo na začetku obeh seznamov in na vsakem koraku pogledamo, kateri od njiju ima na trenutnem mestu manjši element; ta element premaknemo v izhodni seznam in se premaknemo za eno mesto naprej po tistem vhodnem seznamu, iz katerega smo ta element dobili. Časovna zahtevnost takega zlivanja je le $O(2^k)$; da pa smo sploh prišli do seznama vseh vsot za prvih $k-1$ kmetov, smo morali pred tem zlivati dva seznama za prvih $k-2$ kmetov in tako naprej. Skupna cena vseh teh zlivanj je $O(2^k + 2^{k-1} + 2^{k-2} + \ldots + 1) = O(2^{k+1})$, kar je še vedno precej hitreje od $O(k \cdot 2^k)$.

Namesto funkcije PripraviVsote in klica qsort za njo bi morali torej poklicati takšno funkcijo:

```
int PripraviUrejeneVsote(znesek_t *cleni, int nClenov, znesek_t *vsote)
{
  int nVsot = 0, nVsot2, i, i1, i2; znesek_t vsota, clen;

  /* Pripravimo si pomožno tabelo, ki bo dovolj velika za polovico vseh možnih vsot. */
  znesek_t *vsote2 = (znesek_t *) malloc(sizeof(znesek_t) << (nClenov > 0 ? nClenov - 1 : 0));

  /* Začnemo z eno samo vsoto (prvih 0 členov). */
  vsote[nVsot++] = 0;

  /* Pripravimo večje vsote. */
  for (i = 0; i < nClenov; i++) {

    /* Na tem mestu imamo v vsote[0..nVsot - 1] urejen seznam vseh možnih vsot
       prvih i členov. Skopirajmo jih v vsote2 in nVsot2. */
    for (i1 = 0, nVsot2 = nVsot; i1 < nVsot2; i1++) vsote2[i1] = vsote[i1];

    /* V mislih si predstavljajmo še eno kopijo tega seznama vsot, pri čemer vsaki
       prištejemo še naslednji člen, cleni[i]. Oba seznama bomo zdaj zlili (v tabelo
       vsote). Števca i1 in i2 povesta naš trentni položaj v obeh seznamih. */
    i1 = 0; i2 = 0; nVsot = 0; clen = cleni[i];
    while (i1 < nVsot2 || i2 < nVsot2) {

      /* Na indeksu i1 v prvem seznamu je vrednost vsote2[i1], na indeksu i2 v
         drugem seznamu pa je vrednost vsoste2[i2] + clen. Manjšo od teh vrednosti
         bomo prenesli v izhodni seznam (vsote) in se premaknili naprej po
         tistem od obeh vhodnih seznamov, iz katerega smo jo dobili. */
      if (i1 == nVsot2 || (i2 < nVsot2 && vsote2[i2] + clen < vsote2[i1]))
        vsota = vsote2[i2++] + clen;
      else vsota = vsote2[i1++];

      /* Spotoma še zavrzimo duplikate in morebitne prevelike vsote. */
      if (vsota <= MaxNakup && (nVsot == 0 || vsota > vsote[nVsot - 1]))
        vsote[nVsot++] = vsota; } }

  free(vsote2); /* Pobrišimo pomožno tabelo vsote2. */
  return nVsot;
}
```

5. Razcep niza

Označimo naš vhodni niz z $a = a_1 a_2 \ldots a_n$. Nalogo lahko rešujemo z rekurzivnim razmislekom: če hočemo razbiti a na k nepraznih podnizov, se moramo nekako odločiti, kako dolg naj bo zadnji od njih — recimo, da se zadnji podniz začne pri indeksu $i+1$ (zadnji podniz je torej $a_{i+1} a_{i+2} \ldots a_n$). Potem nam preostane le še to, da ostanek niza, torej $a_1 a_2 \ldots a_{i-1} a_i$, čim bolje razbijemo na $k-1$ nepraznih podnizov. Tu imamo torej enak problem kot na začetku, le niz je malo krajši (manjka mu zadnjih nekaj znakov) in zahtevan je en podniz manj kot prej.

[1] „V mislih“ pravimo zato, ker si ni treba zares delati kopije seznama, saj lahko sproti računamo njegove elemente iz elementov prvotnega seznama.

V splošnem imamo torej podprobleme takšne oblike: naj bo $g(n', k')$ najmanjša možna vsota ocen podnizov pri razbitju niza $a_1 a_2 \ldots a_{n'}$ na k' nepraznih podnizov. Naloga na koncu sprašuje po $g(n, k)$. Kot je pokazal razmislek v prejšnjem odstavku, lahko rešujemo te podprobleme s formulo $g(n', k') = \min\{g(i, k' - 1) + f(a_{i+1} a_{i+2} \ldots a_{n'}) : k' - 1 \leq i < n'\}$. (Omejitev $k' - 1 \leq i < n'$ izhaja iz dejstva, da če hočemo niz $a_1 \ldots a_i$ razbiti na $k' - 1$ nepraznih podnizov, mora biti ta niz dolg vsaj $k' - 1$ znakov; in podobno, da bo tudi zadnji podniz $a_{i+1} \ldots a_{n'}$ neprazen, mora biti $i < n'$.)

Da ne bomo računali istih vrednosti $g(n', k')$ po večkrat, si jih je koristno zapomniti v neki tabeli. Opazimo lahko, da ko računamo $g(n', k')$, potrebujemo le rešitve oblike $g(i, k' - 1)$, torej za podprobleme s $k' - 1$ podnizi, ne pa tudi za podprobleme s $k' - 2$ ali manj podnizi — tiste lahko sproti pozabljamo. Zato ima spodnji program tabelo g z le dvema vrsticama, eno za k' in eno za $k' - 1$ (rešitev podproblema $g(n', k')$ hranimo v g[k' % 2][n']). Podprobleme je pametno reševati sistematično od manjših k' proti večjim — tako bomo imeli vedno pri roki rešitve manjših podproblemov, ko jih bomo potrebovali.

Razmislimo še o tem, kako bi učinkovito računali funkcijo f za ocenjevanje posameznega podniza. Načeloma moramo prešteti, koliko je v podnizu ničel in koliko enic, ocena f pa je potem manjše od teh dveh števil. Koristno si je vnaprej pripraviti dve tabeli, s_0 in s_1: pri tem naj $s_c[i]$ pove, kolikokrat se pojavlja števka c v nizu $a_{i+1} a_{i+2} \ldots a_n$. To je najlažje računati po padajočih i; na začetku imamo $s_c[n] = 0$, nato pa $s_c[i] = s_c[i+1] + 1$, če je $a_{i+1} = c$, oz. $s_c[i] = s_c[i + 1]$, če $a_{i+1} \neq c$.

Zdaj za poljuben podniz oblike $a_{i+1} a_{i+2} \ldots a_j$ vemo, da vsebuje $s_c[i] - s_c[j]$ pojavitev števke c. Če izračunamo to za $c = 0$ in $c = 1$ in vzamemo manjšo od obeh vrednosti, dobimo ravno oceno $f(a_{i+1} \ldots a_j)$.

```
#include <stdio.h>
#define MaxN 1000

int main()
{
  char a[MaxN + 2];
  int g[2][MaxN + 1], s0[MaxN + 1], s1[MaxN + 1];
  int n, k, nn, kk, i, kand, naj;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("razcep.in", "rt");
  fscanf(f, "%d %d\n", &n, &k);
  fgets(a, MaxN + 2, f); fclose(f);

  /* s0[i] in s1[i] naj bosta število ničel in enic v a[i..n-1]. */
  for (i = n - 1, s0[n] = 0, s1[n] = 0; i >= 0; i--) {
    s0[i] = s0[i + 1] + (a[i] == '0' ? 1 : 0);
    s1[i] = s1[i + 1] + (a[i] == '1' ? 1 : 0); }

  /* g(kk, nn) nam pomeni oceno najboljšega možnega razcepa
     niza a[0..nn-1] na kk nepraznih podnizov. Hranili ga bomo v
     g[kk % 2][nn]. Za začetek jih izračunajmo za kk = 1. */
  for (nn = 0; nn <= n; nn++)
    g[1][nn] = (s0[0] - s0[nn] < s1[0] - s1[nn]) ? s0[0] - s0[nn] : s1[0] - s1[nn];

  /* Rešimo še podprobleme za večje kk. */
  for (kk = 2; kk <= k; kk++) for (nn = kk; nn <= n; nn++)
  {
    naj = n + 1; /* To je večje od ocene vsakega razbitja. */
    /* Niz a[0..nn-1] hočemo razbiti na kk nepraznih podnizov. Zadnji med
       njimi bo torej oblike a[i..nn-1] za nek i, pred tem pa imamo tedaj
       problem, kako razbiti a[0..i-1] na kk - 1 nepraznih podnizov. */
    for (i = kk - 1; i < nn; i++) {
      /* Izračunajmo oceno zadnjega kosa, a[i..nn-1]. */
      kand = s0[i] - s0[nn];
      if (kand > s1[i] - s1[nn]) kand = s1[i] - s1[nn];
      /* Prištejmo še oceno najboljšega razbitja niza a[0..i-1]
         na kk - 1 nepraznih podnizov. */
```

```
        kand += g[(kk - 1) % 2][i];
        /* Če je to najboljša rešitev doslej, si jo zapomnimo. */
        if (kand < naj) naj = kand; }
      /* Shranimo rezultat v tabelo g. */
      g[kk % 2][nn] = naj;
    }
    /* Izpišimo rezultat, torej g(k, n), ki ga hranimo v g[k % 2][n]. */
    f = fopen("razcep.out", "wt");
    fprintf(f, "%d", g[k % 2][n]); fclose(f); return 0;
}
```

Pri vsakem paru (n', k') imamo $O(n')$ dela, da pregledamo vse možne i in poiščemo najboljši razcep. Časovna zahtevnost tega postopka je torej $O(n^2k)$. Za kratke nize, kot so bili tisti pri našem tekmovanju, je to dovolj hitro; mogoče pa je to rešitev še izboljšati.

Vrnimo se k prej omenjeni rekurzivni formuli za $g(n', k')$. V njej med drugim nastopa ocena podniza $a_{i+1} \ldots a_{n'}$; spomnimo se, da lahko to oceno izrazimo kot $f(a_{i+1} \ldots a_{n'}) = \min\{s_c[i] - s_c[n'] : 0 \le c \le 1\}$. Če nesemo to v formulo za $g(n', k')$, dobimo:

$$g(n', k') = \min\{g(i, k' - 1) + s_c[i] - s_c[n'] : k' - 1 \le i < n', 0 \le c \le 1\}.$$

Ker je c neodvisen od i, lahko pri vsakem c posebej izračunamo minimum po vseh i in nato vzamemo minimum po obeh možnih c:

$$g(n', k') = \min\{\min\{g(i, k' - 1) + s_c[i] - s_c[n'] : k' - 1 \le i < n'\} : 0 \le c \le 1\}.$$

Opazimo lahko, da je $s_c[n']$ neodvisen od i, zato ga lahko nesemo ven iz notranjega min:

$$g(n', k') = \min\{\min\{g(i, k' - 1) + s_c[i] : k' - 1 \le i < n'\} - s_c[n'] : 0 \le c \le 1\}.$$

Kaj se zgodi, če namesto n' vzamemo $n' + 1$, torej če razbijamo še za en znak daljši podniz? V notranjem min so posamezni členi enaki kot prej, le da se jim pridruži še en nov člen (za $i = n'$, ki zdaj ustreza omejitvi $i < n' + 1$, prej pa ni ustrezal omejitvi $i < n'$). Koristno je torej, če si za vsak c posebej hranimo vrednost notranjega min; ko se premaknemo z n' na $n' + 1$, lahko obe vrednosti poceni popravimo (saj moramo le pogledati, če je novi člen $g(n', k'-1)+s_c[n']$ kaj manjši od dosedanjega minimuma), nato pa izračunamo $\min\{\ldots\} - s_c[n']$ za oba c-ja in pogledamo, kateri je manjši. Tako imamo pri vsakem (n', k') le $O(1)$ dodatnega dela in časovna zahtevnost celotnega postopka je le še $O(nk)$.

Viri nalog: mafijski semenj — Nino Bašić; kompresija — Primož Gabrijelčič; kontrolne vsote — Boris Gašperin; delni izid — Tomaž Hočevar; znajdi.se — Jurij Kodre; polaganje plošč, nurikabe — Mitja Lasič; analiza signala — Matjaž Leonardis; dva od petih — Mark Martinec; kodiranje — Mark Martinec in Janez Brank; it's raining cubes, strahopetni Hektor, Golovec — Jure Slak; trgovanje z zrni — Patrik Zajec; razcep niza — Janez Brank.

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