8. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2013

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.");
   }
}
```

8. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2013

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 v datoteki. 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 samodejno shranjuje in na zaslonu ti bo pisalo, kdaj se bo shranjevanje naslednjič zgodilo. 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.) **Zaradi varnosti 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. Vandali

Vandal bi iz javne table, ki vsebuje dolg napis T_1, rad naredil nek krajši napis T_2 (niza T_1 in T_2 sta dana) tako, da s flomastrom zakrije določene črke. Recimo:

T_2 : `dol z vlado`
T_1 : `Še dobro, da lahko z volitvami vplivamo na dobrobit naroda.`
izpis : `###do########l#### z v###########l#######a#d###o###########`

Napiši podprogram Vandal(T1, T2), ki dobi niza T_1 in T_2 ter izpiše vandalizirano različico niza T_1, v kateri so nekateri znaki spremenjeni v `#` tako, da preostali znaki tvorijo ravno niz T_2.

Predpostavi, da se T_2 zagotovo pojavlja nekje znotraj napisa T_1; če je pojavitev več, je vseeno, katero uporabiš. (Na primer: pri T_2 = `abc` in T_1 = `cabdbc` lahko izpišemo `#ab##c` ali pa `#a##bc`.)

Tvoj podprogram naj bo takšne oblike:

```
procedure Vandal(T1, T2: string);                { v pascalu }
void Vandal(char *T1, char *T2);                 /* v C/C++ */
void Vandal(string T1, string T2);               // v C++
public static void Vandal(String T1, String T2); // v javi
public static void Vandal(string T1, string T2); // v C#
def Vandal(T1, T2): ...                          # v pythonu
```

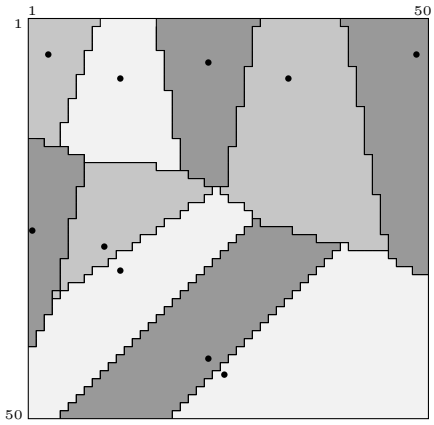
2. Kolera

V devetnajstem stoletju se še ni kaj dosti vedelo o načinu prenašanja nalezljivih bolezni. Leta 1854 so imeli v Londonu velik izbruh črevesne bolezni kolere. Zdravnik John Snow je takrat s svojo analizo pokazal na vzročno zvezo med lokacijo londonskih vodnjakov in lokacijo domovanja obolelih. Izkazalo se je, da so žrtve pile vodo iz istega okuženega vodnjaka na ulici Broad Street. Zdravnik si je narisal zemljevid Londona, vanj vrisal lokacije vodnjakov in za vsak vodnjak z drugo barvo pobarval območje zemljevida, ki je temu vodnjaku najbližje. Ko je vrisal še točke domovanja obolelih, je postala vzročna zveza precej očitna, saj so prebivalci večinoma hodili po vodo k njim najbližjemu vodnjaku.

Nekoliko si poenostavimo zemljevid velemesta in denimo, da nas zanima le območje, ki ga razdelimo na kvadratno mrežo 50×50 celic. V nekaterih celicah te mreže se nahajajo vodnjaki. Vseh vodnjakov je deset, njihove koordinate preberemo z vhodne datoteke: v vsaki vrstici sta dve celi števili, x in y (obe sta z območja od 1 do 50).

Napiši program, ki bo prebral koordinate vodnjakov, potem pa za vsak vodnjak izpisal, kolikšno površino zemljevida pokriva, t.j. kolikšnemu številu celic v tej pravokotni mreži je ta vodnjak najbližji. Za merjenje razdalje med dvema celicama uporabimo Pitagorov izrek: kvadrat razdalje med (x_1, y_1) in (x_2, y_2) je $(x_1 - x_2)^2 + (y_1 - y_2)^2$. Če je od neke celice razdalja do več vodnjakov enaka, je vseeno, h kateremu vodnjaku štejemo tako celico. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `kolera.txt` (kar ti je lažje).

Primer vhodnih podatkov:	Eden od možnih izpisov:
`49 5`	`240`
`33 8`	`420`
`1 27`	`115`
`23 43`	`325`
`25 45`	`452`
`12 8`	`206`
`10 29`	`158`
`3 5`	`97`
`23 6`	`184`
`12 32`	`303`



Prikazan je seveda le eden od številnih možnih izpisov pri teh vhodnih podatkih (saj naloga pravi, da če je neka celica enako oddaljena od dveh ali več vodnjakov, je vseeno, h kateremu jo štejemo). Črne pike na sliki označujejo položaje vodnjakov, osenčeni liki pa kažejo dele mreže, ki so najbližje posameznemu vodnjaku.

3. Kino

Mirko rad hodi v kino (običajnega, z eno samo dvorano), kjer vrtijo filme brez prekinitev enega za drugim. Domov bo šel z avtobusom, vendar pa sovraži stati na postaji in čakati na avtobus. Po vsakem filmu se odloča, ali naj gre na avtobus ali naj pogleda še en film. Pred seboj ima dva seznama:

- vozni red avtobusov: npr. $[(0,40),(1,50),(2,30),(3,30),(4,30),(5,35),(6,5)]$; to so pari (ure, minute), ki povedo, kdaj odpeljejo avtobusi s postaje pred kinom; pri tem se čas meri od nekega izbranega začetnega trenutka, zato lahko število ur sčasoma tudi preseže 24;
- seznam trajanj filmov, ki jih vrtijo v kinu: na primer $[(1,30),(2,10),(1,40),(0,50)]$, torej spet v obliki parov (ure, minute).

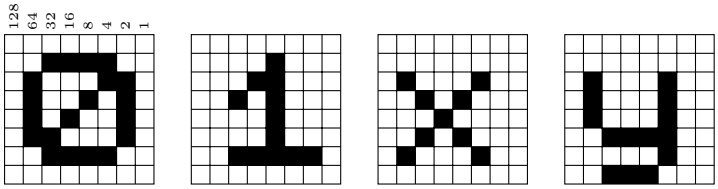
Prvi film se začne ob trenutku $(0,0)$. Med filmi ni prekinitve. Mirko pogleda vedno vsaj prvi film. Vhodni podatki so taki, da zagotovo lahko dobi avtobus za domov, če pogleda samo prvi film. Rad bi minimiziral čas čakanja na postaji (to med drugim tudi pomeni, da seveda noče ostati v kinu tako dolgo, da bi zamudil še zadnji avtobus). **Opiši postopek**, ki izračuna, koliko filmov naj pogleda. Mirko lahko pride od kina do avtobusne postaje v trenutku, tako da lahko ujame avtobus tudi, če le-ta odpelje ob istem času, ob katerem se konča zadnji film, ki si ga je ogledal.

Pri zgornjem primeru se na primer izkaže, da je najbolje, če pogleda tri filme. V tem primeru mora čakati na avtobus 15 minut; če bi pogledal en film, bi moral čakati 20 minut, po drugem filmu bi moral čakati kar 50 minut, po četrtem filmu pa bi celo zamudil zadnji avtobus (in to za 5 minut).

4. Igličnі tiskalnik

Na nekaterih starejših računalnikih je bil vsak znak predstavljen s črno-belo sliko velikosti 8×8 točk. Vsaka slika je bila v pomnilniku predstavljena s tabelo 8 bajtov, pri čemer je n-ti bajt vseboval opis n-te vrstice (gledano od zgoraj navzdol). Vsaka točka (piksel) v vrstici je imela svojo utež, uteži posameznih točk od desne proti levi so bile potence števila 2, torej 1, 2, 4, 8, 16, 32, 64, 128. Vrstica je bila predstavljena z vsoto uteži črnih točk.

Znaki `0`, `1`, `x` in `y` so recimo predstavljeni takole:



V tretji vrstici (od zgoraj navzdol) slike znaka `0` so druga, tretja in sedma točka (od desne proti levi) črne. Skupna vsota njihovih uteži je $2+4+64 = 70$, zato ima tretji bajt v opisu znaka `0` vrednost 70. Znak `0` je bil torej predstavljen z naslednjim zaporedjem 8 bajtov: 0, 60, 70, 74, 82, 98, 60, 0.

Napiši podprogram Izpisi(s), ki izpiše bitno sliko niza `s`. Pri tem naj predpostavi, da so bitne slike vseh možnih znakov že pripravljene in shranjene v globalni spremenljivki znaki; to je tabela 256×8 števil, pri čemer je prvi indeks številka znaka (od 0 do 255), drugi indeks pa številka vrstice (od 0 do 7). Podprogram naj niz izpiše tako, da vse črne točke v bitnih slikah izpiše kot znak „`#`“, vse bele točke pa kot pike „`.`“. Tako bi podprogram Izpisi na primer izpisal niz `001xy` kot

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

Še deklaracije v raznih programskih jezikih:

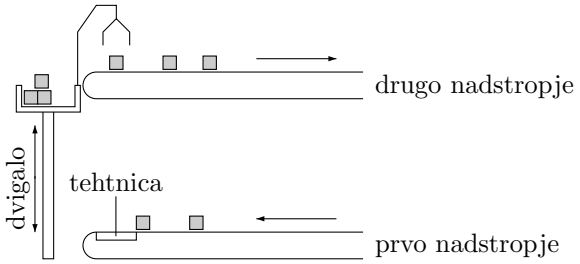
```
unsigned char znaki[256][8];                    /* v C/C++ */
void Izpisi(char *s);                           /* v C/C++ */
void Izpisi(string s);                          // v C++

var znaki: array [0..255, 0..7] of byte;        { v pascalu }
procedure Izpisi(s: string);

public static byte[,] znaki = new byte[256, 8]; // v C#
public static void Izpisi(string s);

public static byte[][] znaki = new byte[256][8]; // v javi
public static void Izpisi(String s);

znaki = [[...], [...], ...]                     # v pythonu
def Izpisi(s): ...
```

Nasvet: v pythonu in pascalu lahko številsko kodo znaka `c` dobiš tako, da pokličeš standardno funkcijo ord(c).

5. Dvigalo

V dvonadstropni tovarni robotizirano dvigalo skrbi za prenos zabojev med nadstropji. Zaboji prihajajo po tekočem traku na tehtnico, ločen sistem pa trak zaustavi, kadar je na tehtnici zaboj, ter ga požene, ko se izprazni.

Del dvigala z nosilnostjo 500 kg je robotska roka, ki v spodnjem nadstropju prelaga zaboje s tehtnice v dvigalo, v zgornjem nadstropju pa jih razlaga iz dvigala na trak, po katerem se avtomatsko odpeljejo v nadaljnjo obdelavo.



Napiši program, ki se vrti v neskončni zanki in upravlja dvigalo. Poskrbeti moraš, da masa zabojev na dvigalu nikoli ne preseže nosilnosti dvigala. Predpostavi, da je dvigalo na začetku prazno in parkirano v prvem (torej spodnjem) nadstropju ter da imajo zaboji maso od 1 do 500 kg (vključno). Želimo, da je število premikov med nadstropji čim manjše. (Dvigalo torej vedno naložimo kar najbolj.)

Na voljo imaš naslednje podprograme (funkcije):

- **int** Stehtaj() — Vrne maso zaboja na tehtnici. Če na tehtnici ni zaboja, funkcija počaka, dokler se zaboj ne pojavi, nato ga stehta in vrne rezultat.
- **void** Nalozi() — Z robotsko roko prenese zaboj s tehtnice na dvigalo. Če pri tem masa zabojev na dvigalu preseže nosilnost, se dvigalo pokvari. Podprogram se vrne šele, ko je zaboj naložen. Če je tehtnica prazna ali pa je dvigalo v drugem nadstropju namesto v prvem, podprogram ne naredi ničesar in se takoj vrne.
- **void** Razlozi() — Z robotsko roko prenese enega od zabojev z dvigala na trak v drugem nadstropju. Če je bil trak še zaseden (ker je bil na njem nek prejšnji zaboj), Razlozi najprej počaka, da se prejšnji zaboj odpelje. Če je dvigalo v prvem nadstropju namesto v drugem, Razlozi ne naredi ničesar in se takoj vrne.
- **void** OdpeljiDvigalo(**int** StNadstropja) — Odpelje dvigalo v izbrano nadstropje. Podprogram se vrne šele, ko dvigalo prispe na cilj. Parameter StNadstropja ima lahko vrednost 1 ali 2.

8. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2013

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 v datoteki. 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 samodejno shranjuje in na zaslonu ti bo pisalo, kdaj se bo shranjevanje naslednjič zgodilo. 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.) **Zaradi varnosti 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. Binarni sef

Binarni sef ima na številčnici dve tipki: 0 in 1. Če ima sef šifro 011, se bo odprl takrat, ko bodo zadnje tri pritisnjene tipke zaporedno 0, 1 in 1. Če ne poznamo šifre, lahko poskusimo sef odpreti tako, da vtipkamo vseh 8 trimestnih kombinacij. Vendar gre tudi hitreje — če vtipkamo na primer zaporedje 0001011100, se bodo vrata zagotovo odprla, saj se v tem zaporedju pojavljajo kot podnizi kombinacije 000, 001, 010, 101, 011, 111, 110, 100 — kar je ravno vseh osem kombinacij. **Napiši podprogram** `Sef(s, n)`, ki bo za dano zaporedje `s` in znano dolžino šifre (`n` znakov) preveril, ali zaporedje `s` zagotovo odpre sef. Predpostaviš lahko, da je `n` (dolžina šifre) največ 12, zaporedje `s` pa je dolgo največ 10 000 znakov.

Tvoj podprogram naj bo takšne oblike:

```
function Sef(s: string; n: integer): boolean;    { v pascalu }
bool Sef(char *s, int n);                        /* v C/C++ */
bool Sef(string s, int n);                       // v C++
public static boolean Sef(String s, int n);      // v javi
public static bool Sef(string s, int n);         // v C#
def Sef(s, n): ...                               # v pythonu; naj vrne True/False
```

2. Sumljiva imenovanja

Pravkar ustanovljena Komisija za nadzor zaposlovanja v javni upravi dobiva številne vloge za oceno zakonitosti imenovanja novih uradnikov. Vsaka vloga je sestavljena iz treh stvari: seznam kvalifikacij, ki so zahtevane za to delovno mesto; seznam kvalifikacij, ki jih je imel človek, ki je to delovno mesto dobil; nato pa še seznami kvalifikacij vseh ostalih kandidatov, ki so se potegovali za to delovno mesto, vendar ga niso dobili.

Komisija mora za vsako imenovanje ugotoviti, ali je *zakonito*, *sumljivo* ali celo *nezakonito*. Pri tem uporablja naslednja pravila:

- imenovanje je *nezakonito*, če kandidat nima vseh kvalifikacij, zahtevanih na tem delovnem mestu;
- imenovanje je *sumljivo*, če kandidat sicer ima vse potrebne kvalifikacije, vendar obstaja boljši kandidat; s tem je mišljen tak kandidat, ki ima vse kvalifikacije kot tisti, ki je bil imenovan, in še kakšno zraven;
- sicer je imenovanje *zakonito*.

Napiši program, ki prebere vlogo in izpiše, ali je bilo imenovanje nezakonito, sumljivo ali zakonito.

Vhodni podatki: privzemimo, da obstaja natanko n (za nek $n \leq 32$) možnih kvalifikacij. Tako lahko seznam zahtevanih kvalifikacij podamo kar z nizom oblike: $s = c_1c_2c_3 \ldots c_{n-1}c_n$, kjer je lahko vsaka izmed črk c_i le `D` ali `N` in kjer črka `D` na i-tem mestu pomeni, da je i-ta kvalifikacija zahtevana, črka `N` pa, da ni. Na primer, niz `DNDNDNNNDDDNNDNDNDNNDNNNDDDNNDNN` predstavlja delovno mesto, kjer so zahtevane kvalifikacije št. 1, 3, 5, 9, 10, ..., kvalifikacije št. 2, 4, 6, 7, 8, ... pa ne. Podobno predstavimo seznam kvalifikacij, ki jih imajo kandidati. Tako kot prej črka `D` na i-tem mestu pomeni, da kandidat i-to kvalifikacijo ima, črka `N` pa, da je nima.

Tvoj program lahko vhodne podatke bere s standardnega vhoda ali pa iz datoteke `vhod.txt` (kar ti je lažje). V prvi vrstici je niz, ki predstavlja zahtevane kvalifikacije za delovno mesto; druga vrstica predstavlja kvalifikacije kandidata, ki je bil zaposlen; vse naslednje vrstice pa predstavljajo kvalifikacije vseh ostalih kandidatov.

Tvoja rešitev naj deluje tudi v primerih, ko je število vrstic v vhodni datoteki zelo veliko — preveliko, da bi celotno vsebino vhodne datoteke naenkrat shranili v pomnilnik.

Predpostaviš lahko, da obstaja funkcija `vStevilo(s)`, ki spremeni niz kvalifikacij `s` v število, ki ima v binarnem zapisu na i-tem mestu 1, če je imel niz kvalifikacij tam črko `D` in 0, če je bila tam črka `N`. Ni pa nujno, da to funkcijo uporabiš (nalogo lahko čisto dobro rešiš tudi brez nje).

Izhodni podatki: izpiši, ali je bilo imenovanje nezakonito, sumljivo ali zakonito.

Primer vhodne datoteke:

```
NNDDDDDDNDDNDNNDNDDDNNDNDNNNDDNN
DDDDDDDDNDDDDDDDDDDDNDDDDDDDDDNN
DDDDDDDDNDDDDDDDDDDDDDDDDDDDDND
NDNNDDDNNNNNNDNNNNDNDNNDDNNDNNNN
NNNNNDNDNDDNNDDNDNNNDNDNDDNDDDDD
NNNDDNDNDNDDDNDDDNNNNDDDDDNNNDND
NDDDDNNNDNNNNDDNNNNNNDNNNDNNDNDN
NNDDDNDDNDDNNDDDDDDNDDDDNNNDDDNN
NNNNNDNDNDDDNDNDDNDNDNNNNDNDNNNN
NDDDNNNDDDNNDNDDNNNDNDDDNNDNNNND
NNNNDNDDDDDDNNDNNDDDDDDNDNDNNDDN
```

Pripadajoč izpis:

```
sumljivo
```

(Pojasnilo: drugi kandidat ima vse kvalifikacije, ki jih ima prvi, in še dve taki, ki ju prvi nima, zato je imenovanje sumljivo.)

3. Dekodiranje nizov

Dan je naslednji postopek za kodiranje besedila (niza znakov):

- Niz začnemo brati od začetka, dokler ne pridemo do prvega samoglasnika. Nato vse prebrane znake, vključno s tistim samoglasnikom, napišemo v obratnem vrstnem redu. Ob tem si shranimo podatek, koliko znakov smo zamenjali pri tej prvi menjavi.
- Sedaj preberemo vse znake od prvega samoglasnika dalje (toda brez tega samoglasnika), dokler ne pridemo do drugega samoglasnika. Zopet vse prebrane znake, vključno z drugim samoglasnikom, napišemo v obratnem vrstnem redu. S tem je drugi samoglasnik prišel na drugo mesto v nizu, takoj za prvim samoglasnikom. Shranimo podatek, koliko znakov smo zamenjali pri tej drugi menjavi.
- Postopek ponavljamo, dokler ne pridemo do konca besedila.

Primer: oglejmo si postopek kodiranja niza „`SOBA Z RAZGLEDOM NA VRT`“. Za samoglasnike štejejo znaki `A`, `E`, `I`, `O` in `U`. V vsaki vrstici je podčrtan tisti del niza, ki ga bomo v tem koraku obrnili; številka na desni pa je dolžina tega dela.

```
SOBA Z RAZGLEDOM NA VRT   (2)
OSBA Z RAZGLEDOM NA VRT   (3)
OABS Z RAZGLEDOM NA VRT   (7)
OAAR Z SBZGLEDOM NA VRT  (10)
OAAELGZBS Z RDOM NA VRT  (11)
OAAEODR Z SBZGLM NA VRT  (14)
OAAEOAN MLGZBS Z RD VRT
```

Opiši postopek za dekodiranje tako zakodiranega niza. Postopek torej kot vhodne podatke dobi kodirani niz (v zgornjem primeru: `OAAEOAN MLGZBS Z RD VRT`) in seznam s številom zamenjanih znakov pri vsaki menjavi (v zgornjem primeru: 2, 3, 7, 10, 11, 14). Predpostavi, da niz vsebuje le velike črke angleške abecede (od `A` do `Z`) in presledke.

Dekodiranje za zgornji primer:

```
(14)  OAAEOAN MLGZBS Z RD VRT
(11)  OAAEODR Z SBZGLM NA VRT
(10)  OAAELGZBS Z RDOM NA VRT
 (7)  OAAR Z SBZGLEDOM NA VRT
 (3)  OABS Z RAZGLEDOM NA VRT
 (2)  OSBA Z RAZGLEDOM NA VRT
      SOBA Z RAZGLEDOM NA VRT
```

4. Silhuete

Neko mesto ima obliko pravokotne mreže, razdeljene na $w \times h$ parcel (w stolpcev, h vrstic). Na vsaki parceli stoji stolpnica, ki ima obliko kvadra. V tlorisu so si vsi ti kvadri enaki, razlikujejo pa se po višini; na parceli v r-ti vrstici in c-tem stolpcu stoji nebotičnik z višino a_{rc} (to je celo število, večje od 0).

Mesto si (recimo s fotoaparatom) „ogledamo“ od spredaj in od strani. Na tak način dobimo samo podatek o najvišji stolpnici v vsaki vrstici in vsakem stolpcu. Recimo, da je x_c višina najvišje stolpnice v c-tem stolpcu, y_r pa višina najvišje stolpnice v r-ti vrstici. Iz teh podatkov (torej $x_1, x_2, \ldots, x_w$ in $y_1, y_2, \ldots, y_h$) sicer v splošnem ne moremo natančno določiti višin posameznih stolpnic (torej posameznih vrednosti a_{rc}), lahko pa preverimo, ali podatki sploh predstavljajo možno konfiguracijo in izračunamo eno od možnih postavitev.

Napiši program, ki prebere vhodne podatke in izpiše eno od možnih postavitev ali pa „`taka postavitev ne obstaja`“, če podatki niso veljavni. Vhodni podatki so zapisani v treh vrsticah in sicer takole:

$$\begin{array}{cccc} w & h & & \\ x_1 & x_2 & \ldots & x_w \\ y_1 & y_2 & \ldots & y_h \end{array}$$

Tvoj program jih lahko bere s standardnega vhoda ali pa iz datoteke `silhuete.txt` (kar ti je lažje). Predpostavi, da velja $w \leq 1000$ in $h \leq 1000$.

Primer: recimo, da imamo mrežo 4×3 stolpnic z višinami

9	2	3	2
2	1	6	1
6	3	6	9

Maksimalne višine po stolpcih so torej (od leve proti desni) 9, 3, 6, 9; po vrsticah pa (od zgoraj navzdol) 9, 6, 9. Program bi v tem primeru kot vhodne podatke dobil:

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

Nekaj možnih razporedov pri teh vhodnih podatkih (program bi torej lahko izpisal katerega koli izmed njih, obstaja pa še veliko drugih, ki bi bili prav tako dobri):

```
9 2 3 2          9 3 6 9          9 1 2 1          4 3 5 9
2 1 6 1          5 3 6 6          2 3 1 6          4 3 6 5
6 3 6 9          9 2 6 9          9 1 6 9          9 2 2 9
```

Primer vhodnih podatkov, pri katerih veljavna predstavitev ne obstaja:

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

5. Ribič

Nadebuden ribič se je odpravil k potoku, v katerem namerava z mrežo dolžine d uloviti točno k rib, ki jih bo prinesel domov za kosilo. Voda je kristalno čista, zato točno ve, kje se nahaja katera od n rib. Za potrebe naloge lahko potok predstavimo z ravno črto, ribe pa s točkami x_i na njej. Mrežo bo vrgel tako, da se bo začela na neki celoštevilski koordinati x (ki je lahko tudi negativna). Pri tem bo ujel vse ribe, ki se nahajajo na koordinatah od vključno x do vključno $x + d - 1$. **Opiši postopek**, ki izračuna, na koliko načinov lahko ribič vrže mrežo, da bo ujel točno k rib.

Predpostavi, da so dolžina d in koordinate x_i cela števila v razponu od 1 do 10^9, k in n pa sta celi števili med 1 in 10^6. Zato naj bo tvoj postopek čim bolj učinkovit (bolj učinkovite rešitve dobijo več točk). Koordinate x_i so že podane v naraščajočem vrstnem redu in nobeni dve ribi se ne nahajata na isti koordinati.

Primer: če imamo $n = 6$ rib na koordinatah $1, 5, 6, 10, 12, 15$ in mrežo dolžine $d = 8$, s katero bi radi ujeli natanko $k = 3$ ribe, potem je primernih položajev mreže (torej primernih vrednosti x) devet (to so $-1, 0, 1, 3, 4, 6, 8, 9, 10$). Položaj $x = 5$ na primer ni primeren, ker pri njem ujamemo 4 ribe, ne 3; položaja $x = 2$ in $x = 7$ nista primerna, ker takrat ujamemo le 2 ribi, ne 3.

8. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2013

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Delovno okolje

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

Na vsakem računalniku imaš na voljo imenik `U:\_Osebno`, v katerem lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C# ali java, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz JDK 1.7 in s prevajalnikom za C# iz Visual Studia 2008. Za delo lahko uporabiš `FP` oz. `ppc386` (Free Pascal), `GCC/G++` (GNU C/C++ — command line compiler), `javac` (za javo 1.7), Visual Studio 2010 in druga orodja.

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `RTK.EXE`, ki ga lahko uporabiš za oddajanje svojih rešitev. Tukaj si lahko tudi ogledaš anonimizirane rezultate ostalih tekmovalcev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka, brez šumnikov).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

`rtk` *imenaloge*.`pas`
`rtk` *imenaloge*.`c`
`rtk` *imenaloge*.`cpp`
`rtk` *ImeNaloge*.`java`
`rtk` *ImeNaloge*.`cs`

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Datoteka z izvorno kodo, ki jo oddajaš, ne sme biti daljša od 30 KB. Na spletni strani boš dobil za vsak testni primer obvestilo o tem, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund ali pa porabil več kot 200 MB pomnilnika, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

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

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. Nato se točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N - 1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

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

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

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

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

Primer vhodne datoteke:

```
123 456
```

Ustrezna izhodna datoteka:

```
5790
```

Primeri rešitev (dobiš jih tudi kot datoteke na `http://rtk/`):

- V pascalu:

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

- V C-ju:

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

- V C++:

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

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

- V javi:

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

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

- V C#:

```
using System.IO;

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

8. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2013

NALOGE ZA TRETJO SKUPINO

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

1. Moderna umetnost (`umetnost.in`, `umetnost.out`)

Slikar bo na prihajajoči razstavi predstavil zbirko slik, ki bodo v celoti sestavljene iz raznobarvnih navpičnih črt različnih širin. Platno je razdelil na n enako širokih stolpcev in določil, kakšne barve mora biti kateri od njih. Za slikanje bo uporabil kar slikopleskarski valj, s katerim vsakič pobarva k sosednjih stolpcev (vse z isto barvo). Ker se mu mudi, bi rad svoje izdelke končal v čim manjšem številu navpičnih potegov z valjem. Okoli platna je podložil časopisni papir, da ni škode, če z valjem seže preko roba. Prav tako lahko isti stolpec prebarva večkrat. Koliko potegov potrebuje, če slika optimalno?

Vhodna datoteka: v prvi vrstici sta dve celi števili, najprej n in nato k, ločeni s presledkom. Veljalo bo $1 \leq k \leq n \leq 10^6$. Sledi n vrstic, ki opisujejo želene barve posameznih stolpcev (od leve proti desni). Vsaka od teh vrstic vsebuje le eno celo število, to je številka barve ustreznega stolpca. Barve so oštevilčene s števili od 1 do n (pri čemer seveda ni nujno, da se vseh n možnih barv dejansko pojavlja na sliki). Pri 40 % testnih primerov bo veljalo tudi $n \leq 100$.

Izhodna datoteka: vanjo izpiši eno samo celo število, namreč najmanjše število potez z valjem, ki jih slikar potrebuje, da pobarva sliko v skladu z zahtevami naloge.

Primer vhodne datoteke:

```
7 4
2
2
2
6
6
2
2
```

Pripadajoča izhodna datoteka:

```
3
```

Komentar primera: slikar lahko z eno potezo pobarva stolpca barve 6, z drugo potezo pobarva leve tri stolpce in s tretjo še desna dva. Pri tej tretji potezi seže valj preko roba (desno od slike) in pobarva tudi nekaj podloženega časopisnega papirja. Slikar nekatere stolpce pobarva večkrat, vendar je končen videz slike pravilen, ker se barve ne mešajo in je vidna le zadnje nanešena barva.

2. Kompleksnost števil (`kompleksnost.in`, `kompleksnost.out`)

Dano celo število n bi radi izrazili kot rezultat aritmetičnega izraza, v katerem nastopajo le oklepaji, zaklepaji, operatorja + in ·, kot operand pa le število 1. Pri tem + ni dovoljeno uporabljati v primerih, ko bi za izračun takega izraza morali kdaj sešteti dve števili, večji od 1 (+ smemo torej uporabljati le za prištevanje 1). **Napiši program**, ki za dani n ugotovi, kakšno je najmanjše število enic, ki jih potrebujemo, da sestavimo takšen izraz z vrednostjo n.

Primer: $n = 12$ lahko izrazimo kot

$$\begin{array}{cc} 1+1+1+1+1+1+1+1+1+1+1+1 & (12 \text{ enic}) \\ (1+1+1+1+1+1)\cdot(1+1) & (8 \text{ enic}) \\ (1+1+1+1)\cdot(1+1+1) & (7 \text{ enic}) \\ (1+1)\cdot(1+1)\cdot(1+1+1) & (7 \text{ enic}) \\ (1+1+1)\cdot(1+1+1)+1+1+1 & (9 \text{ enic}) \end{array}$$

in še na veliko drugih načinov. Najmanjše število enic, ki jih potrebujemo, da sestavimo izraz z vrednostjo 12, je torej 7.

Primer neveljavnega izraza za $n = 12$:

$$((1+1)\cdot(1+1+1))+1+1+((1+1)\cdot(1+1))$$

Ko ga računamo, namreč iz njega nastane $6+1+1+4$ in ne glede na to, v kakšnem vrstnem redu bi hoteli sešteti te štiri seštevance, bi prej ali slej morali sešteti dve števili, ki bi bili obe večji od 1.

Vhodna datoteka: v prvi vrstici je celo število t, ki pove, za koliko n-jev nas pri tem primeru zanima najmanjše potrebno število enic. Sledi t vrstic, vsaka od njih vsebuje po eno celo število, to je nek n, za katerega nas zanima najmanjše potrebno število enic. Ta števila so podana v padajočem vrstnem redu.

Veljalo bo $1 \le t \le 10$. Za vsak n velja $1 \le n \le 3\,000\,000$. V 80 % testnih primerov bo za vsak n veljalo veljalo tudi $n \le 300\,000$, v 40 % testnih primerov pa celo $1 \le n \le 100$.

Izhodna datoteka: vanjo izpiši t celih števil (vsako v svojo vrstico), ki po vrsti za vsakega od n-jev iz vhodne datoteke povedo najmanjše število enic, ki jih potrebujemo, da po pravilih iz besedila naloge sestavimo aritmetični izraz z vrednostjo n.

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:
`3`	`10`
`22`	`7`
`12`	`5`
`6`	

Komentar: 22 lahko na primer z desetimi enicami računamo takole:

$$(1+(1+1)\cdot(1+(1+1)+1+1))\cdot(1+1)=22.$$

3. Požar (`pozar.in`, `pozar.out`)

Podjetje ima v najemu poslovne prostore, ki so razdeljeni v pravokotno mrežo s h vrsticami in w stolpci. Vsaka celica v tej mreži ustreza eni pisarni. Vsi zaposleni so že odšli na zaslužen nočni počitek, nesreča pa žal ne počiva. Pisarno, ki se nahaja v vrstici y_0 in stolpcu x_0, je zajel močan ogenj, ki se hitro širi na sosednje pisarne. V vsaki sekundi se ogenj z goreče pisarne razširi na sosednje pisarne, ki si z njo delijo katero od štirih sten (ne pa na tiste, ki si z njo delijo le enega od vogalov). V podjetju imajo protipožarni alarm, ki pa se sproži šele, ko gori vsaj k pisarn. **Napiši program**, ki bo izračunal, po koliko sekundah se bo sprožil alarm.

Vhodna datoteka: vsebuje več požarov, za katere mora tvoj program izračunati čas alarma. V prvi vrstici je podano število požarov p. Vsaka naslednja vrtica opisuje požar s števili w, h, x_0, y_0 in k. Veljalo bo $1 \leq p \leq 20$, $1 \leq w \leq 10^9$, $1 \leq h \leq 10^9$, $1 \leq x_0 \leq w$, $1 \leq y_0 \leq h$, $1 \leq k \leq w \cdot h$.

Pri 40 % testnih primerov bosta dimenziji mreže kvečjemu 1000. Poleg tega se bo v 60 % testnih primerov alarm pri vseh požarih oglasil v 10^6 sekundah ali prej.

Izhodna datoteka: za vsak požar izpiši v svoji vrstici število, ki pove, po najmanj koliko sekundah gori vsaj k pisarn.

Primer vhodne datoteke:

```
4
5 3 4 2 10
5 3 4 2 11
4 1 1 1 4
5 4 2 2 12
```

Pripadajoča izhodna datoteka:

```
2
3
3
3
```

Naslednja slika se nanaša na zadnji požar v gornjem primeru vhodne datoteke ($w = 5$, $h = 4$, $x_0 = y_0 = 1$). Slika za vsako celico mreže kaže, koliko sekund po začetku požara se vžge. Vidimo lahko, da po dveh sekundah gori šele 11 celic, po treh pa že 16 celic, zato je pri $k = 12$ pravilni odgovor 3.

2	1	2	3	4
1	0	1	2	3
2	1	2	3	4
3	2	3	4	5

4. Številčenje (`stevilcenje.in`, `stevilcenje.out`)

Zapisati hočemo naravna števila od vključno a do vključno b (seveda v desetiškem zapisu, kot ponavadi). **Napiši program**, ki ugotovi, kolikokrat se v tem številčenju pojavi posamezna števka (od 0 do 9).

Primer: pri $a = 9$, $b = 12$ imamo števila $9, 10, 11, 12$; tu se torej števka 1 pojavi štirikrat, števke 0, 9 in 2 po enkrat, ostale števke pa nikoli.

Vhodna datoteka: vsebuje le eno vrstico, v njej pa sta dve celi števili, najprej a in nato b, ločeni z enim presledkom. Veljalo bo $1 \leq a \leq b \leq 2^{50}$. V 40 % testnih primerov bo veljalo tudi $b \leq 10^6$.

Izhodna datoteka: vanjo izpiši 10 celih števil, vsako v svojo vrstico; ta števila naj po vrsti za vsako od števk od 0 do 9 povedo, kolikokrat se pojavlja v desetiškem zapisu števil od a do b.

Primer vhodne datoteke:

```
9 12
```

Pripadajoča izhodna datoteka:

```
1
4
1
0
0
0
0
0
0
1
```

5. Natrpan urnik (`urnik.in`, `urnik.out`)

Na planetu K-72 v glavni provinci C-01 imajo zelo zanimiv šolski sistem. Njihovo šolsko leto traja zelo dolgo, tam do več milijonov dni; po drugi strani pa imajo le 5 predmetov. Posledica velikega števila dni v letu je tudi to, da imajo čez celo leto zelo veliko testov. VX-48 iz okrožja H-18 je pridna učenka, ki si vsakič, ko pri kakšnem predmetu izve za datum testa, le-tega vestno zapiše v koledar. Tako se njen letni urnik kar polni in polni, njeni skrbni starši pa jo pogosto sprašujejo, koliko testov nekega predmeta ima v določenem obdobju. **Napiši program**, ki ji bo pomagal odgovarjati na takšna vprašanja.

Vhodna datoteka vsebuje zaporedje vnosov v beležko in poizvedb (lahko so poljubno premešane, torej ni nujno, da pridejo vsi vpisi pred vsemi poizvedbami). Da vhodna datoteka ne bo predolga, so vpisi in poizvedbe združeni v skupine (bloke) in vsaka vrstica vhodne datoteke predstavlja bodisi skupino vnosov bodisi skupino poizvedb.

Prva vrstica vhodne datoteke vsebuje dve celi števili, b in D, ločeni s presledkom; b je število skupin, D pa število dni v šolskem letu. Sledi b vrstic, vsaka od njih pa predstavlja bodisi skupino vnosov v beležko bodisi skupino poizvedb.

Vrstica, ki predstavlja skupino vnosov v beležko, se začne z znakom `V`, sledijo pa mu števila n, p, d in s, ločena s presledkom. To pomeni, da gre za skupino n vnosov, vsi se nanašajo na teste pri predmetu p, datumi teh testov pa so d, $d+s$, $d+2s$, …, $d+(n-1)s$.

Vrstica, ki predstavlja skupino poizvedb, se začne z znakom `P`, sledijo pa mu števila n, p, d_1, d_2, s_1 in s_2, ločena s presledkom. To pomeni, da gre za skupino n poizvedb, pri čemer i-ta od teh poizvedb (za vsak i od 1 do n) sprašuje po tem, koliko testov predmeta p je od vključno dne $d_1+(i-1)s_1$ do vključno dne $d_2+(i-1)s_2$.

Vsi datumi, ki nastopajo v vnosih in poizvedbah, so od 1 do D; končni datum vsake poizvedbe je večji ali enak od začetnega datuma tiste poizvedbe; nikoli se ne zgodi, da bi na nek datum večkrat vnesli test istega predmeta.

Dolžina leta je $1 \le D \le 10^6$; skupno število skupin, torej b, je $1 \le b \le 10^6$; za skupno število vseh vnosov in poizvedb (recimo mu N; to je vsota n-jev po vseh b skupinah iz vhodne datoteke) pa velja $N \le 5 \cdot 10^6$. Pri vsaki skupini velja $n \ge 1$, $1 \le p \le 5$, $s > 0$, $s_1 > 0$, $s_2 > 1$.

Pri 20 % testnih primerov bo veljalo $N \le 1000$ in $D \le 1000$; pri 40 % testnih primerov bo veljalo $N \le 10^5$ in $D \le 10^5$; pri 80 % testnih primerov bo veljalo $N \le 10^6$.

Izhodna datoteka: za vsako poizvedbo iz vhodne datoteke izračunaj število testov predmeta p v obdobju, na katerega se nanaša poizvedba, in izpiši vsoto vseh teh števil. (Pri posamezni poizvedbi štejejo seveda le tisti testi, ki so bili do te poizvedbe že vnešeni v beležko, ne pa morebitni testi, ki bodo vnešeni šele kasneje.) Nasvet: ta vsota je lahko večja od 2^{32}, zato je koristno uporabiti 64-bitne celoštevilske tipe (npr. **long long** v C/C++, **long** v C#/javi, `int64` v pascalu).

Primer vhodne datoteke:

```
6 20
V 3 2 3 4
V 2 4 6 3
P 5 4 2 5 2 3
P 3 2 1 7 4 3
V 1 2 6 1
P 3 2 1 7 4 3
```

Pripadajoča izhodna datoteka:

```
14
```

Pojasnilo: prva skupina v vhodni datoteki vnese tri teste za predmet 2 in to na dneve 3, 7 in 11; druga skupina vnese dva testa za predmet 4 in to na dneva 6 in 9; tretja skupina vsebuje pet poizvedb za predmet 4, in to za obdobja $[2,5]$, $[4,8]$, $[6,11]$, $[8,14]$ in $[10,17]$ (odgovori na te poizvedbe so po vrsti: 0, 1, 2, 1, 0); četrta skupina vsebuje tri poizvedbe za predmet 2, in to za obdobja $[1,7]$, $[5,10]$ in $[9,13]$ (odgovori so po vrsti 2, 1, 1); peta vnese še en test za predmet 2, namreč na dan 6; šesta skupina pa vsebuje iste tri poizvedbe kot četrta skupina, le da so odgovori zdaj 3, 2, 1. Skupno je torej v naši vhodni datoteki enajst poizvedb, vsota odgovorov nanje pa je $0+1+2+1+0+2+1+1+3+2+1 = 14$; to je tudi končni rezultat, ki ga moramo izpisati v izhodno datoteko.

8. tekmovanje ACM v znanju računalništva za srednješolce

23. marca 2013

REŠITVE NALOG ZA PRVO SKUPINO

1. Vandali

Iskanje podniza T_2 znotraj niza T_1 si lahko predstavljamo takole: najprej moramo poiskati kakšno pojavitev prve črke niza T_2 v T_1; nato moramo nekje kasneje v T_1 poiskati kakšno pojavitev druge črke niza T_2; in tako naprej. Spodnji podprogram se po nizu T_1 sprehaja z zanko **while**, medtem pa kazalec T2 kaže na tisti znak podniza, ki ga trenutno iščemo v T_1. Ko tak znak najdemo, ga izpišemo in se premaknemo naprej po T_2; namesto ostalih znakov niza T1 pa izpisujemo #.

```
#include <stdio.h>

void Vandal(char *T1, char *T2)
{
  while (*T1)
  {
    if (*T1 == *T2) putchar(*T2++);
    else putchar('#');
    T1++;
  }
}
```

2. Kolera

Z gnezdenima zankama po x in y preglejmo vse točke in za vsako poiščimo najbližji vodnjak. V ta namen uporabimo še eno zanko, ki gre po vseh vodnjakih in za vsakega izračuna razdaljo do (x, y). To primerja z razdaljo do najbližjega doslej znanega vodnjaka (dNaj) in če je trenutni vodnjak še bližji, si ga zapomnimo (v spremenljivki vNaj). Na koncu te zanke vemo, kateri vodnjak je res najbližji, in povečamo za 1 njemu pripadajočo vrednost v tabeli povrsina. Na koncu se moramo le še sprehoditi z zanko po tej tabeli in jo izpisati.

```
#include <stdio.h>
#define N 50 /* velikost mreže */
#define StVodnjakov 10

int main()
{
  int povrsina[StVodnjakov], vx[StVodnjakov], vy[StVodnjakov];
  int x, y, d, v, dNaj, vNaj;

  /* Preberimo koordinate vodnjakov. */
  for (v = 0; v < StVodnjakov; v++) {
    scanf("%d %d", &vx[v], &vy[v]);
    povrsina[v] = 0; }

  /* Za vsako točko poiščimo najbližji vodnjak. */
  for (y = 1; y <= N; y++) for (x = 1; x <= N; x++)
  {
    for (v = 0; v < StVodnjakov; v++) {
      /* Izračunajmo razdaljo od (x, y) do v-tega vodnjaka. */
      d = (x - vx[v]) * (x - vx[v]) + (y - vy[v]) * (y - vy[v]);
      /* Če je bližji od doslej najbližjega vodnjaka, si ga zapomnimo. */
      if (v == 0 || d < dNaj) vNaj = v, dNaj = d; }
    povrsina[vNaj]++;
```

```
  }

  /* Izpišimo rezultate. */
  for (v = 0; v < StVodnjakov; v++) printf("%d\n", povrsina[v]);
  return 0;
}
```

3. Kino

Za vsak film bi radi ugotovili, kateri je najzgodnejši avtobus, ki pripelje po koncu tega filma (če sploh obstaja kakšen tak avtobus); ko bomo vedeli to, bomo tudi zlahka izračunali, kako dolgo bi morali čakati nanj. Pri tem nam ni treba zaporedja avtobusov pregledovati vsakič od začetka, ampak lahko kar nadaljujemo pri tistem avtobusu, kjer smo se ustavili pri prejšnjem filmu (saj je jasno, da tisti avtobusi, ki pridejo pred zaključkom k-tega filma, pridejo tudi pred zaključkom $(k+1)$-vega filma). Ko za nek film najdemo prvi naslednji avtobus, lahko izračunamo, kako dolgo bomo čakali nanj, najmanjši čas čakanja pa si zapomnimo (skupaj s filmom, pri katerem smo ga dosegli; spodnji postopek ima v ta namen spremenljivki $f^\star$ in $c^\star$).

Zapišimo naš postopek še s psevdokodo; čase prihodov avtobusov bomo označili z $p_1, p_2, \ldots, p_n$, trajanja filmov pa z $d_1, d_2, \ldots, d_k$. (Za lažje računanje s časi je koristno, če čase iz parov (ure, minute) sproti preračunavamo v minute.)

```
t := 0; a := 1; f* := 1;
for f := 1 to k:
  t := t + d_f;
  (* Film f se konča ob času t. Kateri je prvi naslednji avtobus? *)
  while a ≤ n:
    if p_a < t then a := a + 1 else break;
  if a ≤ n:
    (* Našli smo avtobus a; nanj bomo čakali t − p_a minut.
       Je to najmanjši čas čakanja doslej? *)
    if f = 1 or t − p_a < c*:
      f* := f; c* := t − p_a;
```

Na koncu tega postopka je v $f^\star$ število filmov, ki si jih moramo ogledati, da bomo morali na avtobus čakati najmanj časa.

4. Igličnі tiskalnik

Zahtevani izpis najlažje pripravimo po vrsticah. Posamezno vrstico izpisa dobimo tako, da gremo po vrsti po vseh znakih niza `s` in za vsak znak izpišemo ustrezno vrstico slike tega znaka (ki jo dobimo v tabeli `znaki`). Ostane nam še vprašanje, kako iz byta, s katerim je opisana vrstica znaka v tabeli `znaki`, ugotovimo, kateri piksli morajo biti prižgani, kateri pa so ugasnjeni. Še najlažje je, če uporabimo operatorje za delo z biti; bit 7 opisuje stanje najbolj levega piksla, bit 6 opisuje stanje drugega piksla z leve in tako naprej. Da preverimo, če je bit b prižgan, lahko na primer zamaknemo naš byte za b bitov navzdol (z operatorjem $\gg$) in nato preverimo, če v rezultatu prižgan bit 0 (po operaciji `& 1` nam ostane od števila le bit 0, torej moramo le še preveriti, če je ta rezultat ne ničeln).

```
#include <stdio.h>

unsigned char znaki[256][8];

void Izpisi(char *s)
{
  int y, x, i; unsigned char znak, vrsticaZnaka;
  for (y = 0; y < 8; y++)
  {
    /* Sprehodimo se po vseh znakih niza. */
    for (i = 0; s[i]; i++) {
      znak = (unsigned char) s[i];
      vrsticaZnaka = znaki[znak][y];
```

```
      /* V vrsticaZnaka je zdaj opis y-te vrstice znaka s[i]. */
      for (x = 0; x < 8; x++)
        if ((vrsticaZnaka >> (7 - x)) & 1) putchar('#'); else putchar('.'); }
    putchar('\n'); /* Izpišimo še znak za konec vrstice. */
  }
}
```

5. Dvigalo

Program v zanki tehta zaboje in jih dodaja v dvigalo; pri tem si v spremenljivki stZabojev zapomni, koliko zabojev je že naložil na dvigalo, v skupnaMasa pa vsoto njihovih mas. Skupno maso potrebujemo zato, da lahko preverimo, ali smemo naslednji zaboj še naložiti na dvigalo ali pa bi s tem že presegli nosilnost. Če naslednjega zaboja ne moremo več dodati na dvigalo, odpeljemo dvigalo v drugo nadstropje in raztovorimo vse zaboje (kličemo funkcijo Razlozi tolikokrat, kolikor je zabojev v dvigalu, torej stZabojev). Nato le še odpeljemo dvigalo nazaj v prvo nadstropje in že smo pripravljeni na natovarjanje novih zabojev. Opisani postopek je zato ovit še v eno (neskončno) zanko.

```
#include <stdbool.h>
#define Nosilnost 500

int main()
{
  int stZabojev = 0, skupnaMasa = 0, m;
  while (true)
  {
    /* Natovarjajmo zaboje, dokler se le da. */
    while (true)
    {
      m = Stehtaj();
      /* Naslednji zaboj ima maso m, ali ga lahko dodamo v dvigalo? */
      if (skupnaMasa + m > Nosilnost) break;
      /* Naložimo ga na dvigalo in si zapomnimo novo skupno maso. */
      Nalozi();
      stZabojev++; skupnaMasa += m;
    }
    /* Odpeljimo dvigalo v drugo nadstropje, ga raztovorimo in nato zapeljimo
       nazaj v prvo nadstropje. */
    OdpeljiDvigalo(2);
    while (stZabojev > 0) { Razlozi(); stZabojev--; }
    skupnaMasa = 0;
    OdpeljiDvigalo(1);
  }
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Binarni sef

Ker ima vsaka števka v kombinaciji dve možni vrednosti (0 in 1), obstaja skupno 2^n možnih n-mestnih kombinacij. Pri $n = 12$ je to na primer $2^{12} = 4096$; lahko si torej privoščimo tabelo 2^n logičnih vrednosti (**bool** oz. boolean), v kateri bomo za vsako možno n-mestno kombinacijo hranili podatek o tem, ali smo jo doslej v našem vhodnem nizu s že videli ali ne.

Ko se premikamo naprej po nizu s, lahko vse n-mestne kombinacije, ki se pojavljajo v njem kot podnizi, izračunamo takole: recimo, da smo trenutno na k-tem mestu v nizu s in da imamo trenutni podniz, $s[k-n+1]\ldots s[k]$, predstavljen v n-bitnem številu x (pri čemer bit b vsebuje vrednost števke $s[k-b]$). Če zdaj zamaknemo x za en bit v levo, bo dosedanji podniz $s[k-n+1]\ldots s[k]$ namesto v bitih od 0 do $n-1$ zapisana v bitih od 1 do n; če nato bit n ugasnemo, nam v bitih od 1 do $n-1$ ostane opis podniza $s[k-n+2]\ldots s[k]$; in če zdaj v bit 0 vpišemo vrednost $s[k+1]$, imamo zdaj v bitih od

0 do $n-1$ ravno opis podniza $s[k-n+2]\ldots s[k+1]$ — to pa je ravno naslednji podniz, ki nas bo zanimal (ko se premaknemo s k na $k+1$).

```
#include <stdbool.h>
#define MaxN 12

bool Sef(char *s, int n)
{
  bool prisotna[1 << MaxN];
  int x, i, stPrisotnih = 0;
  for (x = 0; x < (1 << n); x++) prisotna[x] = false;
  for (i = 0, x = 0; s[i]; i++)
  {
    x <<= 1; /* Zamaknimo x za eno mesto gor. */
    x &= ~(1 << n); /* Ugasnimo bit n. */
    if (s[i] == '1') x |= 1; /* Če je treba, prižgimo bit 0. */
    if (i < n - 1) continue; /* Mogoče še nimamo n bitov. */
    /* Označimo, da je kombinacija x prisotna. */
    if (! prisotna[x]) prisotna[x] = true, stPrisotnih++;
  }
  return stPrisotnih == (1 << n);
}
```

2. Sumljiva imenovanja

Nalogo lahko rešimo zelo elegantno, če na vsak niz kvalifikacij gledamo kot na 32-bitno celo število (kot pravi besedilo naloge, lahko celo predpostavimo, da imamo pri roki funkcijo vStevilo za pretvorbo nizov v števila). Recimo, da število z predstavlja zahtevane kvalifikacije, k pa kvalifikacije nekega kandidata. Bit b je torej v z prižgan, če je zahtevana kvalifikacija b, v k pa je ta bit prižgan, če kandidat kvalifikacijo b dejansko ima.

Če hočemo preveriti, ali ima kandidat vse zahtevane kvalifikacije, moramo torej preveriti, ali so v k prižgani vsi tisti biti, ki so prižgani v z (poleg njih sme biti seveda prižgan še kakšen od bitov, ki so v z ugasnjeni). Pri tem nam pride prav operator &: v rezultatu z & k so prižgani tisti biti, ki so prižgani tako v z kot v k. Torej, če je imel k prižgane vse bite, ki so bili prižgani tudi v z, potem je z & k kar enak z; s tem lahko preverimo, če je imel nek kandidat vse zahtevane kvalifikacije.

Na enak način lahko preverimo tudi, če ima nek drugi kandidat vse kvalifikacije, ki jih je imel sprejeti kandidat; če je k' & k enako k, potem vemo, da ima k' vse tiste kvalifikacije, ki jih ima k; in če poleg tega k' in k nista enaka, potem vemo, da ima k' še kakšno kvalifikacijo, ki je k nima (tako lahko odkrivamo sumljive zaposlitve).

```
#include <stdio.h>

int main()
{
  char s[34]; unsigned int z, k, kk;

  gets(s); z = vStevilo(s); /* Preberimo zahtevane kvalifikacije. */
  gets(s); k = vStevilo(s); /* Preberimo kvalifikacije sprejetega kandidata. */

  /* Preverimo, ali ima kandidat vse zahtevane kvalifikacije. */
  if ((z & k) != z) { printf("nezakonito\n"); return 0; }

  /* Preglejmo ostale kandidate. */
  while (gets(s)) {
    kk = vStevilo(s);
    /* Ali ima kk vse kvalifikacije, ki jih ima k? */
    if ((k & kk) == k)
      /* Ali ima še kakšno kvalifikacijo, ki je k nima? */
      if (kk != k) { printf("sumljivo\n"); return 0; }}
  /* Če smo prišli do sem, je z imenovanjem vse v redu. */
  printf("zakonito\n"); return 0;
}
```

3. Dekodiranje nizov

Opazimo lahko, da se začetek podniza, ki ga obračamo, po vsakem koraku premakne za eno mesto v desno: pri k-tem obračanju smo imeli podniz, ki se je začel s k-tim znakom vhodnega niza. Naloga pravi, da pri dekodiranju kot vhodni podatek dobimo tudi seznam dolžin obrnjenega podniza pri vseh obračanjih; recimo, da so to dolžine $d_1, d_2, \ldots, d_m$. Iz tega torej vidimo, da je bilo obračanj m, torej se je moralo zadnje od teh obračanj — to je tisto, s katerim je kodirani niz dobil svojo končno podobo (iz katere ga mi zdaj skušamo dekodirati) — začeti z m-tim znakom niza. Vemo pa tudi, da je to obračanje zajelo podniz dolžine d_m, tako da zdaj točno vemo, za kateri podniz je šlo, in ga lahko obrnemo nazaj. Nadaljujemo lahko na enak način: predzadnje obračanje se je moralo začeti pri $(m-1)$-vem znaku in je zajelo podniz dolžine d_{m-1}, tako da lahko tudi tega obrnemo nazaj itd.

Zapišimo naš postopek še s psevdokodo:

vhod: niz $s = s_1 s_2 \ldots s_n$ in seznam dolžin $d_1, d_2, \ldots, d_m$;

for $k := m$ **downto** 1:
  (* *Obrni podniz* $s_k s_{k+1} \ldots s_{k+d_k-1}$. *)
  $i := k$; $j := k + d_k - 1$;
  while $i < j$:
    $t := s_i$; $s_i := s_j$; $s_j := t$; $i := i + 1$; $j := j - 1$;

4. Silhuete

Naloga pravi, da je x_c višina najvišje stolpnice v c-tem stolpcu; če poiščemo maksimum vrednosti x_c po vseh stolpcih (torej po vseh c od 1 do w), mora biti to potemtakem višina najvišje stolpnice sploh. Podobno mora biti tudi maksimum vrednosti y_r po vseh r (od 1 do h) višina najvišje stolpnice sploh. Če v naših vhodnih podatkih tadva maksimuma nista enaka, lahko takoj zaključimo, da taka postavitev ne obstaja.

Če pa sta maksimuma enaka, lahko razmišljamo takole. Stolpec, pri katerem x_c doseže svoj maksimum, označimo s C, podobno pa tudi vrstico, pri kateri y_r doseže svoj maksimum, označimo z R. Vemo torej, da velja $x_C = y_R$. Postavimo zdaj v vrstico R stolpnice z višinami $x_1, \ldots, x_w$ (torej $a_{Rc} = x_c$ za vsak c), v stolpec C pa stolpnice z višinami $y_1, \ldots, y_h$ (torej $a_{rC} = y_r$ za vsak r). Za stolpnico a_{RC} smo torej zahtevali, da mora biti visoka x_C in hkrati tudi y_R, kar pa na srečo ni problem, saj sta x_C in y_R enaka. Vse ostale stolpnice v mestu lahko zdaj postavimo na najmanjšo možno višino, torej 1.

Prepričajmo se, da tako dobljena postavitev ustreza našim omejitvam. V R-ti vrstici imamo zdaj stolpnice z višinami $x_1, \ldots, x_w$, najvišja je torej visoka x_C, kar pa je isto kot y_R; torej y_R res pove višino najvišje stolpnice v R-ti vrstici. V kateri koli drugi vrstici, recimo r-ti, pa imamo stolpnice z višino 1, razen tiste v C-tem stolpcu, ki je visoka y_r; torej nam y_r res pove višino najvišje stolpnic e vr-ti vrstici. Na analogen način bi se lahko prepričali tudi, da je x_c (za vsak c) res višina najvišje stolpnice v c-tem stolpcu.

```
#include <stdio.h>

#define MaxW 1000
#define MaxH 1000

int main()
{
  int w, h, x[MaxW], y[MaxH], r, c, a, R, C;

  /* Preberimo vhodne podatke. */
  scanf("%d %d", &w, &h);
  for (c = 0; c < w; c++) scanf("%d", &x[c]);
  for (r = 0; r < h; r++) scanf("%d", &y[r]);

  /* Poiščimo maksimalno višino. */
  for (C = 0, c = 1; c < w; c++) if (x[c] > x[C]) C = c;
  for (R = 0, r = 1; r < h; r++) if (y[r] > y[R]) R = r;
  if (x[C] != y[R]) { printf("taka postavitev ne obstaja\n"); return 0; }
```

```
  /* Izpišimo primerno postavitev. */
  for (r = 0; r < h; r++)
    for (c = 0; c < w; c++)
      printf("%d%c", (r == R ? x[c] : c == C ? y[r] : 1), (c == w - 1 ? '\n' : ' '));
  return 0;
}
```

Sestavljanja razporeda se lahko lotimo tudi še kako drugače (in dobimo drugačen, vendar tudi veljaven razpored); lahko bi na primer vzeli $a_{rc} = \min\{x_c, y_r\}$.

5. Ribič

Recimo, da mrežo počasi premikamo od leve proti desni (z drugimi besedami: počasi povečujemo x, torej koordinato levega konca mreže) in opazujemo, pri katerih x se število ujetih rib spremeni. Vidimo lahko, da so spremembe možne le v primerih, ko se (za nek i) vrednost x poveča z x_i na $x_i + 1$ (takrat i-ta riba pade iz mreže) ali pa se x poveča z $x_j - d$ na $x_j - d + 1$ (takrat pride j-ta riba v mrežo). Pri vseh drugih premikih mreže ostane število ujetih rib nespremenjeno.

Ker imamo seznam koordinat rib že urejen naraščajoče, se lahko zdaj po njem sprehajamo z i in j in gledamo, katera je naslednja koordinata začetka mreže (torej x), pri kateri pride do spremembe v številu ujetih rib. Če je bila prejšnja sprememba pri x, naslednja pa nastopi pri x', to pomeni, da je za vse položaje mreže od vključno x do vključno $x' - 1$ (tu je torej $x' - x$ možnih položajev) ulov enak; če je to ravno ulov, ki ga iščemo (torej natanko k rib), potem vemo, da teh $x' - x$ ugodnih položajev šteje k našemu končnemu rezultatu (spodnji postopek ga računa v r).

$i := 1$; $j := 1$; $x := x_1 - d$; $u := 0$; $r := 0$; **while** $i \le n$:
  (* *i je naslednja riba, ki bo padla iz mreže, j pa naslednja riba, ki bo prišla v mrežo. Kaj od tega dvojega se zgodi prej?* *)
  if $j \le n$ **then** $x_\oplus := x_j - d + 1$ **else** $x_\oplus := \infty$;
  $x_\ominus := x_i + 1$;
  $x' := \min\{x_\oplus, x_\ominus\}$; (* *Tu nastopi naslednja sprememba.* *)
  (* *Od x do $x' - 1$ je $x' - x$ možnih položajev mreže, pri katerih ujamemo natanko u rib; je to iskani ulov?* *)
  if $u = k$ **then** $r := r + x' - x$;
  (* *Izračunajmo novi ulov in se premaknimo po seznamu rib.* *)
  $x := x'$;
  if $x = x_\oplus$ **then** $u := u + 1$; $j := j + 1$;
  if $x = x_\ominus$ **then** $u := u - 1$; $i := i + 1$;
return r;

REŠITVE NALOG ZA TRETJO SKUPINO

1. Moderna umetnost

Recimo, da morata biti v končni verziji slike dva stolpca, x_1 in x_3, iste barve, nekje med njima pa je še nek stolpec neke druge barve; recimo, da je to stolpec x_2, pri čemer seveda velja $x_1 < x_2 < x_3$. Ali je mogoče, da bi x_1 in x_3 dobila svojo končno podobo v isti potezi z valjem? Pa recimo, da je to res. Poteza, ki pobarva x_1 in x_3, seveda pobarva tudi vse stolpce vmes, med drugim x_2; ker pa bo moral ta na koncu biti druge barve kot x_1 in x_3, to pomeni, da bomo morali x_2 pobarvati še z neko kasnejšo potezo. Ker se je dalo x_1 in x_3 pobarvati v eni potezi, sta gotovo manj kot k enot narazen (torej $x_3 - x_1 < k$). Tista poteza, ki kasneje pobarva x_2 z njegovo pravo barvo, pa seveda pobarva nek blok k stolpcev, ki med drugim vsebuje tudi x_2. Ker leži x_2 nekje med x_1 in x_3 in ker je med x_1 in x_3 največ $d-2$ stolpcev, je nemogoče, da bi tista poteza, ki bo pobarvala x_2, v celoti ležala med x_1 in x_3 — če nočemo, da nam ta poteza pokvari barvo stolpca x_1, bo gotovo pokrila stolpec x_3 in obratno. Tako torej vidimo, da je nemogoče, da bi x_1 in x_3 dobila svojo končno podobo v isti potezi.

Sliko si lahko torej predstavljamo kot sestavljeno iz blokov, pri čemer je vsak blok strnjeno zaporedje stolpcev iste barve. Primer iz besedila naloge ima na primer tri bloke: prvi blok je širok tri stolpce in ima barvo 2, drugi blok je širok dva stolpca in ima barvo 6, tretji blok pa je širok dva stolpca in ima spet barvo 2. Označimo širine blokov z $w_1, w_2, \ldots, w_m$. Širina i-tega bloka je w_i, tako da potrebujemo vsaj $\lceil w_i/k \rceil$ različnih potez, da dobijo vsi stolpci tega bloka svojo končno barvo. V prejšnjem odstavku smo tudi ugotovili, da nobena poteza ne more dati končne barve stolpcem iz dveh ali več različnih blokov; skupno število potez, ki jih potrebujemo, je torej $\sum_{i=1}^{m} \lceil w_i/k \rceil$.

Takšno barvanje lahko izvedemo čisto sistematično: gremo po sliki od leve proti desni, pri vsakem stolpcu preverimo, če ima že pravo barvo, in če je nima, izvedemo novo potezo z začetkom pri tem stolpcu in z barvo, ki jo mora na koncu ta stolpec imeti. Pri tem nam sploh ni treba vzdrževati tabele s trenutnim stanjem slike; za tisto, kar je levo od našega trenutnega položaja, že tako ali tako vemo, da je vse pravilno pobarvano in teh delov slike ne bomo več spreminjali; tisto, kar je desno od našega trenutnega položaja, pa je bilo bodisi pobarvano v zadnji doslej narejeni potezi (moramo si le zapomniti, katere barve je bila in do katerega stolpca je segla; v spodnjem programu imamo zato spremenljivki barvaDo in xPrazno), desno od tam pa je vse še nepobarvano.

```
#include <stdio.h>

#define MaxN 1000000

int main()
{
  int nPotez, barvaDo, xPrazno, x, n, k, b;
  FILE *f = fopen("umetnost.in", "rt");
  fscanf(f, "%d %d", &n, &k);

  xPrazno = 0; nPotez = 0;
  for (x = 0; x < n; x++)
  {
    fscanf(f, "%d", &b);
    if (x < xPrazno && b == barvaDo)
      continue; /* stolpec x je že prave barve */
    nPotez++; barvaDo = b; xPrazno = x + k;
  }
  fclose(f);

  f = fopen("umetnost.out", "wt"); fprintf(f, "%d\n", nPotez); fclose(f); return 0;
}
```

2. Kompleksnost števil

Označimo z $f(n)$ najmanjše število enic, ki jih potrebujemo, da sestavimo izraz z vrednostjo n. Zadnja operacija, ki jo pri tem izračunu izvedemo, je bodisi seštevanje bodisi množenje. Če je seštevanje, mora imeti eden od seštevancev vrednost 1 (tako zahteva besedilo naloge), drugi pa torej vrednost $n-1$, da bo vsota potem n. Naš izraz bo torej oblike $1+E$, pri čemer mora imeti podizraz E vrednost $n-1$. Po definiciji funkcije f vemo, da za izraz z vrednostjo $n-1$ potrebujemo najmanj $f(n-1)$ enic, torej bo imel naš izraz oblike $1+E$ (z vrednostjo n) vsega skupaj $f(n-1)+1$ enic.

Druga možnost je, da je zadnja operacija v našem izrazu množenje; naš izraz je torej oblike $E \cdot E'$. Naj bo k vrednost podizraza E; k mora biti torej nek delitelj števila n, drugi podizraz E' pa mora imeti vrednost n/k. Vemo, da za izraz z vrednostjo k potrebujemo najmanj $f(k)$ enic, za izraz z vrednostjo n/k pa najmanj $f(n/k)$ enic. Naš izraz oblike $E \cdot E'$ (z vrednostjo n) bo imel torej $f(k)+f(n/k)$ enic. Število n ima lahko seveda precej deliteljev k, zato moramo med njimi izbrati tistega, pri katerem bo skupno število enic najmanjše. Lahko pa se omejimo na primere, ko je $k \le \sqrt{n}$, kajti če je $k > \sqrt{n}$, potem je $n/k \le \sqrt{n}$ in bi dobili isti izraz kot pri nekem manjšem k, le vrstni red faktorjev bi bil zamenjan (na primer $6 \cdot 2$ namesto $2 \cdot 6$).

Tako torej vidimo, da lahko funkcijo f računamo takole:

$$f(n) = \min\{1 + f(n-1), \min\{f(d) + f(n/d) : 2 \le d \le \sqrt{n},\ d \textit{ deli } n\}\}.$$

To formulo lahko precej neposredno predelamo v rekurzivni podprogram (funkcijo), ki računa f in pri tem kliče samega sebe, da pride do vrednosti $f(n-1)$, $f(d)$ in podobno.

```
int KolikoEnic(int n)
{
  int f, d, c;
  if (n == 1) return 1;
  f = KolikoEnic(n - 1) + 1; /* število enic v izrazu 1 + (n - 1) */
  for (d = 2; d * d <= n; d++) {
    if (n % d != 0) continue; /* mogoče d sploh ni delitelj n-ja */
    c = KolikoEnic(d) + KolikoEnic(n / d); /* število enic v izrazu d * (n / d) */
    if (c < f) f = c; }
  return f;
}
```

Slabost take rešitve je, da bi se funkcija pri takšnem izračunu večkrat klicala z enako vrednostjo argumenta n, tako da bi po nepotrebnem izgubljala čas z večkratnim izračunom ene in iste vrednosti funkcije f. Časovna zahtevnost takšne funkcije bi bila eksponentna v odvisnosti od n in v sprejemljivo hitrem času bi lahko obdelali n-je velikosti nekaj 100, za $n = 1000$ pa bi bila že prepočasna. Pri testnih primerih z našega tekmovanja bi ta rešitev dobila 40 % točk.

Do učinkovite rešitve pridemo, če si vsako vrednost $f(n)$, čim jo izračunamo, zapomnimo v neki tabeli; če jo nekoč kasneje spet potrebujemo, jo poberemo iz tabele, namesto da bi jo računali še enkrat. Zdaj lahko vrednosti funkcije $f(n)$ računamo čisto sistematično od manjših n proti večjim:

```
void KolikoEnic2(int n, int *f)
{
  int d, c, m;
  f[0] = 0; f[1] = 1;
  for (m = 2; m <= n; m++) {
    f[m] = f[m - 1] + 1;
    for (d = 2; d * d <= m; d++) {
      if (m % d != 0) continue;
      c = f[d] + f[m / d];
      if (c < f[m]) f[m] = c; }}
}
```

Zdaj imamo pri vsakem m-ju $O(\sqrt{m})$ dela (zaradi notranje zanke po d), zunanja zanka po m pa gre do n, tako da je časovna zahtevnost celotnega postopka $O(n\sqrt{n})$. Besedilo naloge pravi, da v enem testnem primeru pravzaprav dobimo več n-jev, vendar so urejeni padajoče; torej, ko bomo z gornjim podprogramom izračunali $f(n)$ za prvega od teh n-jev, bodo v tabeli f že pripravljene tudi vrednosti funkcije f za vse možne manjše n-je, tako da moramo za vse nadaljnje n-je iz vhodne datoteke le pogledati v tabelo in izpisati ustrezen odgovor. Pri testnih primerih z našega tekmovanja bi takšna rešitev dovolj hitro rešila 80 % točk.

Videli smo, da imamo pri posameznem m-ju $O(\sqrt{m})$ dela z iskanjem deliteljev m-ja; marsikateri m ima precej manj kot toliko deliteljev, tako da naša rešitev porabi precej časa s preizkušanjem neobetavnih vrednosti d. Veliko lažje je, če problem obrnemo: namesto da bi se pri danem m spraševali, kateri d so njegovi delitelji, se lahko pri danem d vprašamo, kateri m so njegovi večkratniki; in tu je odgovor zelo enostaven: to so $2d, 3d, 4d$ in tako naprej. Tako pridemo do naslednje rešitve:

```
void KolikoEnic3(int n, int *f)
{
  int m, k, mk;

  /* Na začetku postavimo vse f[m] na neko veliko vrednost, ki jo bomo kasneje zmanjševali,
     ko bomo odkrivali boljše načine za izražavo števila m z enicami. */
  for (m = 0; m <= n; m++)
    f[m] = m; /* to je gotovo večje ali enako od prave vrednosti f(m) */

  for (m = 1; m <= n; m++) {
```

```
    /* Trenutno je v f[m] najmanjše število enic, ki jih potrebujemo,
       če hočemo m izraziti kot zmnožek d * (m/d).
       Poglejmo še, s koliko enicami ga lahko izrazimo kot vsoto 1 + (m - 1). */
    if (1 + f[m - 1] < f[m]) f[m] = 1 + f[m - 1];
    /* Zdaj imamo v f[m] pravo vrednost funkcije f(m). Izraz za število m lahko zdaj
       uporabimo kot enega od faktorjev v izrazih oblike m * k; poglejmo, če lahko na ta
       način izboljšamo kakšno od dosedanjih vrednosti f[m * k]. */
    for (k = 2, mk = m + m; k <= m && mk <= n; k++, mk += m)
      if (f[m] + f[k] < f[mk]) f[mk] = f[m] + f[k]; }
}
```

Pri notranji zanki smo šli s k le do m, saj za večje k še ne poznamo prave vrednosti $f(k)$; produkte, pri katerih je $k > m$, bomo obravnavali takrat, ko bomo z zunanjo zanko prišli do k.

Kakšna je časovna zahtevnost tega postopka? Notranja zanka (po k) izvede največ n/m iteracij, vsaka od teh iteracij pa nam vzame $O(1)$ časa. Zunanja zanka gre z m od 1 do n; skupno število iteracij notranje zanke je torej kvečjemu $n/1+n/2+n/3+\ldots+n/n = n\sum_{m=1}^{n}(1/m)$. Vsota $\sum_{m=1}^{n}(1/m)$ je v matematiki znana kot n-to harmonično število H_n in je približno enaka $\ln n$. Časovna zahtevnost našega postopka je torej $O(n \log n)$. S tem bomo lahko dovolj hitro rešili tudi največje testne primere.

Zapišimo še preostanek programa:

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

void KolikoEnic3(int n, int *f) { ... } /* glej zgoraj */

int main()
{
  FILE *g = fopen("kompleksnost.in", "rt"), *h = fopen("kompleksnost.out", "wt");
  int n, t, *f;
  fscanf(g, "%d", &t); /* število n-jev v tej vhodni datoteki */
  fscanf(g, "%d", &n); /* največji n v tej vhodni datoteki */

  /* Alocirajmo pomnilnik in izračunajmo f(1), ..., f(n). */
  f = (int *) malloc(sizeof(int) * (n + 1));
  KolikoEnic2(n, f);

  /* Izpišimo vse rezultate, po katerih sprašuje vhodna datoteka. */
  fprintf(h, "%d\n", f[n]);
  while (--t > 0) { fscanf(g, "%d", &n); fprintf(h, "%d\n", f[n]); }

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

3. Požar

Oglejmo si najprej preprosto, vendar neučinkovito rešitev, ki bi bila dobra za majhne testne primere. Če se požar začne v (x_0, y_0), bo celica (x, y) zagorela $|x - x_0| + |y - y_0|$ sekund po začetku požara. V največ $w + h - 2$ sekundah bodo torej zagorele vse celice (v tem času bi se na primer požar razširil iz enega vogala mreže v diagonalno nasprotni vogal). Lahko gremo torej po vseh celicah mreže in za vsako izračunamo, kdaj bo zagorela; v tabeli g si za vsak časovni trenutek vzdržujemo podatek o tem, koliko celic zagori v tistem trenutku. Nato se moramo le še sprehoditi po tej tabeli in seštevati števila v njej, dokler vsota ne preseže k; takrat vemo, da smo našli prvi časovni trenutek, v katerem gori t celic.

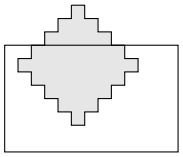
```
typedef long long int64;

int Pozar(int w, int h, int x0, int y0, int64 k)
{
  int x, y, t; int64 *g = (int64 *) malloc((w + h - 1) * sizeof(int64)), vsota;
  for (t = 0; t <= w + h - 2; t++) g[t] = 0;
```

```
  /* Preglejmo mrežo, za vsako celico izračunajmo, kdaj zagori,
     in povečajmo ustrezni element tabele g. */
  for (y = 1; y <= h; y++) for (x = 1; x <= w; x++)
    g[abs(x - x0) + abs(y - y0)]++;
  /* Seštejmo prvih nekaj elementov tabele g, dokler vsota ne doseže k. */
  for (t = 0, vsota = g[t]; vsota < k; ) vsota += g[++t];
  free(g); return t;
}
```

Na našem tekmovanju bi dobila ta rešitev 40 % točk; pri velikih mrežah je prepočasna, pa tudi preveč pomnilnika bi porabila, saj za tabelo g potrebujemo $O(w+h)$ prostora.

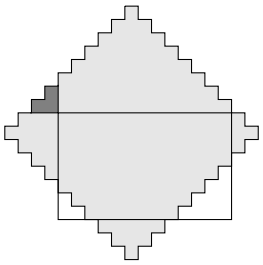
Razmislimo malo bolj geometrijsko. Če si izberemo nek čas t, koliko celic gori t sekund po začetku požara? Načeloma so to vse celice (x, y), za katere velja $|x - x_0| + |y - y_0| \leq t$. Na mreži tvorijo obliko kara $\diamond$ z oglišči $(x_0 \pm t, y)$ in $(x_0, y \pm t)$. Števila celic v tem karu ni težko izračunati: če jih seštevamo kar po vrsticah, vidimo, da jih je $1+3+5+\ldots+(2t-1)+(2t+1)+(2t-1)+\ldots+5+3+1$; to je skupaj $(t+1)^2+t^2$.

Ta karo se vsako sekundo malo poveča; stvari se zapletejo, ko trči ob robove mreže. Takrat nam formula iz prejšnjega odstavka daje prevelike rezultate, saj šteje tudi tiste dele kara, ki segajo čez rob mreže. Oglejmo si na primer spodnjo sliko, pri kateri del kara štrli nad zgornji rob mreže:



Vemo, da je zgornji vogal kara $(x_0, y_0 - t)$; najmanjša veljavna y-koordinata pa je 1. Torej nad zgornji rob mreže sega $d := 1 - (y_0 - t)$ vrstic našega kara. (Če je $d \leq 0$, to pomeni, da naš karo ne štrli čez zgornji rob.) Te vrstice tvorijo trikotnik s ploščino $1+3+5+\ldots+(2d-1) = d^2$; to moramo torej odšteti od ploščine našega kara. Podoben razmislek moramo opraviti še za ostale tri stranice mreže.

Še dodaten zaplet pa nastopi, ko se karo poveča do te mere, da nek del kara leži tako nad zgornjim robom kot levo od levega roba, na primer takole:



Temno osenčeno območje pripada tako trikotniku, ki štrli nad zgornji rob, kot trikotniku, ki štrli levo od levega roba. Ker smo doslej od ploščine kara odšteli ploščino obeh trikotnikov, to pomeni, da smo ploščino temnega območja neupravičeno odšteli dvakrat namesto samo enkrat. Zdaj jo moramo torej enkrat prišteti nazaj. Kolikšno je to območje? Celica v zgornjem levem kotu mreže je $(1, 1)$; prva celica našega temnega območja (zunaj meja mreže), ki jo požar doseže, je torej tista diagonalno stran od nje, to je $(0, 0)$. Ta celica zagori $|0 - x_0| + |0 - y_0|$ sekund po začetku požara; po tistem se požar širi še $d := t - (|0 - x_0| + |0 - y_0|)$ sekund. (Če je $d < 0$, pomeni, da požar temnega območja sploh ne doseže.) Temno območje je zdaj trikotnik s ploščino $1+2+3+\ldots+(d+1) = (d+1)(d+2)/2$. To moramo prišteti dosedanji ploščini našega gorečega območja. Podoben razmislek moramo opraviti še za ostale tri vogale, razlika je le v tem, da namesto koordinat $(0, 0)$ vzamemo $(w+1, 0)$, $(0, h+1)$ in $(w+1, h+1)$.

```
int64 KolikoGori(int64 w, int64 h, int64 x0, int64 y0, int64 t)
{
```

```
  int64 a, d;
  a = (t + 1) * (t + 1) + t * t; /* karo */
  /* Odštejmo tisto, kar štrli čez robove. */
  d = 1 − (y0 − t); if (d > 0) a −= d * d;
  d = 1 − (x0 − t); if (d > 0) a −= d * d;
  d = (y0 + t) − h; if (d > 0) a −= d * d;
  d = (x0 + t) − w; if (d > 0) a −= d * d;
  /* Prištejmo tisto, kar smo neupravičeno odšteli dvakrat. */
  d = t − abs(0 − x0) − abs(0 − y0); if (d >= 0) a += (d + 1) * (d + 2) / 2;
  d = t − abs(w + 1 − x0) − abs(0 − y0); if (d >= 0) a += (d + 1) * (d + 2) / 2;
  d = t − abs(0 − x0) − abs(h + 1 − y0); if (d >= 0) a += (d + 1) * (d + 2) / 2;
  d = t − abs(w + 1 − x0) − abs(h + 1 − y0); if (d >= 0) a += (d + 1) * (d + 2) / 2;
  return a;
}
```

Vidimo torej, da smo gorečo površino pri danem času t izračunali v $O(1)$ časa. Zdaj bi načeloma lahko t postopoma povečevali, dokler goreča površina ne doseže k:

```
int Pozar2(int w, int h, int x0, int y0, int64 k)
{
  int t = 0;
  while (KolikoGori(w, h, x0, y0, t) < k) t++;
  return t;
}
```

To je že precej bolje od prejšnje rešitve; še vseeno pa se lahko zgodi, da je potrebni čas gorenja tja do $w + h - 2$ (npr. če se požar začne v enem od vogalov mreže in hočemo, da zagori celotna mreža). Časovna zahtevnost te rešitve je torej $O(w + h)$, kar je za največje testne primere še vedno prepočasi; bi pa dovolj hitro rešili tistih 60 % primerov, za katere besedilo naloge zagotavlja, da bo čas gorenja $\leq 10^6$ sekund.

Bolje pa je, če čas gorenja določamo z bisekcijo: 0 sekund je premalo (razen če je $k = 1$), $w+h$ sekund je gotovo dovolj, pravilna rešitev je nekje vmes; zdaj pa ta interval na vsakem koraku razpolovimo, dokler se ne zoži na eno samo sekundo; tisto je potem čas gorenja, ki ga iščemo.

```
int Pozar3(int w, int h, int x0, int y0, int64 k)
{
  int64 tL = −1, tD = w + h, t, g;
  while (tD − tL > 1) {
    /* Invarianta: tL sekund je premalo, tD sekund je dovolj. */
    t = (tL + tD) / 2;
    g = KolikoGori(w, h, x0, y0, t);
    if (g < k) tL = t; else tD = t; }
  /* Zdaj je tL = tD − 1; torej je tD − 1 sekund še premalo, tD pa že dovolj. */
  return (int) tD;
}
```

Zapišimo še preostanek rešitve:

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

int Pozar3(int w, int h, int x0, int y0, int64 k) { ... } /* glej zgoraj */

int main()
{
  int p, w, h, x0, y0; int64 k;
  FILE *f = fopen("pozar.in", "rt"), *g = fopen("pozar.out", "wt");
  fscanf(f, "%d", &p); /* preberimo število požarov */
  while (p−− > 0) { /* obdelajmo vse požare */
    fscanf(f, "%d %d %d %d %lld", &w, &h, &x0, &y0, &k);
    fprintf(g, "%d\n", Pozar3(w, h, x0, y0, k)); }
  fclose(f); fclose(g);
}
```

4. Številčenje

Naloga sprašuje po pogostosti števk v zapisu števil od a do b, vendar lahko to predelamo v malo preprostejši problem: kolikokrat se neka števka pojavi, če zapišemo vsa števila, manjša od n (torej od 1 do $n-1$)? Če znamo izračunati to za poljuben n, bomo morali le vzeti $n = b+1$ in $n = a$ ter izračunati razliko tako dobljenih pogostosti — to bodo potem ravno pogostosti v zapisu števil od a do b.

Zapišimo zdaj naš n v desetiškem zapisu: $n = n_{k-1}n_{k-2}\ldots n_2n_1n_0$, pri čemer je vodilna števka n_{k-1} seveda večja od 0 (sicer je sploh ne bi pisali). Zanima nas pogostost števk v številih, manjših od n. Ta števila lahko razdelimo na tista, ki so enako dolga kot n (torej imajo k) števk, in tista, ki so krajša (torej imajo t števk za nek t od 1 do $k-1$).

Vseh t-mestnih števil je seveda $9 \cdot 10^{t-1}$ (za prvo števko je 9 možnosti, ker mora biti različna od 0, za vsako od ostalih $t-1$ števk pa je 10 možnosti). Na spodnjih $t-1$ mestih je torej skupaj $9 \cdot 10^{t-1} \cdot (t-1)$ števk in ker se vse števke od 0 do 9 pojavljajo na teh mestih enako pogosto, odpade na vsako od njih ena desetina teh pojavitev, torej $9 \cdot 10^{t-2} \cdot (t-1)$. Poleg tega se vsaka števka razen 0 pojavlja še enkrat kot vodilna števka v 10^{t-1} številih. Opisani razmislek ponovimo za vse t od 1 do k in tako preštejemo pojavitve vseh števk v vseh številih, ki so krajša od n.

Razmislimo zdaj še o k-mestnih številih; tako število (recimo x) je enako dolgo kot n in se mogoče celo ujema z njim v nekaj vodilnih števkah, prej ali slej pa mora nastopiti neujemanje, kjer ima naš x manjšo števko od istoležne števke n-ja. Naj bo zdaj t indeks najvišje števke, kjer nastopi neujemanje med x in n. Koliko je pri tem t-ju možnih x-ov? Števka x_t mora biti manjša od n_t, zato pridejo zanjo v poštev vrednosti $0, 1, \ldots, n_t - 1$; rahlo poseben primer je $t = k$, ko je x_t vodilna števka, zato ne sme imeti vrednosti 0. Naj bo torej n'_t število možnih vrednosti števke x_t (torej $n'_t = n_t$ za $t < k$ in $n'_k = n_k - 1$). Števke levo od x_t morajo biti enake kot pri n, zato nimamo glede njih nobene izbire; števke desno od x_t pa so lahko poljubne, torej imamo zanje 10^t možnosti.

Tako torej vidimo, da je pri danem t možnih $n'_t \cdot 10^t$ primernih x-ov. Na spodnjih t mestih (desno od x_t) je vsega skupaj $n'_t \cdot 10^t \cdot t$ števk in ker se vse vrednosti pojavljajo tu enako pogosto, odpade na vsako od njih $n'_t \cdot 10^{t-1} \cdot t$ pojavitev. Na mestu t dobi vsaka števka od 0 do $n_t - 1$ (ali pa od 1 do $n_t - 1$, če je $t = k$) po 10^t pojavitev. Na mestih levo od t pa moramo le pogledati, kolikokrat se vsaka števka pojavi v tem delu n-ja, in to pomnožiti z $n'_t \cdot 10^t$, saj so pri vseh x-ih (pri tem t) te števke enake kot pri n.

```
#include <stdio.h>

#define MaxStevk 17
typedef long long int64;

void Kolikokrat(int64 n, int64 *p)
{
  int ns[MaxStevk], pn[10], k = 0, t, d, nt; int64 pow10[MaxStevk];

  /* Inicializirajmo tabelo p. */
  for (d = 0; d <= 9; d++) p[d] = 0;
  if (n <= 0) return;

  /* Razbijmo n na števke. */
  while (n > 0) { ns[k++] = n % 10; n /= 10; }

  /* Izračunajmo si tabelo potenc števila 10. */
  for (pow10[0] = 1, t = 1; t < MaxStevk; t++) pow10[t] = 10 * pow10[t - 1];

  /* Preštejmo pojavitve števk v številih, krajših od n. */
  for (t = 1; t < k; t++) { /* Gledamo t-mestna števila. */

    /* Vodilne števke. */
    for (d = 1; d <= 9; d++) p[d] += pow10[t - 1];

    /* Števke na nižjih mestih. */
    if (t > 1) for (d = 0; d <= 9; d++) p[d] += 9 * pow10[t - 2] * (t - 1); }

  /* Preštejmo pojavitve števk v številih x, enako dolgih kot n. */
  for (d = 0; d <= 9; d++) pn[d] = 0;
  for (t = k - 1; t >= 0; t--) { /* t je najvišje mesto neujemanja med n in x. */
```

```
    nt = ns[t] − (t == k − 1 ? 1 : 0); /* možne vrednosti števke x_t */
    /* Vsi x se v mestih levo od t ujemajo z n in tam se števka d pojavi pn[d]-krat. */
    for (d = 0; d <= 9; d++) p[d] += pn[d] * nt * pow10[t];
    /* Prištejmo pojavitve na mestu x_t. */
    for (d = (t == k − 1 ? 1 : 0); d < ns[t]; d++) p[d] += pow10[t];
    /* Prištejmo še pojavitve na nižjih mestih. */
    if (t > 0) for (d = 0; d <= 9; d++) p[d] += nt * pow10[t − 1] * t;
    pn[ns[t]]++; }
}

int main()
{
  int64 a, b, pa[10], pb[10]; int d;
  FILE *f = fopen("stevilcenje.in", "rt");
  fscanf(f, "%lld %lld", &a, &b);
  fclose(f);
  Kolikokrat(a, pa); Kolikokrat(b + 1, pb);
  f = fopen("stevilcenje.out", "wt");
  for (d = 0; d <= 9; d++) fprintf(f, "%lld\n", pb[d] − pa[d]);
  fclose(f); return 0;
}
```

5. Natrpan urnik

Te naloge se lahko lotimo na veliko različnih načinov, od preprostejših in počasnejših do učinkovitejših, vendar bolj zapletenih. Za vsak predmet moramo vzdrževati množico dni, v katerih nastopijo testi tistega predmeta; vsaka od teh množic je podmnožica množice $\{1, \ldots, D\}$, vsebuje pa največ N elementov; operaciji, ki ju moramo na tej množici podpirati, pa sta dodajanje novega elementa in poizvedba, pri kateri moramo prešteti, koliko elementov množice leži na intervalu $[d_1, d_2]$. Za število predmetov bomo uporabljali simbol P; pri naši nalogi je $P = 5$. Vse spodaj opisane rešitve dajejo pravilne rezultate in bi na našem tekmovanju dovolj hitro rešile vsaj 40 % testnih primerov.

(1) Zelo preprosta rešitev je, da množico predstavimo preprosto kot (neurejen) seznam, lahko v tabeli ali pa kot verigo, povezano s kazalci (*linked list*). Dodajanje je v tem primeru zelo preprosto in nam vzame le $O(1)$ časa, pri poizvedbi pa se moramo sprehoditi po celotnem seznamu in za vsak datum v njem preverjati, ali pripada intervalu $[d_1, d_2]$ ali ne; cena poizvedbe je zato $O(N)$. Tudi poraba pomnilnika je $O(N)$. Takšna rešitev je primerna za manjše testne primere, sploh če je poizvedb malo (dodajanj pa je lahko veliko).

(2) Še ena preprosta rešitev je, da imamo tabelo D logičnih vrednosti, ki nam za vsak dan povedo, ali je tisti dan kakšen test iz tega predmeta ali ne. Na začetku postavimo vse elemente tabele na **false**, pri vsakem dodajanju pa postavimo ustrezni element tabele na true. Dodajanje torej vzame le $O(1)$ časa. Za poizvedbo pa moramo pregledati del tabele od indeksa d_1 do indeksa d_2 in šteti, koliko elementov ima vrednost **true**; zato traja poizvedba $O(D)$ časa. Tudi poraba pomnilnika je zdaj $O(PD)$ namesto $O(N)$. Ta rešitev je torej slabša od prejšnje, če je testov veliko manj kot dni (testov gotovo ne more biti več kot dni, saj naloga pravi, da je na en dan lahko le en test posameznega predmeta). V praksi pa v naših testnih primerih število testov ni bilo tako majhno in rešitev s tabelo D logičnih vrednosti se ne obnese nič slabše od prejšnje.

(3) Malo boljša različica te rešitve je, da namesto tabele elementov tipa **bool** oz. boolean naredimo tabelo D bitov (oz. tabelo $\lceil D/8 \rceil$ bytov). Štetje testov je zdaj še hitrejše, malo zato, ker porabi tabela manj pomnilnika in zato bolje izkoristi procesorjev predpomnilnik, malo pa zato, ker si lahko za vsako možno vrednost enega byta vnaprej potabeliramo število prižganih bitov, zato med odgovarjanjem na poizvedbo lahko pregledamo po osem dni v enem koraku. Asimptotično sicer nismo ničesar pridobili, smo pa postopek pospešili za nek konstantni faktor, ki nam lahko omogoči rešiti kakšen testni primer več kot prej. Na našem tekmovanju bi dobra implementacija te rešitve lahko dosegla do 60 % točk.

(4) Namesto neurejenega seznama testov (za vsak predmet) imamo lahko seznam testov, urejen naraščajoče po datumu. Pri poizvedbi zdaj z bisekcijo poiščemo prvi test od dne d_1 naprej in prvi test po dnevu d_2; razlika med njima je zdaj ravno število testov v poizvedovalnem obdobju. Časovna zahtevnost poizvedbe je tako le $O(\log N)$. Po drugi strani pa je dodajanje zdaj drago: ko vrinemo nov test v urejen seznam, moramo kasnejše teste pomakniti za eno mesto naprej po tabeli, tako da dodajanje traja $O(N)$ časa. V primeru, ko pride med dvema poizvedbama več dodajanj, si lahko privoščimo majhno izboljšavo: elemente začasno dodajajmo v nek pomožni neurejen seznam; tik pred naslednjo poizvedbo pa ta seznam uredimo in z zlivanjem dodajmo njegove elemente v glavni urejeni seznam. Cena bloka b dodajanj je tako le $O(b \log b + N)$ namesto $O(bN)$. Tudi ta rešitev bi na našem tekmovanju lahko dosegla do 60 % točk.

(5) Leto lahko razdelimo na približno $\sqrt{D}$ „mesecev“ s po $\sqrt{D}$ dnevi. Za vsak mesec (in vsak predmet) imejmo neurejen seznam testov tega predmeta v tem mesecu, poleg tega pa v neki tabeli hranimo tudi dolžino tega seznama. Dodajanje nam še vedno vzame le $O(1)$ časa, saj moramo dodati element v enega od seznamov in povečati podatek o njegovi dolžini v tabeli. Pri poizvedbi pa je zdaj tako: za tiste mesece, ki v celoti ležijo znotraj poizvedovalnega območja, moramo le sešteti število testov v njih (za vsak mesec preberemo število testov iz tabele); s tem je le $O(\sqrt{D})$ dela, saj je mesecev le $\sqrt{D}$. Ostaneta nam še mesec na začetku in na koncu poizvedovalnega območja; tadva mogoče ne ležita v celoti v našem poizvedovalnem območju, zato moramo iti po seznamu testov za vsakega od teh dveh mesecev in za vsak test preverjati, ali leži na našem poizvedovalnem območju ali ne. Ker ima vsak mesec le $\sqrt{D}$ dni in ker na vsak dan pride največ en test, sta tadva seznama dolga le $O(\sqrt{D})$. Tako torej vidimo, da je cena ene poizvedbe $O(\sqrt{D})$, poraba pomnilnika pa $O(N + p\sqrt{D})$. Na naših testnih primerih bi ta rešitev dobila 80 % točk.

(6) Leto lahko predstavimo s tabelo D elementov, ki za vsak dan povedo, ali je takrat test ali ne; nato razdelimo leto na $D/2$ dvodnevnih obdobij in imejmo tabelo $D/2$ elementov, ki za vsako tako dvodnevno obdobje povedo, koliko testov je v tem dvodnevnem obdobju; podobno naredimo še eno tabelo z $D/4$ elementi, eno z $D/8$ elementi in tako naprej. Vse te tabele skupaj imajo približno $2D$ elementov, tako da je poraba pomnilnika zdaj $O(PD)$, vendar z veliko večjim konstantnim faktorjem kot pri rešitvah (2) in (3). Pri dodajanju novega testa moramo povečati za 1 ustrezni element vsake od teh tabel; dodajanje zato traja $O(\log D)$ časa, ker je tudi tabel toliko.

Pri poizvedbi pa lahko razmišljamo takole: če nas zanima, koliko je testov v prvih d dnevih leta, lahko zapišemo d v dvojiškem zapisu, torej kot vsoto potenc števila 2: na primer $81 = 64 + 16 + 1$; to pomeni, da lahko vzamemo prvo 64-dnevno obdobje (iz tabele, ki se nanaša na 64-dnevna obdobja), ki pokriva dneve 1..64, nato vzamemo eno od 16-dnevnih obdobij (tisto, ki pokriva dneve 65..80; iz tabele, ki se nanaša na 16-dnevna obdobja) in končno eno enodnevno obdobje (namreč dan 81). Tako torej vidimo, da moramo iz vsake od naših $\log_2 D$ tabel uporabiti kvečjemu en element, da dobimo število dni v pripadajočem obdobju; nato jih moramo le še sešteti, pa dobimo število testov v prvih d dnevih. Če nas zanima število testov od dneva d_1 do d_2, lahko najprej izračunamo število testov v prvih d_2 dnevih in nato od njega odštejemo število testov v prvih $d_1 - 1$ dnevih.

Ta in vse nadaljnje rešitve že rešijo vse testne primere z našega tekmovanja.

(7) Zelo elegantna rešitev je tudi Fenwickovo drevo; tu imamo za vsak predmet tabelo z D elementi, pri čemer element na indeksu k vsebuje število testov v obdobju od vključno dneva $f(k) + 1$ do vključno dneva k. Pri tem je $f(k)$ število, ki ga dobimo, če v k ugasnemo najnižji prižgani bit. Pokazati je mogoče, da lahko s takšno tabelo tako dodajanja kot poizvedbe izvajamo v času $O(\log n)$. Prednost v primerjavi z rešitvijo (6) je v tem, da porabimo pol manj pomnilnika.

(8) Ena od slabosti prejšnjih dveh rešitev je, da je poraba pomnilnika sorazmerna s številom dni v letu namesto s številom testov. Ker je lahko v letu največ N testov, ima vsaka od tabel iz rešitve (6) lahko največ N neničelnih elementov. Lahko bi jo torej predelali v razpršeno tabelo in hranili samo neničelne elemente; s tem se poraba pomnilnika zmanjša na $O(PN \log D)$, dodajanje in poizvedba pa sta še vedno možni v času $O(\log D)$, čeprav z večjim konstantnim faktorjem kot prej. Še en pogled na to rešitev je, da si družino tabel v (6) predstavljamo kot polno binarno drevo z $\log_2 D$

nivoji in D listi; če ga predstavimo eksplicitno kot drevo (namesto s tabelami), si ni težko predstavljati, da lahko iz njega porežemo prazna poddrevesa (taka, ki ne vsebujejo nobenega testa).

(9) Za predstavitev množice datumov lahko uporabimo kakšnega od binarnih iskalnih dreves, pazimo le na to, da se nam ne bo izrodilo (uporabimo na primer rdeče-črno drevo ali pa AVL-drevo). V vsako vozlišče dodajmo še podatek o tem, koliko je vozlišč v celotnem poddrevesu z začetkom pri tem vozlišču. Tako lahko v času $O(\log N)$ ne le dodajamo in iščemo elemente, ampak tudi ugotovimo, koliko elementov leži na danem poizvedovalnem intervalu. Prednost v primerjavi s (6), (7) in (8) je, da porabimo zdaj le $O(N)$ pomnilnika, ne več $O(PD)$ ali $O(PN \log D)$.

Oglejmo še si konkreten primer implementacije rešitve (6).

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

#define P 5
#define MaxNivojev 21 /* ≤ log2 D */
/* V tabeli, na katero kaže a[p], so po vrsti zložene vse tabele za predmet p;
   zac[k] pove, na katerem indeksu se začne k-ta od teh tabel
   (tista, v kateri vsak element predstavlja neko obdobje 2^k dni).
   Skupno število teh tabel pa je „visina“. */
int D, *a[P], zac[MaxNivojev], visina;

void PripraviTabele()
{
  int p, i, dolzNivoja = D, skupajDolz = D;
  visina = 1; zac[0] = 0;
  while (dolzNivoja > 1) {
    zac[visina++] = skupajDolz;
    dolzNivoja = (dolzNivoja + 1) / 2;
    skupajDolz += dolzNivoja; }
  for (p = 0; p < P; p++) {
    a[p] = (int *) malloc(skupajDolz * sizeof(int));
    for (i = 0; i < skupajDolz; i++) a[p][i] = 0; }
}

void Dodaj(int *tabela, int d) {
  int nivo;
  assert(tabela[zac[0] + d] == 0);
  for (nivo = 0; nivo < visina; nivo++)
    tabela[zac[nivo] + (d >> nivo)]++; }

int KolikoPred(int *tabela, int d) {
  int nivo = 0, rezultat = 0;
  while (d > 0) {
    if (d & 1) rezultat += tabela[zac[nivo] + d - 1];
    d >>= 1; nivo++; }
  return rezultat; }

int main()
{
  int nSkupin, n, p, d1, d2, s1, s2; char ukaz[2];
  long long rezultat = 0;

  FILE *f = fopen("urnik.in", "rt");
  fscanf(f, "%d %d", &nSkupin, &D);
  PripraviTabele();

  while (nSkupin-- > 0)
  {
    fscanf(f, "%s", ukaz);
    if (ukaz[0] == 'V') {
      fscanf(f, "%d %d %d %d", &n, &p, &d1, &s1);
      while (n-- > 0) { Dodaj(a[p - 1], d1 - 1); d1 += s1; }
      }
```

```
    else if (ukaz[0] == 'P') {
      fscanf(f, "%d %d %d %d %d %d", &n, &p, &d1, &d2, &s1, &s2);
      while (n-- > 0) {
        rezultat += KolikoPred(a[p - 1], d2) - KolikoPred(a[p - 1], d1 - 1);
        d1 += s1; d2 += s2; }}
  }

  fclose(f);
  f = fopen("urnik.out", "wt"); fprintf(f, "%lld\n", rezultat); fclose(f);
  for (p = 0; p < P; p++) free(a[p]);
  return 0;
}
```

Viri nalog: kino — Nino Bašić, dvigalo, silhuete — Primož Gabrijelčič, ribič, moderna umetnost, požar — Tomaž Hočevar, binarni sef — Jurij Kodre, dekodiranje nizov — Mitja Lasič, kolera — Mark Martinec, sumljiva imenovanja, natrpan urnik — Jure Slak, igličnі tiskalnik — Boštjan Slivnik, vandali, številčenje — Mitja Trampuš, kompleksnost števil — Janez Brank. Hvala Tomažu Hočevarju za pomoč pri implementaciji rešitev nalog za 3. skupino. Primer „soba z razgledom na vrt“ pri nalogi „dekodiranje nizov“ je iz 3. poglavja *Desetega brata*.

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