12. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2017

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.");
   }
}
```

12. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2017

NALOGE ZA PRVO SKUPINO

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

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

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

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

1. Zaokrožanje temperature

Leta 1966 so pri ameriškem zveznem koordinatorju za meteorologijo ugotovili, da meteorologi po državi v svojih poročilih ne zaokrožajo odčitanih temperatur na cele stopinje vsi na enak način, zato so izdelali priporočilo, ki določa, kako je treba odčitek (realno število) zaokrožiti na najbližje celo število. Pri tem naj se vrednost, ki je točno na sredini med dvema celima številoma, zaokroži vedno navzgor (t.j. proti višji temperaturi, to velja tudi za negativne odčitke). Tako se npr. vrednost 1,5 zaokroži na 2, vrednost $-1{,}5$ pa na -1.

V deželah, kjer merimo temperaturo v stopinjah Celzija (kjer ničla ustreza ledišču vode in so zato meritve blizu ničle pomembne), je v navadi še dodatno pravilo, ki določa, da temperature med $-0{,}5$ (vključno) in 0 (izključno) stopinjami sicer zaokrožimo na 0, a zapišemo kot „-0“, torej z negativnim predznakom.

Napiši program, ki bo s standardnega vhoda prebiral realna števila, jih zaokrožal na cela števila upoštevaje obe gornji pravili in jih izpisoval.

Lahko predpostaviš, da ti je na voljo funkcija `Odrezi(x)`, ki realnemu številu `x` v argumentu odreže decimalke in vrne celo število. Če želiš, lahko privzameš točen format vhoda, npr. predznak, celi del in dve decimalki (v tem primeru tudi dokumentiraj, kaj si predpostavil o formatu vhoda), ali pa uporabiš standardni način za branje realnih števil v svojem programskem jeziku.

Opomba: če te morda mika, da bi za zaokrožanje uporabil kakšno drugo funkcijo iz svojega izbranega programskega jezika, je treba biti pri tem zelo previden, skrbno prebrati dokumentacijo in utemeljiti odgovor. Večinoma namreč tovrstne funkcije ne izpolnjujejo pogojev te naloge brez dodatne prilagoditve.

2. Najlepši esej

Ministrstvo za lepobesedje je razpisalo natečaj za najlepše eseje. Pri ocenjevanju esejev je določilo naslednje pravilo: esej je lep, če so v njem vse besede dolge od 3 do 8 znakov.

Pomagaj ministrstvu in **napiši program** (ali del programa), ki prebere en esej in izpiše, ali je esej lep (vse besede so znotraj predpisane dolžine). Če esej ni lep, naj izpiše, koliko besed ima krajših od 3 znake in koliko daljših od 8 znakov (za vsak slučaj, če ne bo nobenega eseja, ki bi zadoščal določenemu pogoju, kajti potem se bo posebna komisija odločala med tistimi eseji, ki imajo najmanj besed krajših od 3 znake in daljših od 8 znakov).

Beseda je sestavljena iz malih in velikih črk angleške besede, to so znaki od „`a`“ do „`z`“ in od „`A`“ do „`Z`“. Med besedami so presledki, vejice, pike in drugi znaki, ki niso črke po angleški abecedi. Beseda je vedno v eni vrstici (besede niso nikoli deljene). Tvoja rešitev lahko bere esej s standardnega vhoda ali pa iz datoteke `esej.txt` (karkoli ti je lažje). Posamezne vrstice vhodnega besedila so dolge po največ 100 znakov.

3. Pomanjkanje sendvičev

Vodstvo Caféja Maφja se je odločilo, da lačnim študentom ponudi 6 tipov sendvičev na študentske bone. Prosijo te, da jim pomagaš in **napišeš program** za robota, ki bo prodajal sendviče. Program naj na začetku prebere podatke o trenutni zalogi sendvičev. Nato naj kupce sprašuje po želenih sendvičih in vsakemu sproti postreže, če je sendvič še na zalogi, sicer pa naj izpiše, da ga ni. Program naj se konča, ko kupec zahteva sendvič št. 0. Na koncu naj program izpiše še podatke o najpopularnejšem sendviču (tistem, ki je bil največkrat zahtevan) in izpiše številke sendvičev, ki jih je bilo premalo. Predpostaviš lahko, da v vhodnih podatkih ni napak (npr. zahtevana številka sendviča je vedno od 0 do 6).

Primer:

Program izpiše:	Uporabnik vnese:
`Zaloga sendvicev st. 1:`	`1`
`Zaloga sendvicev st. 2:`	`4`
`Zaloga sendvicev st. 3:`	`0`
`Zaloga sendvicev st. 4:`	`5`
`Zaloga sendvicev st. 5:`	`1`
`Zaloga sendvicev st. 6:`	`365`
`Pozdravljeni, kateri sendvic zelite?`	`1`
`Izvolite sendvic st. 1. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`3`
`Sendvicev tipa 3 nam je zal zmanjkalo. Vec srece prihodnjic!`	
`Pozdravljeni, kateri sendvic zelite?`	`4`
`Izvolite sendvic st. 4. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`4`
`Izvolite sendvic st. 4. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`5`
`Izvolite sendvic st. 5. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`4`
`Izvolite sendvic st. 4. Dober tek!`	
`Pozdravljeni, kateri sendvic zelite?`	`1`
`Sendvicev tipa 1 nam je zal zmanjkalo. Vec srece prihodnjic!`	
`Pozdravljeni, kateri sendvic zelite?`	`0`
`Premalo je bilo sendvicev st. 1 3.`	
`Najbolj popularni so sendvici st. 4.`	

(*Opomba:* zgornji primer je le za ilustracijo, tvoj program lahko izpis oblikuje tudi malo drugače.)

4. Prehod za pešce

Na prehodu za pešce čez prometno cesto se pogosto dogajajo nesreče, ker pešci prečkajo cesto pri rdeči luči. Preden se poda v urejanje prometa, si mestna uprava želi izvedeti, koliko je takšnih nevzgojenih meščanov. Zato so na prehod namestili števec, ki zazna vsako osebo, ki stopi na cestišče. Na tebi pa je, da napišeš program, ki bo opravljal meritve.

Na voljo je funkcija Dogodek, ki vrne celo število, ko se zgodi eden od dogodkov, ki jih zna sistem meriti. Takšni dogodki so trije. Ko se prižge zelena luč, funkcija vrne vrednost 1. Ko se prižge rdeča luč, vrne vrednost 2. Ko stopi pešec na cestišče, vrne vrednost 3. Funkcija čaka (ne vrne ničesar), dokler se ne zgodi ena od teh stvari. Rdeča in zelena luč se prižigata izmenično.

```
function Dogodek: integer;      { v pascalu }
int Dogodek();                  /* v C/C++ in podobnih jezikih */
def Dogodek(): ...              # v pythonu; vrne int
```

Napiši program, ki se bo vrtel v neskončni zanki in bo vsakič, ko se bo prižgala zelena luč, izpisal, koliko pešcev je pri prejšnji rdeči luči stopilo na cesto. Izpisuješ lahko na zaslon ali v datoteko `pesci.txt`, kar ti je lažje.

5. Pike za tisočice

Pri zapisovanju velikih števil pogosto zaradi preglednosti ločimo skupine treh števk s piko: na primer, število 12345 zapišemo kot „12.345“; število 5379473457 zapišemo kot „5.379.473.457“; število 9876,01234 pa zapišemo kot „9.876,01234“ (brez narekovajev). Kot vidimo iz zadnjega primera, vrivamo pike le levo od decimalne vejice, desno od nje pa ne.

Napiši podprogram (funkcijo) IzpisStevila(s), ki kot parameter dobi niz s, ki predstavlja neko število v desetiškem zapisu. V tem nizu nastopajo le znaki od 0 do 9 in mogoče še decimalna vejica. Tvoj podprogram naj dobljeno število izpiše s pikami, kot je bilo to opisano v prejšnjem odstavku.

Tvoj podprogram naj bo takšne oblike:

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

12. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2017

NALOGE ZA DRUGO SKUPINO

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

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

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

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

1. Zvončki

Imamo podano skladbo — zaporedje tonov, ki ga lahko predstavimo kot zaporedje števil med 0 in n. Podanih imamo m skupin po 10 različnih zvončkov, ki so postavljene ena ob drugi. Vsak zvonček igra nek vnaprej določen ton; v različnih skupinah so lahko zvončki, ki igrajo isti ton. Če stojimo pred skupino i, lahko zaigramo ton poljubnega zvončka iz skupin i, $i-1$ in $i+1$ (z nekaj izjemami na robovih: pri $i=1$ skupina $i-1$ ne obstaja, pri $i=m$ pa ne obstaja skupina $i+1$). Sicer se moramo premakniti do neke skupine, da tak zvonček dosežemo. (Premaknemo se lahko do poljubne skupine, ne nujno le do sosednje.) Premikom bi se radi izognili. **Opiši postopek**, ki za dano zaporedje tonov ugotovi, s koliko minimalno premiki ga lahko zaigramo. Začetni položaj si lahko izberemo poljubno.

Primer: da bo manj pisanja, si oglejmo primer, v katerem imamo skupine po 3 zvončke namesto po 10 zvončkov, drugače pa je vse enako kot v zgornjem opisu naloge. Recimo, da imamo naslednjih $m = 9$ skupin:

(*Naloga se nadaljuje na naslednji strani.*)

i	1	2	3	4	5	6	7	8	9
zvončki	1	5	7	2	9	8	8	7	6
v sku-	2	1	3	4	4	6	9	0	5
pini i	3	4	8	5	2	1	3	5	3

In recimo, da bi radi zaigrali naslednje zaporedje devetih tonov:

$$7,\ 2,\ 3,\ 9,\ 0,\ 6,\ 5,\ 3,\ 2.$$

Potem je najmanjše potrebno število premikov enako 2. Dobimo ga na primer tako, da začnemo pri $i = 2$, kjer zaigramo tone 7, 2 in 3; nato se premaknemo na $i = 8$, kjer zaigramo tone 9, 0, 6 in 5; in nato se premaknemo na $i = 1$, kjer zaigramo še tona 3 in 2.

2. Rastlinjak

V rastlinjaku je postavljenih nekaj merilnikov temperature zraka, ki svojo vsakokratno meritev prek radijskega oddajnika pošiljajo osrednji nadzorni postaji. Meritve izvajajo in pošiljajo periodično večkrat na uro. Merilniki so samostojni in med seboj niso časovno usklajeni, tudi čas med zaporednimi meritvami ni posebno natančen, tako da se vrstni red prejetih meritev lahko tudi spreminja. Ker se zaradi časovne neusklajenosti ali radijskih motenj lahko zgodi, da bi se kakšna meritev izgubila, je vsak merilnik nastavljen tako, da svojo vsakokratno meritev pošlje trikrat zapored v kratkem časovnem razmaku (od tega se lahko kakšna ponovitev tudi izgubi na poti do sprejemnika, drugih napak pri prenosu pa ni). Vsako oddano sporočilo vsebuje oznako (številko) merilnika in odčitano temperaturo.

Vsak merilnik je opremljen z enolično številko merilnika med 1 in 9 (vseh merilnikov je največ devet). Sprejemna postaja vsako prejeto sporočilo opremi še s časom, ko je bilo sprejeto, in ga zapiše na izhod kot vrstico, ki vsebuje tri polja, ločena s presledkom: čas, številka merilnika in temperatura. Vse ponovitve sporočil so opremljene z enakim časom. Izhod sprejemne postaje je povezan z vhodom v računalnik, tako da program, ki se tam izvaja, lahko sproti bere vrstice prek svojega standardnega vhoda, kot prihajajo iz sprejemnika.

Napiši program, ki bo z vhoda bral zaporedne meritve in sproti izpisoval vsak nov odčitek temperature, skupaj s številko merilnika — takoj, ko se nov odčitek pojavi. Ponovljenih kopij iste meritve naj program ne izpisuje.

Primer vhodnih podatkov:

```
2017-02-23T09:58:38 1 11.0
2017-02-23T09:58:38 1 11.0
2017-02-23T09:58:38 1 11.0
2017-02-23T09:59:00 3 15.1
2017-02-23T09:59:00 2 14.7
2017-02-23T09:59:00 3 15.1
2017-02-23T09:59:00 3 15.1
2017-02-23T09:59:00 2 14.7
2017-02-23T09:59:00 2 14.7
2017-02-23T10:02:21 1 11.0
2017-02-23T10:02:47 3 14.8
2017-02-23T10:02:47 3 14.8
2017-02-23T10:02:47 3 14.8
2017-02-23T10:02:55 2 14.6
```

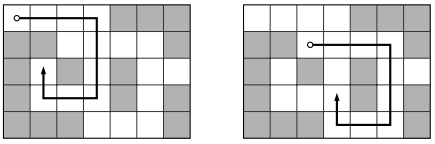
Pripadajoči izpis:

```
1 11.0
3 15.1
2 14.7
1 11.0
3 14.8
2 14.6
```

Tvoja rešitev lahko predpostavi, da so podatki v približno taki obliki, kot jo kaže gornji primer: največ 20 znakov za čas in datum (ta ne more vsebovati presledkov), nato en presledek, en znak s številko merilnika, še en presledek in nato temperatura, zapisana na eno decimalko.

3. Labirint

V nekem trenutku Miha odpre oči in ugotovi, da se nahaja sredi labirinta. Ne ve točno, kje v labirintu se nahaja, najde pa v žepu svoj mobilni telefon. Ker ima Miha predplačniški paket, ugotovi, da ima dobroimetja le še za 100 sporočil SMS. Odloči se, da bo za vsak svoj premik za eno polje v labirintu prijateljem poslal sporočilo SMS. Miha v vsakem sporočilu SMS (ki vsebuje le en znak) prijateljem napiše, v katero smer se je trenutno premaknil (`S` = premaknil sem se v smeri severa, `J` = v smeri juga, `V` = vzhoda, `Z` = zahoda). Čez čas se Miha utrudi hoditi po labirintu in prijateljem pošlje SMS z znakom `@`. Miha se takrat ustavi in čaka na trenutni poziciji. Mihovi prijatelji pa so si med tem priskrbeli zemljevid labirinta in hočejo Mihu pomagati ugotoviti, kje točno v labirintu se nahaja. Poskušajte jim pri tem pomagati in **napišite program**, ki prebere opis labirinta in Mihovih premikov ter izpiše vse možne koordinate Mihovega trenutnega položaja.



Na obeh slikah je enak labirint in pot, ki ustreza enakemu zaporedju premikov (tri korake na vzhod, tri na jug, dva na zahod in enega na sever). Kot vidimo iz slik, je tako zaporedje premikov možno pri dveh različnih začetnih položajih, zato tudi Mihovega sedanjega (končnega) položaja ne moremo enolično določiti.

Vhodni podatki: prva vrstica vhoda vsebuje dve naravni števili h in w (med 3 in 100), ki označujeta velikost labirinta. Labirint ima obliko pravokotne kariraste mreže, sestavljene iz h vrstic in w stolpcev. V sledečih h vrsticah vhoda je predstavljen labirint v obliki tabele ničel in enic. Ničle označujejo prosta polja v labirintu, enice pa neprehodna polja (zidove). Od $(h+2)$-ge vrstice naprej pa se po vrsti nahaja vsebina vseh sporočil SMS, ki jih je Miha poslal prijateljem (v vsaki vrstici po eno sporočilo).

Izhodni podatki: program naj izpiše koordinate možnih položajev (v poljubnem vrstnem redu), vsakega v svoji vrstici v obliki x y. Pri tem x pomeni številko stolpca (1 je najbolj levi, w je najbolj desni stolpec), y pa številko vrstice (1 je najbolj zgornja vrstica, h je najbolj spodnja vrstica).

Tvoj program lahko za branje in pisanje uporablja standardni vhod in izhod ali pa datoteke (karkoli ti je lažje).

Primer vhodne datoteke:

```
5 7
0 0 0 0 1 1 1
1 1 0 0 0 0 1
1 0 1 0 1 0 0
1 0 0 0 1 0 1
1 1 1 0 0 0 1
V
V
V
J
J
J
Z
Z
S
@
```

Pripadajoča izhodna datoteka:

```
2 3
4 4
```

4. Šifriranje

Tajni agent Janez šifrira svoja sporočila s ključem dolžine 5 znakov po naslednjih pravilih:

- Ključ sestavljajo same male črke angleške abecede (od `a` do `z`).
- S prvo črko ključa šifrira 1., 6., 11., 16. itd. znak sporočila; z drugo črko ključa šifrira 2., 7., 12., 17. itd. znak sporočila; itd.
- Šifriranje posameznega znaka sporočila (recimo z) z neko črko ključa (recimo c) poteka takole: če z ni črka angleške abecede, ostane pri šifriranju nespremenjen; sicer pogledamo, na katerem mestu v abecedi je tista črka c (tako nastane neko število n od 1 do 26), in nato znak z ciklično zamaknemo za n mest naprej po abecedi. Na primer, če je $c =$ `d`, nastane iz njega $n = 4$ in pri cikličnem zamiku za 4 mesta se znak `a` zašifrira v `e`, znak `b` v `f`, itd., znak `v` v `z`, znak `w` v `a`, znak `x` v `b`, znak `y` v `c` in znak `z` v `d`.

Dobil si zašifrirano sporočilo, ključa pa ne poznaš. Toda veš, da se Janez, ki pošilja sporočila, vedno na koncu podpiše „`Lp, Janez`“, tako da bodo to gotovo zadnji znaki v sporočilu. **Napiši program**, ki odšifrira sporočilo in ga izpiše. Tvoj program lahko prebere sporočilo s standardnega vhoda ali pa iz datoteke `sporocilo.txt` (kar ti je lažje). Predpostaviš lahko, da je celotno sporočilo v eni sami vrstici, dolgi največ 10 000 znakov.

Primer sporočila pred in po šifriranju, če za ključ uporabimo besedo `labod`:

Nešifrirano sporočilo:	`Resi se, kdor se more! Lp, Janez`
Znaki ključa:	`labodlabodlabodlabodlabodlabodla`
Šifrirano sporočilo:	`Dfux ef, oppt wq odvq! At, Lprqa`

5. Neskončna pokrajina

Z Daljnega vzhoda smo uvozili poceni 3D skener — napravo, ki zna meriti višino točk na neki pravokotni površini. Žal pa so izdelovalci izdelavo programskega vmesnika prepustili najcenejšemu podizvajalcu (pa še tega so očitno poiskali nekje med učenci lokalne osnovne šole), zato nam je na voljo le ena sama funkcija za merjenje — Visina(x, y). Tej pošljemo celoštevilski koordinati točke, ki ji želimo izmeriti višino, vrne pa nam — seveda — izmerjeno višino. Funkcija je takšne oblike:

```
function Visina(x, y: integer): integer;   { v pascalu }
int Visina(int x, int y);                  /* v C/C++ in podobnih jezikih */
def Visina(x, y): ...                      # v pythonu; vrne int
```

Drugih informacij nismo dobili, zato ne vemo niti, kaj pomenita števili, ki ju pošljemo v funkcijo (v kakšnih enotah sta podani). S poskušanjem smo ugotovili le, da vrednosti ne smeta biti negativni (potem se program sesuje) in da je izmerjena vrednost vedno večja ali enaka 0. S postavljanjem različnih elementov na začetno koordinato smo tudi hitro ugotovili, kako program meri višino. Ne uspe pa nam umeriti koordinatnega sistema, ker je genialni programer funkcijo Visina napisal tako, da ob neveljavnih pozitivnih koordinatah vrne vrednost 0. Zato bomo problem predali tebi.

Napiši podprogram Poisci(ciljnaVisina), ki bo poiskal in izpisal koordinate (x, y) polja, ki je visoko toliko kot podana vrednost (ciljnaVisina). Edino zagotovilo, ki ga imaš, je, da zagotovo obstajata nenegativni koordinati x in y, pri katerih bo funkcija Visina(x, y) vrnila vrednost ciljnaVisina. Če ti je lažje, lahko koordinate zapišeš v datoteko, namesto da bi jih izpisal na zaslon.

Upoštevaj, da je v mreži lahko veliko polj višine 0, torej se ne moreš zanašati na to, da rezultat 0 pomeni, da si funkcijo Visina poklical z neveljavnimi koordinatami. Upoštevaj tudi, da je delovanje skenerja relativno počasno in da zato ne moreš poklicati funkcije Visina kar za vse možne celoštevilske koordinate (torej za vse take koordinate, ki jih je mogoče predstaviti z vrednostjo tipa integer oz. **int**).

12. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2017

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Delovno okolje

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

Na vsakem računalniku imaš na voljo disk `D:\`, v katerem lahko kreiraš svoje datoteke in imenike. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali VB.NET, mi pa jih bomo preverili s 64-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz OpenJDK 9 in s prevajalnikom Mono 4.2 za C# in VB.NET. Za delo lahko uporabiš Lazarus (IDE za pascal), `gcc`/`g++` (GNU C/C++ — command line compiler), `javac` (za javo 1.8), Visual Studio, Eclipse in druga orodja.

Na spletni strani `http://rtk2017.fri1.uni-lj.si/` boš dobil nekaj testnih primerov.

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

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

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

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih (praviloma desetih). Za vsak testni primer se bo izpisalo, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund ali pa porabil preveč pomnilnika, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru. (Omejitev pomnilnika je 250 MB v pascalu, C in C++ ter 500 MB v javi, C# in VB.NET.)

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

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih; pri 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://rtk2017.fri1.uni-lj.si/`):

- V pascalu:

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

- V C-ju:

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

- V C++:

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

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

- V javi:

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

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

- V C#:

```
using System.IO;

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

- V Visual Basic.NETu:

```
Imports System.IO

Module Poskus

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

End Module
```

12. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2017

NALOGE ZA TRETJO SKUPINO

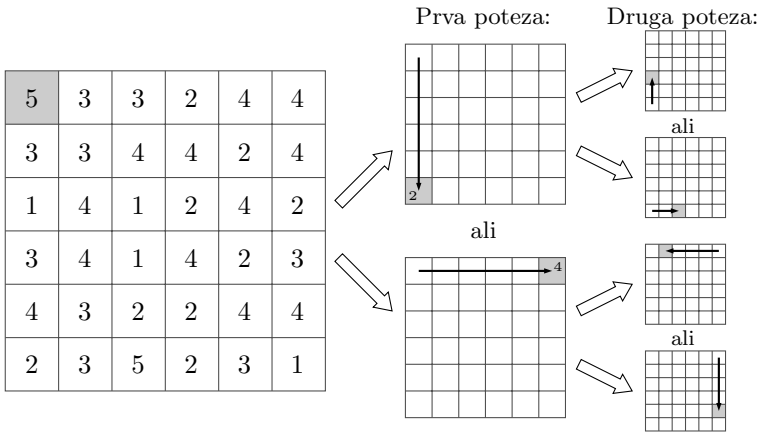
Rešitve bodo objavljene na `http://rtk.ijs.si/`.

1. Back from the Klondike (`klondike.in`, `klondike.out`)

Na igralni plošči $w \times h$ polj je v vsakem polju zapisano neko celo število, večje ali enako 1. Igrati začnemo v zgornjem levem kotu in želimo priti v spodnji desni kot. Skozi vso igro velja, da se iz polja, na katerem se trenutno nahajamo, lahko vedno premaknemo samo v eno od štirih smeri (levo, desno, gor, dol) po naslednjih pravilih:

- Premakniti se moramo vedno za natanko toliko polj, kolikor znaša zapisano število v polju, v katerem ravnokar stojimo.
- V izbrano smer se lahko premaknemo le, če je v tej smeri še najmanj toliko polj do roba plošče, kolikor znaša zapisano število v polju, v katerem stojimo.

Primer (sivo polje označuje naš trenutni položaj):



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

Napiši program, ki prebere opis mreže in izračuna najmanjše število potez, s katerim je mogoče priti iz začetnega v končno polje.

Vhodna datoteka: v prvi vrstici sta dve celi števili, w (širina mreže) in h (višina mreže), ločeni s presledkom. Veljalo bo $1 \leq w \leq 1000$ in $1 \leq h \leq 1000$. Sledi h vrstic, ki opisujejo mrežo; v vsaki od njih je w pozitivnih celih števil, ločenih s po enim presledkom.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število potez, s katerim je mogoče (po pravilih iz besedila naloge) priti iz zgornjega levega kota mreže v spodnji desni kot mreže. Če takšna pot sploh ne obstaja, izpiši `-1`.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
13
```

2. Trojane (`trojane.in`, `trojane.out`)

Tekmovalcu Rajku s Štajerske se neznansko mudi na računalniško tekmovanje. Zjutraj je zaspal, zdaj pa poskuša ujeti začetek tekmovanja za vsako ceno. No, za skoraj vsako ceno. Rajko ve, da je na Trojanah polno relacijskih radarjev, ki mu bodo pri prehitri vožnji prinesli točke zvestobe in izpraznili bančni račun. Na radiu je slišal podatke o vseh radarjih, zdaj pa ga zanima, v kakšnem najkrajšem času lahko prevozi Trojane, ne da bi dobil kakšno kazen. Rajka sicer ne skrbi za varnost ali cestnoprometne predpise.

Trojane so cesta, dolga m kilometrov. Rajko se na začetku nahaja na začetku (torej na točki 0) in bi rad čim hitreje prišel do konca (torej do točke m). Njegov avto pri najvišji hitrosti en kilometer prevozi v u sekundah. Na Trojanah se nahaja n radarjev. Vsak radar ima začetno točko s_i, končno točko t_i (zanju velja $0 \leq s_i < t_i \leq m$) in v_i, to je čas, ki ga mora avto porabiti na odseku, da je njegova povprečna hitrost dovolj nizka, da ne dobi kazni. Če torej Rajko odsek od s_i do t_i prevozi v manj kot v_i sekundah, bo dobil mastno kazen. **Napiši program**, ki izračuna, najmanj koliko časa potrebuje Rajko, da pride čez Trojane, če noče dobiti nobene kazni. Zaviranje in pospeševanje zanemarimo in predpostavimo, da se zgodi v trenutku. Prav tako zanemarimo morebitne fizikalne zakone in geografske značilnosti, ki bi nasprotovali opisu naloge ali vhodnim podatkom.

Vhodna datoteka: v prvi vrstici so podana cela števila m, n in u, ločena s po enim presledkom. Nato sledi n vrstic, v i-ti od njih (za vsak i od 1 do n) so tri cela števila, s_i, t_i in v_i, ločena s po enim presledkom.

Veljalo bo $n \leq 10^6$, $m \leq 10^9$, $u \leq 10^9$ in $v_i \leq 10^9$. Pri 20 % testnih primerov bo veljalo $n \leq 20$ in $m \leq 20$. Pri nadaljnjih 40 % testnih primerov bo veljalo $n \leq 10^3$ in $m \leq 10^3$.

Izhodna datoteka: vanjo izpiši eno samo število, in sicer najmanjši možni čas, ki ga Rajko potrebuje za vožnjo v skladu z zahtevami naloge.

Primer vhodne datoteke:

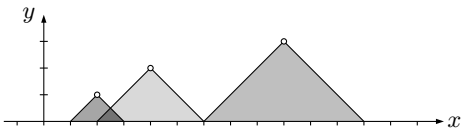
```
8 3 1
1 4 4
3 6 3
5 7 5
```

Pripadajoča izhodna datoteka:

```
12
```

3. V soju žarometov (`zarometi.in`, `zarometi.out`)

Društvo računalnikarjev organizira vsakoletni festival „STOPIE“ za pomoč ljudem, ki še vedno uporabljajo Internet Explorer. Da bi zbrali čim več denarja, so povabili q popularnih glasbenikov, ki so napisali himno festivala in jo bodo ob zaključku skupaj izvedli na odru. Kot odrski tehnik si postavil n reflektorjev na različne položaje in vsakemu nastavil neko intenziteto žarenja. Vsi reflektorji so obrnjeni naravnost proti odru in osvetljujejo stožec s kotom 90°, kot je prikazano na spodnji sliki. Nastopajoči v zaključni točki imajo že predpisane položaje na odru, zanima pa jih, v kako močnem soju žarometov se bo kopal vsak izmed njih. Moč soja žarometov na neki točki na odru je vsota intenzitet vseh žarometov, ki osvetljujejo to točko. (Če se v neki točki stikata dva stožca, jo osvetljujeta oba.) **Napiši program**, ki sprejme postavitev žarometov in njihove intenzitete ter odgovarja na vprašanja nastopajočih.



Vhodna datoteka: v prvi vrstici je dano število žarometov n (zanj velja $1 \le n \le 10^6$). Nato sledi n vrstic; i-ta od njih vsebuje tri cela števila x_i, y_i in c_i, ločena s po enim presledkom, ki predstavljajo x- in y-koordinato i-tega žarometa ter njegovo intenziteto. (Veljalo bo $-10^9 \le x_i \le 10^9$, $0 < y_i \le 10^9$ in $0 < c_i \le 10^9$.) Nato sledi število q, ki predstavlja število poizvedb nastopajočih (veljalo bo $1 \le q \le 10^6$). Sledi q vrstic; j-ta od njih vsebuje celo število p_j, ki pomeni naslednje: pri $j = 1$ je p_j položaj prvega nastopajočega na odru; pri $j > 1$ pa je p_j razlika med položajem j-tega nastopajočega in med rezultatom prejšnje (torej $(j-1)$-ve) poizvedbe. Da dobiš položaj j-tega nastopajočega, moraš torej rezultatu prejšnje poizvedbe prišteti p_j. Za vsak j velja $|p_j| \le 10^{15}$, poleg tega pa za položaj vsakega nastopajočega velja, da je po absolutni vrednosti manjši od $2 \cdot 10^9$.

V 30 % testnih primerov bo veljalo tudi $n \cdot q \le 10^6$.

Izhodna datoteka: za vsako poizvedbo izpiši po eno vrstico, vanjo pa eno samo celo število, namreč vsoto intenzitet vseh žarometov, ki sijejo na položaj nastopajočega, po katerem sprašuje tista poizvedba.

Primer vhodne datoteke:

```
3
4 2 3
9 3 5
2 1 8
6
6
0
8
4
-2
-6
```

Pripadajoča izhodna datoteka:

```
8
5
0
3
8
11
```

4. Najkrajša pot (`pot.in`, `pot.out`)

V Digitalnem kraljestvu se nahaja n mest (označena so s številkami od 1 do n), ki so med sabo povezana z več cestami. (Vsaka cesta neposredno povezuje dve mesti in je dvosmerna.)

Kraljeva palača se nahaja v mestu s številko 1, poletna rezidenca pa v mestu s številko n. Vsako poletje se kralj želi odpraviti iz palače v poletno rezidenco in priti tja po poti, ki uporabi kar se da majhno število različnih cest.

Obenem pa želi vsako leto v poletno rezidenco iti po drugi poti.

V kraljestvu bodo letos zgradili novo cesto. **Napiši program**, ki za dano omrežje cest izračuna, koliko je največje možno število različnih *najkrajših* poti med mestoma 1 in n, potem ko omrežju dodamo novo cesto. (Najkrajša pot je tista, ki uporabi najmanj cest.)

Dve poti sta različni, če se razlikujeta v katerikoli cesti.

Pozor: posamezna mesta so lahko med sabo povezana z več različnimi cestami. Tudi novo cesto lahko dodaš med takim parom mest, ki sta že od prej povezani z eno ali več cestami. Ni nujno, da se dá po obstoječih cestah sploh priti od palače do poletne rezidence.

Vhodna datoteka: v prvi vrstici sta podani celi števili n in m, ločeni s presledkom. Pri tem je n število mest, m pa je število parov mest, ki so povezana z eno ali več cestami. Nato sledi m vrstic, v vsaki od njih pa so podana tri cela števila a_i, b_i in c_i, ločena s po enim presledkom; to pomeni, da sta mesti a_i in b_i povezani s c_i cestami.

Veljalo bo $2 \leq n \leq 10^6$, $1 \leq m \leq 10^6$ in $1 \leq c_i \leq 10^9$. Pri 20 % testnih primerov bo veljalo $n \leq 10$ in $m \leq 10$, rezultat pa ne bo presegal 10^6. Pri nadaljnjih 20 % testnih primerov bo veljalo $n \leq 10^3$ in $m \leq 10^3$. Pri nadaljnjih 20 % testnih primerov bo veljalo, da je $m = n - 1$ in da med vsakim parom mest obstaja ena pot.

Izhodna datoteka: tvoj program naj izpiše zgolj največje število najkrajših poti med mestoma 1 in n, ki ga lahko dobimo, če dodamo eno povezavo. Vsi testni primeri pri tej nalogi bodo sestavljeni tako, da to število ne bo presegalo 10^{18}.

Primer vhodne datoteke:

```
7 9
1 2 3
1 3 2
1 4 4
1 5 8
2 4 2
3 4 2
4 5 1
6 7 10
5 7 1
```

Pripadajoča izhodna datoteka:

```
18
```

5. Listi (`listi.in`, `listi.out`)

Na mizi imamo razmetanih n pravokotnih listov papirja različnih dimenzij. Ker na mizi zasedajo precej prostora, bi radi naredili red in jih zložili na enega ali več kupov. Za vsak list poznamo njegovo širino in višino. Listov na smemo obračati, da ostane besedilo na njih pravilno orientirano. Liste bomo zložili na kup tako, da bodo imeli poravnan levi spodnji kot. Tako je površina, ki jo zaseda posamezen kup listov, enaka produktu največje širine in največje višine listov s tega kupa. Skupna zasedena površina pa je enaka vsoti površin, ki jih zasedajo posamezni kupi. **Napiši program**, ki izračuna, kakšna je minimalna zasedena površina, če liste optimalno zložimo v kupe, in kakšno je minimalno število kupov, s katerim lahko to dosežemo.

Vhodna datoteka: v prvi vrstici datoteke se nahaja število listov $n \leq 10\,000$. Sledi n vrstic, ki opisujejo posamezne liste. Vsak list je opisan s širino w in višino h ($1 \leq w \leq 10^9$ in $1 \leq h \leq 10^9$), ki sta ločeni s presledkom. V 50 % testnih primerov bo veljalo $n \leq 20$.

Izhodna datoteka: vanjo izpiši dve celi števili, ločeni s presledkom: najprej minimalno zasedeno površino in nato minimalno število kupov.

Primer vhodne datoteke:

```
4
20 40
2 50
100 20
10 30
```

Pripadajoča izhodna datoteka:

```
2900 3
```

Komentar: v optimalni rešitvi gornjega primera bosta prvi in četrti list tvorila en kup, drugi in tretji list pa bosta vsak sam na svojem kupu. Tako imamo tri kupe, zasedena površina pa je $20 \cdot 40 + 2 \cdot 50 + 100 \cdot 20 = 2900$.

12. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2017

REŠITVE NALOG ZA PRVO SKUPINO

1. Zaokrožanje temperature

Temperature bomo brali v zanki in jih sproti zaokrožali ter izpisovali. Trenutni temperaturi t najprej s funkcijo Odrezi odrežimo decimalke; dobljeno celo število imenujmo u.

Če je $t > 0$, se pri rezanju decimalk zmanjša ali pa ostane nespremenjen (če je bil že t sam po sebi celo število); v tem primeru je torej u vrednost, ki jo dobimo, če t zaokrožimo navzdol. Razlika $t - u$ je tedaj torej ≥ 0. Če je ta razlika $\geq 1/2$, bi morali po pravilih naloge zaokrožiti t navzgor, zato v tem primeru u povečamo za 1.

Če pa je bil $t < 0$, se pri rezanju decimalk poveča ali ostane nespremenjen (na primer: iz $-2{,}7$ nastane -2, to pa je večje od $-2{,}7$), torej je u takrat vrednost, ki jo dobimo, če t zaokrožimo navzgor. Razlika $t - u$ je tedaj torej ≤ 0. Če je ta razlika $< -1/2$, bi morali po pravilih naloge zaokrožiti t navzdol, zato v tem primeru u zmanjšamo za 1.

Zdaj lahko izpišemo u, paziti moramo le še na primere, ko je t negativen in se je zaokrožil na $u = 0$; takrat moramo najprej izpisati še minus, da bo nastal izpis -0 namesto le 0.

```
#include <stdio.h>

int main()
{
  double t;
  while (1 == scanf("%lf", &t)) /* Preberimo temperaturo. */
  {
    int u = Odrezi(t); /* Odrežimo ji decimalke. */
    /* Po potrebi popravimo zaokrožanje v pravo smer. */
    if (t - u >= 0.5) u++;
    else if (t - u < -0.5) u--;
    /* Izpišimo zaokroženo število, pazimo na -0. */
    if (t < 0 && u == 0) printf("-");
    printf("%d\n", u);
  }
}
```

2. Najlepši esej

Vhodno besedilo lahko beremo po znakih; dokler beremo črke, v spremenljivki dolzina štejmo dolžino (doslej prebranega dela) trenutne besede. Ko pridemo do znaka, ki ni črka, vemo, da je trenutne besede konec. (Paziti moramo na to, da pravilno zaznamo tudi konec zadnje besede; spodnji program se pri tem opira na dejstvo, da funkcija fgetc iz standardne knjižnice takrat vrne EOF, kar tudi ni črka.) Zdaj torej poznamo njeno pravo dolžino in lahko preverimo, če je prekratka (krajša od 3 znake) ali predolga (daljša od 8 znakov); v tem primeru tudi povečamo spremenljivki prekratke in predolge, ki štejeta prekratke in predolge besede. Nato pa postavimo spremenljivko dolzina nazaj na 0, da bomo pripravljeni na branje naslednje besede. Na koncu vemo, da je esej lep le, če sta spremenljivki prekratke in predolge obe enaki 0.

```
#include <stdio.h>

int main()
{
  int prekratke = 0, predolge = 0, dolzina = 0, c;
```

```
  do
  {
    c = fgetc(stdin); /* Preberimo naslednji znak. */
    /* Če smo prebrali črko, povečajmo števec, ki meri dolžino trenutne besede. */
    if ('A' <= c && c <= 'Z' || 'a' <= c && c <= 'z')
      dolzina++;
    /* Sicer smo na koncu besede. */
    else if (dolzina > 0)
    {
      /* Po potrebi povečajmo števca prekratkih in predolgih besed. */
      if (dolzina > 8) predolge++;
      else if (dolzina < 3) prekratke++;
      dolzina = 0; /* Pripravimo se na naslednjo besedo. */
    }
  }
  while (c != EOF);
  /* Izpišimo rezultate. */
  if (prekratke > 0 || predolge > 0)
    printf("Esej ni lep: %d prekratkih, %d predolgih besed.\n", prekratke, predolge);
  else
    printf("Esej je lep.\n");
  return 0;
}
```

3. Pomanjkanje sendvičev

Koristno je imeti dve tabeli: v eni hranimo podatke o zalogi sendvičev posameznega tipa, v drugi pa število zahtev po sendvičih posameznega tipa. To drugo tabelo bomo potrebovali na koncu, da bomo lahko izpisali, kateri tip sendviča je bil največkrat zahtevan. Glede prve tabele pa se lahko vprašamo, ali naj v njej hranimo trenutno stanje zaloge (po vseh dosedanjih zahtevah) ali začetno stanje zaloge (tisto, ki smo ga na začetku izvajanja prebrali od uporabnika). Druga možnost pride bolj prav, ker drugače na koncu ne bomo vedeli, katerih sendvičev je bilo premalo (če bi videli le to, da je stanje zaloge tistega tipa sendvičev na koncu 0, iz tega še ne bi vedeli, ali je bil ta sendvič zahtevan kdaj po tistem, ko mu je zaloga že padla na 0 — šele to pa naredi razliko med tem, ali je bilo sendvičev tega tipa premalo ali pa ravno prav).

Na začetku torej preberimo zalogo in jo shranimo v tabelo zaloga, v tabeli s števci zahtev (stZahtev) pa inicializiramo vse elemente na 0. Nato po vrsti beremo zahteve in povečujemo števce zahtev, pri vsaki pa tudi preverimo, če je število zahtev tega tipa zdaj že preseglo začetno zalogo, tako da vemo, kaj odgovoriti uporabniku. Na koncu lahko s primerjavo obeh tabel ugotovimo, katerih sendvičev je bilo premalo, najpopularnejši tip sendviča pa ugotovimo tako, da poiščemo največjo vrednost v tabeli stZahtev in nato izpišemo tiste indekse, kjer se v tabeli pojavlja ta vrednost (tako bomo pravilno odkrili tudi primere, ko obstaja več enako popularnih najpopularnejših tipov).

```
#include <stdio.h>

int main()
{
  enum { N = 6 }; /* Število tipov sendvičev. */
  /* Preberimo zalogo in inicializirajmo števce zahtev na 0. */
  int zaloga[N], stZahtev[N];
  for (int t = 0; t < N; t++)
  {
    stZahtev[t] = 0;
    printf("Zaloga sendvicev st. %d: ", t + 1);
    scanf("%d", &zaloga[t]);
  }
  /* Prebirajmo zahteve in odgovarjajmo nanje. */
  while (true)
  {
    printf("Pozdravljeni, kateri sendvic zelite? ");
```

```
    int t; scanf("%d", &t); if (t == 0) break;
    /* Popravimo števec zahtev tega tipa in preverimo,
       če so sendviči tega tipa še na zalogi. */
    if (++stZahtev[t - 1] > zaloga[t - 1])
      printf("Sendvicev tipa %d nam je zal zmanjkalo. Vec srece prihodnjic!\n", t);
    else
      printf("Izvolite sendvic st. %d. Dober tek!\n", t);
  }
  /* Izpišimo, katerih sendvičev je bilo premalo. */
  printf("Premalo je bilo sendvicev st.");
  for (int t = 0; t < N; t++)
    if (stZahtev[t] > zaloga[t]) printf(" %d", t + 1);
  /* Izpišimo, po katerih sendvičih je bilo največ zahtev. */
  int naj = 0; for (int t = 0; t < N; t++) if (stZahtev[t] > naj) naj = stZahtev[t];
  printf(".\nNajbolj popularni so sendvici st.");
  for (int t = 0; t < N; t++) if (stZahtev[t] == naj) printf(" %d", t + 1);
  printf(".\n"); return 0;
}
```

Pri delu s tabelami je treba nekaj previdnosti pri indeksih: indeksi v tabeli gredo od 0 do 5, uporabnik pa vnaša števila od 1 do 6. Ena možnost je, da primerno prištevamo ali odštevamo 1, da preračunavamo med obema načinoma številčenja tipov sendvičev (to počne na primer gornja rešitev), druga možnost pa bi bila, da bi deklarirali tabeli s po 7 elementi, ki bi imeli torej indekse od 0 do 6, pri čemer elementa z indeksom 0 potem pač ne bi uporabljali.

4. Prehod za pešce

Potrebovali bomo spremenljivko (v spodnji rešitvi je to stevec), v kateri štejemo pešce, ki so prečkali cesto. Ko se prižge rdeča luč, postavimo ta števec na 0; ko prečka cesto kak pešec, povečamo števec za 1; ko se prižge zelena, pa števec izpišemo. (Tako se bo števec sicer povečeval tudi pri tistih pešcih, ki prečkajo cesto pri zeleni luči, vendar nas to nič ne moti, saj ga bomo tako ali tako postavili nazaj na 0, ko se bo naslednjič prižgala rdeča luč.)

```
#include <stdio.h>

int main()
{
  int stevec = 0;
  while (true)
  {
    int dogodek = Dogodek();
    if (dogodek == 1) printf("%d\n", stevec);
    else if (dogodek == 2) stevec = 0;
    else stevec++;
  }
}
```

5. Pike za tisočice

Za začetek se sprehodimo v zanki po nizu in poiščimo indeks n, na katerem se v njem pojavlja decimalna vejica; če pa te sploh ni, bomo namesto nje za n vzeli dolžino niza. Zdaj lahko razmišljamo takole: piko za tisočice moramo vriniti pred znake $n-3$, $n-6$, $n-9$ in tako nazaj. Z drugimi besedami, pred znak i moramo vriniti piko natanko tedaj, če je razlika $n-i$ večkratnik števila 3. Izjema je le prvi znak niza ($i = 0$), pred katerega pike ne vrivamo. Zdaj se lahko sprehodimo še enkrat po nizu od začetka do konca; pri vsakem indeksu preverimo, če moramo pred trenutni znak vriniti piko; če da, jo izpišemo, nato pa v vsakem primeru izpišemo še trenutni znak.

```
#include <stdio.h>

void IzpisStevila(const char *s)
```

```
{
  /* Poiščimo decimalno vejico ali konec niza. */
  int n = 0;
  while (s[n] && s[n] != ',') n++;

  /* Izpišimo ustrezno popravljen niz. */
  for (int i = 0; s[i]; i++)
  {
    /* Če smo levo od decimalne vejice (vendar ne na začetku niza)
       in je število znakov med trenutnim položajem in decimalno vejico
       večkratnik 3, vrinemo piko za tisočice. */
    if (i > 0 && i < n && (n - i) % 3 == 0) fputc('.', stdout);

    /* Izpišimo trenutni znak. */
    fputc(s[i], stdout);
  }
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Zvončki

Če s trenutnega položaja lahko zaigramo naslednji ton naše melodije, potem ni razloga, da bi se pred igranjem tega tona kam premaknili; kamorkoli se že imamo namen premakniti, se lahko tja premaknemo tudi po tem tonu, njega pa zaigramo še s trenutnega položaja. Ta razmislek nam pove, da moramo o premikih razmišljati šele takrat, ko s trenutnega položaja ne dosežemo zvončka, ki ga potrebujemo za naslednji ton (torej ko ni primernega zvončka niti v skupini, pred katero stojimo, niti v njeni levi ali desni sosedi).

Ko pa se vendarle moramo premakniti, si moramo nekako izbrati novi položaj. Takrat si je pametno novi položaj izbrati tako, da se nam potem čim dlje ne bo treba spet premikati. O tem se lahko prepričamo takole. Recimo, da primerjamo dva položaja, a in b; naj bo n_a število naslednjih tonov melodije, ki jih bomo lahko odigrali s položaja a, in podobno n_b za položaj b; in recimo, da je $n_a > n_b$. Ali je mogoče, da bi se bilo vendarle bolje zdaj premakniti na b namesto na a? Če se premaknemo na b, bomo tam zaigrali naslednjih n_b tonov, potem pa se bomo morali spet premakniti, recimo k neki skupini c. Toda če se namesto tega zdaj premaknemo na a, bomo tudi tam lahko zaigrali naslednjih n_b tonov (pravzaprav celo n_a tonov, kar je več kot n_b), potem pa se lahko še vedno premaknemo na c, če se že hočemo. Torej gotovo nismo nič na slabšem, če se premaknemo na a namesto na b.

Z enakim razmislekom si izberemo tudi začetni položaj; postavimo se k tisti skupini, pri kateri bomo lahko zaigrali največ tonov z začetka melodije, preden se bomo morali prvič premakniti.

Za čim učinkovitejše preverjanje tega, ali lahko nek ton zaigramo z določenega položaja, bi si lahko pred začetkom našega postopka pripravili razpršeno tabelo (*hash table*), v kateri bi kot ključe shranili pare $\langle$*številka skupine*, *zvonček*$\rangle$ za vse zvončke vseh skupin; tako bi lahko v $O(1)$ časa preverili, ali je iskani zvonček v trenutni skupini (ali pa v eni od sosednjih dveh skupin). Če n (število različnih zvončkov) ni prevelik, pa bi lahko namesto razpršene tabele pripravili kar bitno karto — tabelo $n \times m$ bitov, ki bi nam za vsako kombinacijo tona in skupine povedali, ali je v tisti skupini zvonček, ki igra tisti ton.

2. Rastlinjak

Naloga zahteva, da vsako meritev izpišemo le prvič, ko se pojavi, njene ponovitve pa ignoriramo. Ker se lahko v vhodnih podatkih prepletajo meritve z različnih merilnikov, si moramo za vsak merilnik zapomniti čas zadnje meritve, ki smo jo dobili od njega (spodnji program ima v ta namen tabelo zadnjiCas). Ko pride nova meritev, jo primerjamo s časom zadnje meritve istega merilnika in če sta enaka, vemo, da gre za ponovitev že videne meritve. Če pa te meritve še nismo videli, jo lahko zdaj izpišemo, njen čas pa si zapomnimo v zadnjiCas.

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

int main()
{
  enum { N = 9 };
  string zadnjiCas[N];
  while (true)
  {
    string cas, temp; int n;

    /* Preberimo naslednjo meritev. */
    cin >> cas >> n >> temp;
    if (! cin.good()) break; /* Najbrž smo na koncu vhoda. */

    /* Ali smo to meritev že videli? */
    if (zadnjiCas[n - 1] == cas) continue;

    /* Če ne, jo zapišimo in si jo zapomnimo. */
```

```
    cout << n << " " << temp << endl;
    zadnjiCas[n - 1] = cas;
  }
  return 0;
}
```

3. Labirint

Med branjem Mihovih premikov je koristno vzdrževati seznam njegovih možnih trenutnih položajev (v spodnjem programu bo to tabela kandidati, število kandidatov v njej pa hrani spremenljivka stKand). Na začetku, preden se prvič premakne, ne vemo o njegovem položaju ničesar drugega kot to, da je trenutno na nekem prostem polju, torej na seznam kandidatov dodamo vsa prosta polja. To lahko počnemo spotoma, medtem ko beremo opis labirinta; ta opis si tudi zapomnimo v neki tabeli, ker ga bomo kasneje še potrebovali.

Nato beremo premike enega po enega; pri vsakem premiku si pripravimo par števil $(\triangle x, \triangle y)$, ki pove, kako se pri tem premiku spremenita Mihovi koordinati. Preglejmo vse kandidate v seznamu in pri vsakem izračunamo novi položaj po tem premiku. Če bi bil novi položaj na neprehodnem polju ali pa zunaj labirinta, kandidata pobrišemo iz seznama (ker očitno Miha ni začel svoje poti na tistem začetnem položaju, iz katerega je ta kandidat nastal).

Ko pridemo do konca zaporedja premikov, moramo seznam kandidatov za Mihov trenutni položaj le še izpisati. Pri izpisu pazimo na to, da naloga šteje koordinate od 1 naprej, za indekse v tabele pa je koristno delati s koordinatami od 0 naprej.

Spodnji program hrani kandidata (x, y) kot celo število $x \cdot w + y$, iz česar ni težko nazaj izračunati x in y; lahko pa bi namesto tega hranili tudi majhne strukture z dvema ločenima atributoma za x in y.

```
#include <stdio.h>

int main()
{
  enum { MaxW = 100, MaxH = 100 };
  int w, h, lab[MaxH][MaxW], kandidati[MaxW * MaxH], stKand = 0;

  /* Preberimo opis labirinta in pripravimo seznam kandidatov za
     Mihov trenutni položaj. Na začetku so to kar vsa prosta polja. */
  scanf("%d %d", &h, &w);
  for (int y = 0; y < h; y++) for (int x = 0; x < w; x++) {
    scanf("%d", &lab[y][x]);
    if (! lab[y][x]) kandidati[stKand++] = y * w + x; }

  /* Preberimo Mihove premike in sproti popravljajmo seznam kandidatov. */
  while (true)
  {
    /* Poglejmo, za kakšen premik gre. */
    char premik[5]; scanf("%s", premik);
    int dx, dy;
    if (premik[0] == 'S') dx = 0, dy = -1;
    else if (premik[0] == 'J') dx = 0, dy = 1;
    else if (premik[0] == 'V') dx = 1, dy = 0;
    else if (premik[0] == 'Z') dx = -1, dy = 0;
    else break;

    /* Preglejmo vse kandidate in jih ustrezno popravimo. */
    for (int i = 0; i < stKand; )
    {
      int x = kandidati[i] % w, y = kandidati[i] / w;
      x += dx; y += dy;

      /* Zdaj sta x in y koordinati, kot bi ju Miha imel po tem premiku.
         Če ta položaj ni veljaven, kandidata pobrišimo, sicer pa
         na njegovo mesto v tabeli vpišimo novi položaj. */
      if (x >= 0 && x < w && y >= 0 && y < h && ! lab[y][x])
        kandidati[i++] = y * w + x;
```

```
        else
          kandidati[i] = kandidati[--stKand];
      }
    }
    /* Izpišimo rezultate. */
    for (int i = 0; i < stKand; i++)
      printf("%d %d\n", kandidati[i] % w + 1, kandidati[i] / w + 1);
    return 0;
}
```

4. Šifriranje

Označimo prvotni niz (pred šifriranjem) s p, šifriranega pa s s; recimo, da sta dolga po n znakov, indekse znakov pa štejmo od 0 do $n-1$, tako kot je to v navadi v C-ju in podobnih jezikih.

Pri šifriranju je posamezni znak $s[i]$ nastal tako, da smo $s[i]$ ciklično zamaknili za $k[i \bmod 5]$ mest naprej; pri tem si $k[0..4]$ predstavljajmo kot številsko predstavitev našega ključa (kjer ima ključ črko `a`, naj ima tabela k vrednost 1, kjer ima ključ črko `b`, naj ima tabela k vrednost 2 itd.). Če si znake nizov s in p namesto kot črke predstavljamo kot števila od 0 do 25, lahko šifriranje opišemo s preprosto formulo:

$$s[i] = (p[i] + k[i \bmod 5]) \bmod 26.$$

Zadnjih pet (pravzaprav celo zadnjih osem) znakov niza p poznamo, ker vemo, da se sporočilo konča na „`Lp, Janez`“; niz s pa poznamo v celoti. Za $i = n-5, \ldots, n-1$ torej poznamo tako $s[i]$ kot $p[i]$, zato lahko iz gornje formule izrazimo znake ključa:

$$k[i \bmod 5] = (s[i] - p[i]) \bmod 26.$$

Ker bomo to naredili za pet zaporednih vrednosti i, bodo ostanki i mod 5 gotovo pokrili vse vrednosti od 0 do 4, tako da bomo dobili cel ključ.

Podobno lahko iz prve formule izrazimo $p[i]$ in tako dobimo formulo za dešifriranje, ki jo bomo lahko uporabljali, ko bomo poznali ključ:

$$p[i] = (s[i] - k[i \bmod 5]) \bmod 26.$$

V praksi je treba biti pri uporabi zadnjih dveh formul malo previden. Obe sta oblike $(x-y) \bmod 26$; če je razlika $x-y$ negativna, operator za računanje ostankov po deljenju — na primer % v C-ju in podobnih jezikih — praviloma ne bo dajal rezultatov, kot jih tu pričakujemo.[1] Zato je bolje namesto $x-y$ vzeti $x-y+26$, kar ima enak ostanek po deljenju s 26, je pa zagotovo večje ali enako 0.

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

void Desifriraj(const char *s)
{
  const char konec[] = "Janez"; /* Zadnjih 5 znakov dešifriranega niza. */
  char kljuc[5];
  int n = strlen(s);

  /* Rekonstruirajmo številsko predstavitev ključa. */
  for (int i = n - 5; i < n; i++)
    kljuc[i % 5] = (s[i] - konec[i - (n - 5)] + 26) % 26;

  /* Dešifrirajmo znake niza in jih izpisujmo. */
  for (int i = 0; i < n; i++)
  {
    char c = s[i]; int k = 26 - kljuc[i % 5];

    /* Če je c črka, jo ciklično zamaknimo za k mest,
       sicer naj c ostane nespremenjen. */
```

[1] Za več o tem glej npr. rešitev naloge 1997.2.1, str. 309 v zbirki *Rešenih nalog s srednješolskih računalniških tekmovanj 1988–2004*.

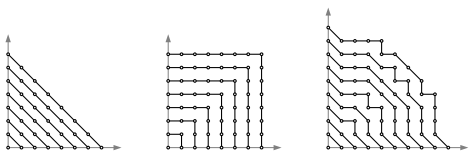
```
    if (c >= 'A' && c <= 'Z') c = (c - 'A' + k) % 26 + 'A';
    else if (c >= 'a' && c <= 'z') c = (c - 'a' + k) % 26 + 'a';
    fputc(c, stdout);
  }
  fputc('\n', stdout);
}

int main()
{
  char s[10001]; fgets(s, sizeof(s), stdin);
  Desifriraj(s); return 0;
}
```

5. Neskončna pokrajina

Če bi poznali velikost naše pravokotne površine, bi lahko vse celoštevilske pare (x, y) na njej pregledali tako, da bi šli sistematično npr. po vrsticah od zgoraj navzdol, v vsaki vrstici pa od leve proti desni; tako bi sčasoma obiskali vse točke v pravokotniku. Ker pa njegove velikosti ne poznamo, bi ta postopek odpovedal že v prvi vrstici, saj ne moremo vedeti, kdaj lahko z njo končamo in se premaknemo v naslednjo vrstico.

Namesto tega moramo najti nek postopek, ki obiskuje točke (x, y) za vse nenegativne celoštevilske x in y v takem vrstnem redu, da pride vsaka točka na vrsto po končno mnogo korakih. Nekaj možnosti kaže naslednja slika:



Verjetno najpreprostejša je prva rešitev, ki pregleduje točke po diagonalah. Za vsak $d \geq 0$ imamo diagonalo od točke $(0, d)$ do $(d, 0)$. Točke na njej imajo koordinate oblike $(x, d - x)$. Vse, kar potrebujemo, je zanka po d in v njej še vgnezdena zanka po x:

```
bool Preizkusi(int x, int y, int ciljnaVisina)
{
  if (Visina(x, y) != ciljnaVisina) return false;
  printf("%d %d\n", x, y); return true;
}

void PoisciPoDiagonalah(int ciljnaVisina)
{
  for (int d = 0; ; d++)
    for (int x = 0; x <= d; x++)
      if (Preizkusi(x, d - x, ciljnaVisina)) return;
}
```

Klic funkcije Visina in izpis rezultata (če smo našli pravo točko) smo preselili v ločen podprogram Preizkusi zato, ker nam bo to prihranilo nekaj ponavljanja približno iste kode tudi v naslednjih rešitvah.

Druga možnost na gornji sliki je, da si predstavljamo ravnino kot zaporedje vse večjih kvadratov s spodnjim levim kotom v $(0, 0)$ in zgornjim desnim v (a, a) za vse večje a. Kvadrat s stranico a se od svojega predhodnika, kvadrata s stranico $a - 1$, razlikuje po tem, da pokrije tudi točke (a, t) in (t, a) za $t = 0, 1, \ldots, a$. Lahko gremo torej v zanki po a in pri vsakem a obiščemo vse tiste točke, ki jih ima kvadrat s stranico a, njegov predhodnik s stranico $a - 1$ pa ne:

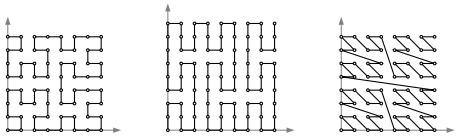
```
void PoisciPoGnomonih(int ciljnaVisina)
{
  for (int a = 0; ; a++)
  {
    for (int y = 0; y <= a; y++)
      if (Preizkusi(a, y, ciljnaVisina)) return;
```

```
    for (int x = a − 1; x >= 0; x−−)
      if (Preizkusi(x, a, ciljnaVisina)) return;
  }
}
```

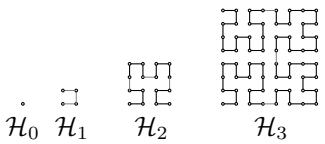
Če namesto vse večjih kvadratov pri takšnem razmisleku uporabimo vse večje kroge (s središčem v koordinatnem izhodišču), dobimo tretjo sliko zgoraj. Namesto stranice kvadrata a zdaj gledamo polmer kroga r; pri polmeru r moramo obiskati vse točke, ki ležijo na kolobarju med krogoma s polmerom $r-1$ in r (lahko tudi na krogu s polmerom r, ne pa na krogu s polmerom $r-1$, kajti tiste smo obiskali že prej). Tak pregled lahko začnemo v točki $(r, 0)$, nato pa razmišljamo takole: iz trenutne točke (x, y) se premaknemo v $(x-1, y)$, če ta točka leži v našem kolobarju; če pa ne, se premaknemo v $(x, y+1)$; če tudi ta ne leži v kolobarju (ker je od koordinatnega izhodišča oddaljena za več kot r), pa se premaknemo v $(x-1, y-1)$.

```
void PoisciPoKolobarjih(int ciljnaVisina)
{
  for (int r = 0; ; r++)
  {
    int x = r, y = 0;
    while (x >= 0)
    {
      if (Preizkusi(x, y, ciljnaVisina)) return;
      if ((x − 1) * (x − 1) + y * y > (r − 1) * (r − 1)) { x−−; continue; }
      y++; if (x * x + y * y > r * r) x−−;
    }
  }
}
```

Pri pregledovanju vseh točk ravnine si lahko pomagamo tudi s fraktalnimi krivuljami. Naslednja slika kaže nekaj primerov: Hilbertovo, Peanovo in Z-krivuljo.

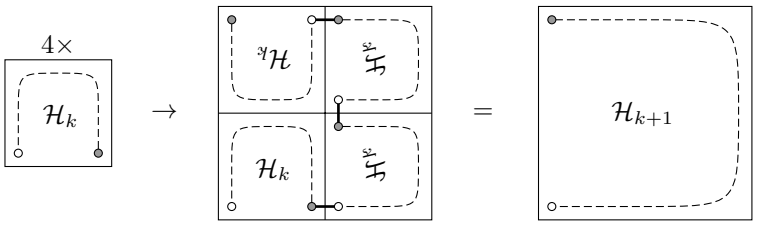


Primer na prvi od teh slik je začetni del Hilbertove krivulje, ene od najbolj znanih fraktalnih krivulj. Dobimo jo tako, da začnemo z eno samo točko, nato pa na vsakem koraku vzamemo štiri kopije prejšnje krivulje in jih povežemo (te povezave so na spodnji sliki prikazane z tanjšimi črtami):



Na sliki vidimo krivulje od $\mathcal{H}_0$ do $\mathcal{H}_3$, očitno pa bi lahko na ta način nadaljevali še poljubno dolgo.

Vidimo lahko, da krivulja $\mathcal{H}_k$ obišče vse tiste točke, ki imajo obe koordinati z območja od 0 do $2^k - 1$. Krivulja $\mathcal{H}_k$ se začne v točki $(0, 0)$, konča pa v $(2^k - 1, 0)$, če je k sod, oz. v $(0, 2^k - 1)$, če je k lih. Pri sodem k lahko štiri kopije krivulje $\mathcal{H}_k$ takole zložimo v $\mathcal{H}_{k+1}$:



Enak razmislek je uporaben tudi pri lihem k, če like pred in po zlaganju prezrcalimo prek simetrale lihih kvadrantov (z drugimi besedami, če zamenjamo x- in y-koordinate).

Tako lahko sestavimo naslednji postopek, ki nam za poljuben $n \geq 0$ izračuna koordinati n-te točke na Hilbertovi krivulji:

```
void TockaNaHilbertoviKrivulji(int n, int &x, int &y)
{
  int k = 0, u = 1; x = 0; y = 0;
  while (n > 0)
  {
    int xx, yy, t;
    if (k & 1) t = x, x = y, y = t;
    switch (n & 3)
    {
    case 0: xx = x; yy = y; break;
    case 1: xx = u + y; yy = x; break;
    case 2: xx = u + y; yy = u + x; break;
    case 3: xx = u - 1 - x; yy = 2 * u - 1 - y; break;
    }
    x = xx; y = yy;
    if (k & 1) t = x, x = y, y = t;
    n >>= 2; u <<= 1; k++;
  }
}
```

Bite n-ja pregledujemo od nižjih proti višjim; na vsakem koraku odčitamo naslednja dva bita n-ja, ki nam povesta, v kateri od štirih kopij krivulje $\mathcal{H}_k$ se nahaja naša točka. Glavni podprogram mora le računati točke v zanki in jih preverjati, dokler ne najde prave:[2]

```
void PoisciPoHilbertoviKrivulji(int ciljnaVisina)
{
  for (int n = 0; ; n++)
  {
    int x, y; TockaNaHilbertoviKrivulji(n, x, y);
    if (Preizkusi(x, y, ciljnaVisina)) return;
  }
}
```

Peanovo krivuljo (druga med tremi fraktalnimi krivuljami zgoraj) dobimo na podoben način, le da na vsakem koraku namesto štirih kopij prejšnje krivulje vzamemo devet kopij in jih zložimo v kvadrat 3×3, v katerem si sledijo v obliki črke N. Pri tem moramo tiste v srednjem stolpcu ($\mathsf{nx} = 1$ v spodnjem podprogramu) prezrcaliti prek vodoravne osi, tiste v srednji vrstici ($\mathsf{ny} = 1$) pa prek navpične osi, da bo krivulja lepo zvezna. (Za namen naše naloge to sicer ni nujno potrebno; naloga zahteva le, da obiščemo vse točke ravnine.) Tako dobimo naslednjo rešitev:

```
void TockaNaPeanoviKrivulji(int n, int &x, int &y)
{
  int u = 1; x = 0; y = 0;
  while (n > 0)
  {
    int ny = n % 3; n /= 3; int nx = n % 3; n /= 3;
    if (nx == 1) y = u - 1 - y, ny = 2 - ny;
    if (ny == 1) x = u - 1 - x;
    x += u * nx; y += u * ny; u *= 3;
  }
}

void PoisciPoPeanoviKrivulji(int ciljnaVisina)
{
```

[2] S Hilbertovo krivuljo smo se na naših tekmovanjih že srečali; gl. nalogo 2002.3.7, str. 497 v zbirki *Rešenih nalog s srednješolskih računalniških tekmovanj 1988–2004*.

```
  for (int n = 0; ; n++)
  {
    int x, y; TockaNaPeanoviKrivulji(n, x, y);
    if (Preizkusi(x, y, ciljnaVisina)) return;
  }
}
```

Ostane nam še tretja izmed zgoraj omenjenih fraktalnih krivulj, to je Z-krivulja. Do nje lahko pridemo zelo preprosto: če nas zanima n-ta točka (za $n \geq 0$) na krivulji, zapišimo n v dvojiškem zapisu, torej kot zaporedje bitov: $n = (n_{k-1}n_{k-2}\ldots n_1n_0)_2$. Če je treba, vrinimo na levi še eno vodilno ničlo, tako da je k (število bitov v tem zaporedju) sod. Zdaj to zaporedje preprosto razpletimo na dve: iz bitov na sodih mestih sestavimo x-koordinato $x = (n_{k-2}n_{k-4}\ldots n_2n_0)_2$, iz bitov na lihih mestih pa y-koordinato $y = (n_{k-1}n_{k-3}\ldots n_3n_1)_2$. Tako dobimo naslednjo rešitev:

```
void TockaNaZKrivulji(int n, int &x, int &y)
{
  x = 0; y = 0;
  for (int b = 1; n > 0; b <<= 1)
  {
    if (n & 1) x |= b; n >>= 1;
    if (n & 1) y |= b; n >>= 1;
  }
}

void PoisciPoZKrivulji(int ciljnaVisina)
{
  for (int n = 0; ; n++)
  {
    int x, y; TockaNaZKrivulji(n, x, y);
    if (Preizkusi(x, y, ciljnaVisina)) return;
  }
}
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Back from the Klondike

V nalogi se skriva problem najkrajše poti v grafu, pri čemer dolžino poti merimo le s številom korakov na njej (številom povezav, vse povezave pa štejejo za enako dolge). Zato jo lahko rešujemo z iskanjem v širino.

V tabeli `d` bomo za vsako doseženo polje hranili dolžino najkrajše poti od začetnega položaja (zgornjega levega polja) do njega; poleg tega pa imamo še vrsto (v spodnjem programu je to tabela `q`), v kateri hranimo polja, do katerih že poznamo najkrajšo pot, nismo pa še pregledali, kam lahko iz tega polja pridemo v naslednjem koraku.

V zanki jemljemo po eno polje iz vrste in zanj pregledamo, kam lahko pridemo iz njega v enem koraku. Možni premiki so največ štirje (za vsako smer po eden, če z njim ne bi padli iz mreže). Pri vsakem premiku pogledamo, če nas je pripeljal v neko tako polje, do katerega prej še nismo prišli; če je tako, si zapomnimo dolžino te poti v tabeli `d` in dodamo novo polje na konec vrste `q`. Tako bomo sčasoma pregledali vsa tista polja, do katerih je iz začetnega polja sploh mogoče priti.

Na koncu tega postopka imamo v tabeli `d` dolžine najkrajših poti od začetnega polja do vseh ostalih, do katerih je iz njega sploh mogoče priti; pri nedosegljivih poljih pa je v `d` še vedno vrednost -1, s katero smo to tabelo na začetku inicializirali.

```
#include <stdio.h>

const int DX[] = { -1, 1, 0, 0 }, DY[] = { 0, 0, -1, 1 };
enum { MaxW = 1000, MaxH = 1000 };
int a[MaxH][MaxW], d[MaxW][MaxH], q[MaxW * MaxH];

int main()
{
  /* Preberimo vhodne podatke in inicializirajmo tabelo dolžin poti. */
  FILE *f = fopen("klondike.in", "rt");
  int w, h; fscanf(f, "%d %d", &w, &h);
  for (int y = 0; y < h; y++) for (int x = 0; x < w; x++) {
    fscanf(f, "%d", &a[y][x]); d[y][x] = -1; }
  fclose(f);

  /* Na začetku poznamo le pot dolžine 0 do polja v zgornjem levem kotu. */
  d[0][0] = 0; int glava = 0, rep = 0; q[rep++] = 0;

  /* Od tam nadaljujmo z iskanjem v širino. */
  while (glava < rep)
  {
    int x = q[glava] % w, y = q[glava] / w; glava++;

    /* Poskusimo se iz (x, y) premakniti v vse štiri smeri za a[y][x]. */
    for (int smer = 0; smer < 4; smer++)
    {
      /* Izračunajmo novi položaj (xx, yy) po tem premiku. */
      int xx = x + DX[smer] * a[y][x], yy = y + DY[smer] * a[y][x];

      /* Ali je novi položaj sploh znotraj mreže? */
      if (xx < 0 || yy < 0 || xx >= w || yy >= h) continue;

      /* Ali smo to polje dosegli že kdaj prej? */
      if (d[yy][xx] >= 0) continue;

      /* Zapomnimo si dolžino poti do (xx, yy) in ga dodajmo v vrsto. */
      d[yy][xx] = d[y][x] + 1; q[rep++] = yy * w + xx;
    }
  }
  /* Izpišimo rezultat. */
  f = fopen("klondike.out", "wt"); fprintf(f, "%d\n", d[h - 1][w - 1]); fclose(f); return 0;
}
```

2. Trojane

Recimo, da bi se ves čas peljali z maksimalno hitrostjo (torej 1 km na u sekund). Kakšne težave bi imeli zaradi tega? Edini trenutek, ko imamo lahko zaradi take vožnje težave,

je tedaj, ko dosežemo končno točko t_i kakšnega radarja. Če se takrat izkaže, da je od trenutka, ko smo prečkali točko s_i (torej začetno točko istega radarja), minilo manj kot v_i časa, moramo tik pred točko t_i malo počakati, dokler ta čas ne mine (lahko si predstavljamo, da avtomobil ustavimo ali pa ga vsaj upočasnimo na neko zelo nizko hitrost).

Da bomo simulirali tak potek vožnje, je koristno zložiti začetne in končne točke vseh omejitev v en seznam (v spodnjem programu je to vektor tocke) in ga urediti po koordinati. Če na isti x-koordinati ležita tako začetna točka ene omejitve kot končna točka neke druge omejitve, postavimo v našem urejenem seznamu končno točko pred začetno, kajti končne točke so tiste, ki lahko vplivajo na čas, ob katerem bomo prečkali to x-koordinato (ker moramo zaradi nekaterih končnih točk upočasniti oz. čakati, preden jih dosežemo).

Za vsako točko našega seznama bomo poleg x-koordinate hranili še indeks omejitve, ki ji pripada, in podatek o tem, ali je to začetek ali konec tiste omejitve. V ločeni tabeli (spodnji program ima v ta namen vektor omejitve) pa za vsako omejitev hranimo najzgodnejši čas, ob katerem smemo prečkati končno točko te omejitve. Na začetku bodo v tej tabeli le časi v_i, ko pa prečkamo začetek neke omejitve, bomo ustreznemu elementu te tabele prišteli trenutni čas (dobljena vsota je ravno najzgodnejši čas, ob katerem smemo prečkati končno točko te omejitve).

```
#include <stdio.h>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  struct Tocka { int x, omejitev, zacetek; };
  vector<Tocka> tocke; vector<long long> omejitve;

  /* Preberimo vhodne podatke. */
  int n, m, u;
  FILE *f = fopen("trojane.in", "rt");
  fscanf(f, "%d %d %d", &m, &n, &u);
  tocke.reserve(2 * n); omejitve.resize(n);

  for (int i = 0; i < n; i++)
  {
    int si, ti, vi; fscanf(f, "%d %d %d", &si, &ti, &vi);

    /* Za vsako omejitev dodamo začetno in končno točko v vektor „tocke“. */
    tocke.push_back({si, i, 1}); tocke.push_back({ti, i, 0});

    /* omejitve[i] sprva vsebuje v_i, kasneje pa mu bomo prišteli še čas,
       ob katerem bomo dosegli s_i, tako da bo dobljena vsota pomenila najzgodnejši
       čas, ob katerem smemo prevoziti točko t_i. */
    omejitve[i] = vi;
  }
  fclose(f);

  /* Uredimo točke po koordinati; če jih je več na isti koordinati,
     postavimo končne pred začetne. */
  sort(tocke.begin(), tocke.end(), [] (const Tocka &a, const Tocka &b) {
    if (a.x != b.x) return a.x < b.x; else return a.zacetek < b.zacetek; });

  /* Odsimulirajmo vožnjo. */
  long long cas = 0;
  for (int i = 0; i <= 2 * n; i++)
  {
    int razdalja = (i == 2 * n ? m : tocke[i].x) - (i == 0 ? 0 : tocke[i - 1].x);
    cas += (long long) u * razdalja; /* Čas vožnje od i - 1 do i. */
    if (i < 2 * n) /* Upoštevajmo morebitno čakanje tik pred i. */
    {
      const Tocka &t = tocke[i];
      if (t.zacetek) omejitve[t.omejitev] += cas;
      else cas = max(cas, omejitve[t.omejitev]);
```

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

Oglejmo si še malo temeljitejši dokaz tega, da je naša požrešna rešitev res optimalna. Recimo, da obstaja še nek drug potek, ki doseže konec odseka (točko m) prej kot naš in je tudi skladen z vsemi omejitvami. Našemu poteku (kot ga sestavi zgoraj opisana rešitev) recimo P, temu drugemu pa P'.

Začetne in končne točke omejitev nam razbijejo naš odsek $[0, m]$ na krajše intervale (kot smo videli tudi v gornji rešitvi): $0 = x_0 < x_1 < \ldots < x_k = m$. Opazimo lahko, da na skladnost poteka z omejitvami nič ne vpliva to, kako ta potek vozi znotraj takega intervala, ampak le to, ob katerem času prečka krajišča intervalov.

Gotovo pri nekem i velja, da P' doseže t_i prej kot P, kajti če to ne bi veljalo, potem bi oba poteka dosegla konec zadnje omejitve istočasno (ali pa P' celo kasneje kot P), od tam pa P vozi ves čas z maksimalno hitrostjo, torej ni mogoče, da bi P' dosegel cilj pred njim. Točka t_i pa je seveda ena od x_j za nek j. Vzemimo zdaj najmanjši tak j, pri katerem P' doseže x_j prej kot P. Gotovo je $j > 0$, saj sta začela (pri $x_0 = 0$) oba ob istem času 0. Pri $j - 1$ torej še velja, da oba poteka prečkata x_{j-1} ob istem času. Zakaj je torej P za pot od x_{j-1} do x_j porabil več časa kot P'?

P ves čas vozi z največjo možno hitrostjo, razen če mora čakati tik pred neko t_i; torej je edina možnost, zakaj je P' prevozil interval $[x_{j-1}, x_j]$ hitreje kot P, ta, da je moral P pred točko x_j čakati zato, ker je bila ta točka konec ene ali več omejitev, torej oblike $x_j = t_i$. Toda začetek vsake take omejitve, s_i, leži še levo od x_j, torej je to še ena od tistih točk, ki sta jih oba poteka prevozila istočasno, recimo ob času u_i. Potek P je čakal le tako dolgo, da je točko x_j prečkal ob času $\max_i\{u_i + v_i\}$, pri čemer gre i po vseh tistih omejitvah, ki se končajo v x_j. Ker je P' prečkal začetke vseh teh omejitev ob istih časih u_i kot potek P, to pomeni, da tudi točke x_j ne sme prečkati prej kot ob času $\max_i\{u_i + v_i\}$. Tako smo prišli v protislovje: najprej smo rekli, da je P' skladen z omejitvami; nato smo ugotovili, da doseže x_j prej kot P; zdaj pa smo videli, da je to slednje mogoče le, če prekrši neko omejitev. Torej ni mogoče, da bi obstajal tak P', ki bi bil skladen z omejitvami in bi dosegel konec odseka hitreje kot P.

3. V soju žarometov

Žaromet v točki (x_i, y_i) osvetljuje na tleh interval x-koordinat od vključno $x_i - y_i$ do vključno $x_i + y_i$; ta žaromet prispeva po c_i k osvetljenosti vsake točke tega intervala. Če se torej v mislih premikamo po x-osi od leve proti desni, lahko do sprememb v osvetljenosti tal pride le pri tistih x-koordinatah, kjer se začne ali konča kakšen izmed teh intervalov. Tako lahko x-os razdelimo na območja s konstantno osvetljenostjo; pri primeru iz besedila naloge (in na tamkajšnji sliki) bi na primer dobili interval $(-\infty, 1)$ z osvetljenostjo 1, nato interval $[-1, 2)$ z osvetljenostjo 8, nato $[2, 3]$ z osvetljetnostjo 11, nato $(3, 6)$ z osvetljenostjo 3, nato točko $x = 6$ z osvetljenostjo 8, nato interval $(6, 12]$ z osvetljenostjo 5 in končno interval $(12, \infty)$ z osvetljenostjo 0.

Za učinkovito odgovarjanje na poizvedbe je koristno, če si pripravimo tak seznam intervalov, urejen po x-koordinati, potem pa lahko pri vsaki poizvedbi z bisekcijo hitro poiščemo interval, na katerem leži točka, po kateri sprašuje ta poizvedba. V ta namen najprej sestavimo seznam parov $\langle x$-koordinata, sprememba intenzitete$\rangle$ (vsak žaromet prispeva dva taka para, enega za začetek in enega za konec svojega intervala) in jih uredimo po x-koordinati. Nato gremo po tem seznamu (po naraščajočih x-koordinatah), sproti seštevamo spremembe v intenziteti in vsakič, ko se x-koordinata res spremeni, začnemo nov interval.

V točkah, kjer se stikata dva ali več stožcev, je potrebno nekaj pazljivosti. Lahko si na primer pri vsaki taki točki poleg intenzitete intervala, ki se začne v tej točki, zapomnimo tudi skupno intenziteto stožcev, ki se končajo v tej točki. Če potem pride poizvedba, ki pade prav na to točko (in ne na interval med njo in naslednjo točko), potem vemo, da jo osvetljujejo tudi tisti stožci, ki se v tej točki končajo, in moramo rezultatu prišteti tudi njihovo intenziteto.

V spodnji rešitvi so intervali opisani kot zaporedje struktur tipa Tocka; vsaka taka struktura t pove, da se na x-koordinati t.x začenja interval z osvetljenostjo t.c (ki se razteza do naslednje točke v zaporedju), točko t.x sámo pa dodatno osvetljujejo še stožci, ki se končajo v njej in ki imajo skupno intenziteto t.k.

```
#include <stdio.h>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("zarometi.in", "rt"), *g = fopen("zarometi.out", "wt");
  int n; ok_= fscanf(f, "%d", &n);
  vector<pair<int, int>> v; v.reserve(2 * n);
  for (int i = 0; i < n; i++)
  {
    int x, y, c; fscanf(f, "%d %d %d", &x, &y, &c);
    /* Za vsak žaromet dodajmo v seznam obe krajišči;
       pri začetnem pomnožimo intenziteto z −1. */
    v.push_back({x − y, −c}); v.push_back({x + y, c});
  }
  /* Uredimo krajišča po x. */
  sort(v.begin(), v.end());
  /* Združimo krajišča z istim x. Pri vsakem x si tudi zapomnimo
     novo intenziteto (ki velja desno od tega x) in še to, kakšna
     je skupna intenziteta žarometov, ki so se pri tistem x končali. */
  struct Tocka { int x; long long c, k; };
  vector<Tocka> tocke;
  long long intenziteta = 0;
  for (int i = 0; i < v.size(); )
  {
    Tocka t; t.x = v[i].first; t.k = 0;
    for ( ; i < v.size() && v[i].first == t.x; i++) {
      int c = v[i].second; intenziteta −= c; if (c > 0) t.k += c; }
    t.c = intenziteta; tocke.push_back(t);
  }
  /* Odgovarjajmo na poizvedbe. */
  int q; fscanf(f, "%d", &q);
  long long rezultat = 0;
  while (q−−)
  {
    long long p; fscanf(f, "%lld", &p); p += rezultat;
    if (p < tocke[0].x)
      rezultat = 0; /* p je levo od vseh stožcev. */
    else
    {
      /* Z bisekcijo poiščimo interval, ki vsebuje točko p. */
      int L = 0, D = (int) tocke.size();
      while (D − L > 1)
      {
        /* Na tem mestu velja: tocke[L].x ≤ p < tocke[D].x
           (če je D = tocke.size(), si mislimo tam x = ∞). */
        int M = (L + D) / 2;
        if (tocke[M].x <= p) L = M; else D = M;
      }
      /* p torej leži na tocke[L].x ≤ p < tocke[L + 1].x. */
      rezultat = tocke[L].c;
      /* Če leži prav na točki L, prištejmo še stožce, ki se v tej točki končajo. */
      if (tocke[L].x == p) rezultat += tocke[L].k;
    }
```

```
    fprintf(g, "%lld\n", rezultat); /* Izpišimo rezultat. */
  }
  fclose(f); fclose(g); return 0;
}
```

4. Najkrajša pot

V nalogi se skriva neusmerjen graf, v katerem mesta predstavljajo točke, ceste pa povezave. Pravzaprav je to multigraf, ker je lahko med dvema točkama tudi več neposrednih povezav. Začetno točko označimo s s, končno pa s t; pri naši nalogi je torej $s = 1$ in $t = n$. Označimo z $d(u, v)$ dolžino najkrajše poti od u do v, s $\#(u, v)$ pa število poti te dolžine od u do v. Ker so ceste v naši nalogi dvosmerne, velja $d(u, v) = d(v, u)$ in $\#(u, v) = \#(v, u)$.

Če začnemo v točki s, lahko z iskanjem v širino izračunamo $d(s, u)$ in $\#(s, u)$ za vse točke u. Postopek iskanja v širino smo videli že v rešitvi prve naloge in bo tukaj zelo podoben, dopolniti ga moramo le še z računanjem števila poti, $\#(s, u)$. Ko iz vrste vzamemo točko u in pregledujemo povezave (ceste), po katerih je mogoče u zapustiti, lahko razmišljamo takole: če vidimo, da obstaja c povezav od u do v, to pomeni, da bi se dalo vsako od $\#(s, u)$ poti dolžine $d(s, u)$ od s do u podaljšati na c načinov v pot dolžine $d(s, u) + 1$ od s do v. Če do v že poznamo kakšno krajšo pot, si s tem ne moremo pomagati, sicer pa lahko zdaj $\#(s, v)$ povečamo za število teh novih poti, torej za $c \cdot \#(s, u)$.

(Če je kakšna točka u nedosegljiva iz s, si lahko predstavljamo $d(s, u) = \infty$ in $\#(s, u) = 0$. V spodnjem programu bomo namesto vrednosti ∞ uporabljali -1.)

Z enakim postopkom lahko izračunamo tudi $d(t, u)$ in $\#(t, u)$ za vse u. (V spodnji rešitvi imamo v ta namen podprogram NajkrajsePoti.)

Naloga zahteva, da dodamo v graf eno povezavo tako, da bo potem število najkrajših poti od s do t čim večje. Recimo, da nova povezava neposredno poveže točki x in y; za namen tega razmisleka se pretvarjajmo, da je povezava usmerjena $x \to y$, o obrnjeni povezavi $y \to x$ pa bomo razmišljali posebej. Najkrajša pot, ki jo je mogoče speljati po novi povezavi $x \to y$, gre torej najprej od s do x, nato po $x \to y$ in nato od y do t, tako da je skupaj dolga $d(s, x) + 1 + d(y, t)$. Pravzaprav taka pot ni ena sama, saj si lahko del poti od s do x izberemo na $\#(s, x)$ načinov, od y do t pa na $\#(y, t)$ načinov, tako da imamo zdaj kar $\#(s, x) \cdot \#(y, t)$ novih poti.

Glede na dolžino teh novih poti ločimo tri primere:

(1) Če je $d(s, x) + 1 + d(y, t) > d(s, t)$, te novi poti ne bodo najkrajše in se torej število najkrajših poti od s do t ne bo nič spremenilo; taka povezava $x \to y$ za nas ni zanimiva.

(2) Druga možnost je, da je nova pot enako dolga kot najkrajša pot doslej, torej $d(s, x) + 1 + d(y, t) = d(s, t)$. V tem primeru bo po novem obstajalo $\#(s, t) + \#(s, x) \cdot \#(y, t)$ najkrajših poti od s do t.

(3) Ostane še možnost, da je nova pot krajša od najkrajše doslej, torej $d(s, x) + 1 + d(y, t) < d(s, t)$. V tem primeru bo po novem pač obstajalo $\#(s, x) \cdot \#(y, t)$ najkrajših poti od s do t.

Načeloma bi lahko pregledali vse možne pare (x, y) in uporabili tistega, pri katerem dobimo po novem največje število najkrajših poti. Težava je, da je točk veliko (do milijon) in si ne moremo privoščiti, da bi se posebej ukvarjali z vsemi možnimi pari točk.

Do učinkovitejše rešitve pridemo z naslednjim razmislekom. Izberimo si le x, ne pa tudi točke y. (Pri tem se omejimo na take x, ki so dosegljivi iz s. Če x sploh ni dosegljiv iz s, potem se tudi poti od s do t pač ne bo dalo speljati prek x.) Tisti y, pri katerih bi zgoraj padli v primer (1), nas tako ali tako ne zanimajo. V primer (2) pademo pri tistih y, za katere velja $d(y, t) = d(s, t) - 1 - d(s, x)$. Med njimi bo največ poti nastalo pri tistem, ki ima med vsemi takimi y največjo vrednost $\#(y, t)$. Koristno bi torej bilo, če bi si za vsako možno oddaljenost δ od t zapomnili maksimum vrednosti $\#(y, t)$ po vseh tistih y, ki imajo $d(y, t) = \delta$. (Spodnji program izračuna in hrani te vrednosti v najSt[δ].) Tako bomo lahko izračunali maksimalno število novih najkrajših poti po vseh takih y, pri katerih pademo v primer (2), ne da bi se nam bilo treba posebej ukvarjati z vsakim od teh y.

Podobno je tudi v primeru (3); da obdelamo tega, si je koristno za vsako δ zapomniti maksimum vrednosti $\#(y,t)$ po vseh tistih y, ki imajo $d(y,t) \leq \delta$. (Spodnji program izračuna in hrani te vrednosti v najStDo[δ].)

Paziti moramo še na možnost, da v prvotnem grafu točka t sploh ni dosegljiva iz s. To pomeni, da do primera (2) sploh ne more priti, do primera (3) pa pride pri vsakem y, iz katerega je dosegljiv t. Ker ima graf n točk, je t dosegljiv iz v v največ $n-1$ korakih (ali pa je nedosegljiv), torej lahko v tem primeru vzamemo $\delta = n-1$.

```
#include <stdio.h>
#include <vector>
using namespace std;

int n;
vector<int> ps, ns; /* prvi sosed, število sosedov */
struct Sosed { int u, c; };
vector<Sosed> sosedi;

/* Za vsako u vrne v d[u] dolžino najkrajše poti od točke „od“ do u,
   v st[u] pa število takih najkrajših poti. Če u ni dosegljiva iz „od“,
   dobimo d[u] = −1 in st[u] = 0. */
void NajkrajsePoti(int od, vector<int> &d, vector<long long> &st)
{
  d.resize(n); st.resize(n);
  for (int u = 0; u < n; u++) d[u] = −1, st[u] = 0;
  vector<int> vrsta; vrsta.push_back(od); d[od] = 0; st[od] = 1;
  int glava = 0;
  while (glava < vrsta.size())
  {
    int u = vrsta[glava++];
    for (int i = ps[u]; i < ps[u] + ns[u]; i++)
    {
      int v = sosedi[i].u, c = sosedi[i].c;
      if (d[v] < 0) { d[v] = d[u] + 1; st[v] = st[u] * c; vrsta.push_back(v); }
      else if (d[v] == d[u] + 1) { st[v] += st[u] * c; }
    }
  }
}

int main()
{
  FILE *f = fopen("pot.in", "rt");
  int m; fscanf(f, "%d %d", &n, &m);
  ps.resize(n); ns.resize(n);
  for (int u = 0; u < n; u++) ns[u] = 0;

  /* Preberimo seznam cest. */
  struct Cesta { int u, v, c; };
  vector<Cesta> ceste; ceste.resize(m);
  for (int i = 0; i < m; i++) {
    Cesta &c = ceste[i]; fscanf(f, "%d %d %d", &c.u, &c.v, &c.c);
    ns[−−c.u]++; ns[−−c.v]++; }
  fclose(f);

  /* Predelajmo ga v sezname sosedov. */
  for (int u = 0, p = 0; u < n; u++) { ps[u] = p; p += ns[u]; ns[u] = 0; }
  sosedi.resize(2 * m);
  for (Cesta c : ceste) {
    sosedi[ps[c.u] + ns[c.u]++] = { c.v, c.c };
    sosedi[ps[c.v] + ns[c.v]++] = { c.u, c.c }; }

  /* Izračunajmo najkrajše poti od s in t do vseh ostalih mest. */
  const int s = 0, t = n − 1;
  vector<int> ds, dt; vector<long long> sts, stt;
  NajkrajsePoti(s, ds, sts); NajkrajsePoti(t, dt, stt);
  int D = ds[t];

  /* Za vsako oddaljenost d poglejmo, katera izmed točk, ki so na
```

```
    oddaljenosti d od t, ima največ poti te dolžine do t.
    To shranimo v najSt[d]. */
  vector<long long> najSt, najStDo;
  najSt.resize(n); for (int d = 0; d < n; d++) najSt[d] = 0;
  for (int u = 0; u < n; u++) {
    int d = dt[u]; if (d < 0) continue;
    if (stt[u] > najSt[d]) najSt[d] = stt[u]; }
  /* V najStDo[d] pa naj bo maksimum vrednosti najSt[0..d]. */
  najStDo.resize(n); for (int d = 0; d < n; d++)
    najStDo[d] = max(najSt[d], d > 0 ? najStDo[d - 1] : 0);
  /* Poiščimo najboljšo rešitev. */
  long long naj = 0;
  for (int u = 0; u < n; u++)
  {
    /* Recimo, da bi želeli dodati neko povezavo (u, v) tako,
       da bi šla nova najkrajša pot od s do t po njej (in najprej skozi u).
       Če u sploh ni dosegljiva iz s, take poti sploh ne bo. */
    int du = ds[u]; if (du < 0) continue;
    /* Poglejmo, kako daleč sme biti v od t-ja, da ne bo takšna
       pot daljša od najkrajše dosedanje poti od s do t. */
    int dv = (D < 0) ? n - 1 : D - 1 - du; if (dv < 0) continue;
    /* Ena možnost je, da vzamemo tako v, da bo nova pot enako dolga
       kot najkrajša doslej. */
    naj = max(naj, sts[u] * najSt[dv] + sts[t]);
    /* Druga možnost je, da bo nova pot krajša. Namesto najStDo[dv - 1]
       lahko vzamemo kar najStDo[dv], kajti če se tidve vrednosti razlikujeta,
       je to zato, ker je najStDo[dv] = najSt[dv], tedaj pa spodnja rešitev
       tako ali tako ne bo boljša od tiste iz prejšnje vrstice, kjer smo
       prišteli tudi sts[t]. */
    naj = max(naj, sts[u] * najStDo[dv]);
  }
  /* Izpišimo rezultat. */
  f = fopen("pot.out", "wt"); fprintf(f, "%lld\n", naj); fclose(f); return 0;
}
```

5. Listi

Uredimo liste padajoče po širini; če je kje več enako širokih, jih uredimo padajoče po višini. Sprehodimo se po tem seznamu; pri vsakem listu poglejmo, če je višji od prejšnjega v seznamu; če ni, ga lahko pobrišemo, kajti prejšnji list je gotovo vsaj tako širok kot trenutni in če je hkrati tudi vsaj tako visok kot trenutni, to pomeni, da lahko trenutni list vedno damo na isti kup kot prejšnjega in se ta kup zaradi njega ne bo nič povečal.

Po tem pregledu nam ostane seznam, v katerem so listi urejeni padajoče po širini, hkrati pa naraščajoče po višini. V tem vrstnem redu jih oštevilčimo od 1 do m (pri tem je m število listov, ki so nam ostali od prvotnih n po morebitnem brisanju nekaterih listov v prejšnjem odstavku). Recimo zdaj, da bi lista i in j dali na isti kup; naj bo $j < i$. Če je med njima na seznamu še nek list k, to pomeni, da velja $j < k < i$; ker so listi urejeni padajoče po širini, je list j vsaj tako širok kot k, in ker so urejeni naraščajoče po višini, je i vsaj tako visok kot k. Če torej lista i in j damo na isti kup, bo ta kup dovolj velik tudi za list k, ne da bi se moral pri tem še kaj povečati. Ta razmislek nam pove, da se lahko omejimo na rešitve, ki tvorijo kupe le iz strnjenih skupin listov (v našem prej opisanem vrstnem redu).

Zdaj lahko nalogo rešujemo z dinamičnim programiranjem. Definirajmo naslednje podprobleme: recimo, da bi radi na kupe čim bolje razdelili le prvih i listov (namesto vseh listov). Skupno površino teh kupov označimo s $f(i)$. Videli smo, da se lahko omejimo na razporede, kjer vsak kup obsega neko strnjeno skupino listov; zadnji kup pri našem podproblemu bo recimo obsegal liste od j do i za nek $j \leq i$. Zadnji kup bo torej velik $w_j \cdot h_i$, pred tem pa imamo neko razbitje prvih $j-1$ listov na kupe, najboljše

tako razbitje pa ima skupno površino $f(j-1)$. Tako torej vidimo:

$$f(i) = \min\{w_j \cdot h_i + f(j-1) \ : \ 1 \le j \le i\}.$$

Robni primer je $f(0) = 0$ (ko ni nobenega lista več, tudi ni treba tvoriti kupov).

Vidimo lahko, da pri izračunu $f(i)$ potrebujemo vrednosti $f(0), \ldots, f(i-1)$, zato je koristno reševati te podprobleme v zanki po naraščajočih i in si rešitve sproti shranjevati v tabelo.

Naloga sprašuje še po najmanjšem številu kupov, s katerimi je mogoče doseči minimalno skupno površino. Označimo to število kupov s $k(i)$. Ko vidimo, pri katerem j je dosežen minimum v formuli za $f(i)$, lahko zaključimo, da je $k(i) = k(j-1) + 1$; če je mogoče enako dober minimalni $f(i)$ doseči pri več različnih j, moramo vzeti med njimi tistega z najmanjšim $k(j-1)$, da bo potem tudi $k(i)$ najmanjši možni.

```
#include <stdio.h>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("listi.in", "rt");
  int n; fscanf(f, "%d", &n);
  struct List { int w, h; };
  vector<List> listi; listi.resize(n);
  for (int i = 0; i < n; i++) { List &L = listi[i]; fscanf(f, "%d %d", &L.w, &L.h); }
  fclose(f);

  /* Uredimo seznam in pobrišimo liste, ki so po obeh dimenzijah
     manjši ali enaki kakšnemu drugemu. */
  sort(listi.begin(), listi.end(), [] (const List& a, const List &b) {
    if (a.w != b.w) return a.w > b.w; else return a.h > b.h; });
  int m = 0;
  for (int i = 0; i < n; i++)
  {
    List L = listi[i];
    if (m > 0 && listi[m - 1].h >= L.h) continue;
    listi[m++] = L;
  }
  listi.resize(m); n = m;

  /* Rešimo podprobleme od manjših proti večjim. */
  vector<long long> povrsina; vector<int> stKupov;
  povrsina.resize(m + 1); stKupov.resize(m + 1);
  povrsina[0] = 0; stKupov[0] = 0; /* Rešitev za 0 listov. */
  for (int i = 0; i < n; i++)
  {
    /* Poiščimo najboljšo rešitev za prvih i + 1 listov (listi 0..i). */
    for (int j = 0; j <= i; j++)
    {
      /* Kako bi bilo, če bi sestavili en kup z listi j..i? */
      long long kand = povrsina[j] + (long long) listi[j].w * listi[i].h;
      if (j > 0 && (kand > povrsina[i + 1] ||
                    kand == povrsina[i + 1] && stKupov[j] + 1 >= stKupov[i + 1])) continue;
      povrsina[i + 1] = kand; stKupov[i + 1] = stKupov[j] + 1;
    }
  }
  /* Izpišimo rezultat. */
  f = fopen("listi.out", "wt"); fprintf(f, "%lld %d\n", povrsina[n], stKupov[n]);
  fclose(f); return 0;
}
```

Viri nalog: prehod za pešce — Primož Gabrijelčič; listi — Tomaž Hočevar; labirint — Branko Kavšek; zvončki, Trojane — Vid Kocijan; neskončna pokrajina — Vid Kocijan in Primož Gabrijelčič; back from the Klondike — Mitja Lasič; najkrajša pot — Matjaž Leonardis in Vid Kocijan; zaokrožanje temperature, rastlinjak — Mark Martinec; najlepši esej — Polona Novak; pomanjkanje sendvičev, šifriranje, v soju žarometov — Jure Slak; pike za tisočice — Janez Brank.

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