6. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2011

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). 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.");
   }
}
```

6. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2011

NALOGE ZA PRVO SKUPINO

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

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev samodejno shranjuje in na zaslonu ti bo pisalo, kdaj se bo shranjevanje naslednjič zgodilo. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) **Zaradi varnosti priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku** (npr. kopiraj in prilepi v Notepad in shrani v datoteko).

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite (bolj učinkovite rešitve dobijo več točk). **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. Vsota

Dano je zaporedje n celih števil, recimo jim $a_1, a_2, \ldots, a_{n-1}, a_n$. Vsa števila so večja od 0. Poleg tega je dano tudi celo število m. **Opiši postopek** (algoritem), ki poišče v tem zaporedju najkrajše tako strnjeno podzaporedje, pri katerem je vsota števil v podzaporedju večja od m. Poišče naj torej tak začetni indeks z in dolžino podzaporedja d, da bo $a_z + a_{z+1} + a_{z+2} + \ldots + a_{z+d-2} + a_{z+d-1} > m$ in da bo d čim manjši.

Tvoj postopek naj bo čim bolj učinkovit, tako da bo hitro našel pravi odgovor tudi pri zelo dolgih zaporedjih (takih z nekaj milijoni členov). Bolj učinkovite rešitve dobijo več točk.

Primer: recimo, da imamo zaporedje

$$5, 6, 5, 6, 9, 9, 9, 6$$

in $m = 24$. Najkrajše strnjeno podzaporedje z vsoto, večjo od m, je dolgo 3 člene: $9 + 9 + 9 = 27$, kar je večje od 24. (Do vsote 25 pri tem primeru ne moremo priti z nobenim strnjenim podzaporedjem, do vsote 26 pa sicer lahko, vendar potrebujemo za to vsaj štiri člene: $6 + 5 + 6 + 9 = 26$.) Pravilni rezultat pri tem primeru je torej $z = 5$, $d = 3$.

Pri tej nalogi ti ni treba pisati programa ali podprograma v kakšnem konkretnem programskem jeziku (lahko pa, če ti je lažje). Tvoj postopek sme predpostaviti, da so števila že podana v nekakšni tabeli ali seznamu, na primer takole:

```
const int n = ...;          /* v C/C++ */
int a[n];

const n = ...;              { v pascalu }
var a: array [0..n − 1] of integer;

int[] a = ...;              // v javi
int n = a.length;

a = [...]                   # v pythonu
n = len(a)
```

2. Lego kocke

Zoran dela v končni kontroli v tovarni legokock. Vrečke s kockami potujejo po tekočem traku, kjer je vsaka vrečka stehtana, predno se zapakira v škatlo. To se zgodi tako, da medtem ko vrečka potuje po tekočem traku, čitalnik črtne kode prebere črtno kodo z vrečke, tehtnica pa jo stehta na miligram natančno. Težo vrečke potem računalnik primerja z minimalno in maksimalno dopustno težo in če ne leži med njima, vrečko odstrani s tekočega traku. Tekoči trak se med tem ne ustavlja in tehtnica neprestano deluje, zato mora program poiskati, kdaj je teža te vrečke na tehtnici največja in potem to težo primerjati z referenčno težo.

Na voljo imamo:

- Tabeli **int** MinTeza[n], MaxTeza[n], ki vsebujeta minimalno in maksimalno dovoljeno težo za vsako vrsto izdelka — indeks je vrednost črtne kode.

Funkcije:

- **int** CrtnaKoda(), ki vrne črtno kodo z vrečke, ki je na tehtnici; če ni vrečke, vrne -1;
- **int** Tehtaj(), ki vrne težo, ki jo trenutno zaznava tehtnica, v miligramih; če na tehtnici trenutno sploh ni vrečke, vrne 0;
- **void** OdstraniVrecko(), ki odstrani vrečko, ki je na tehtnici, s tekočega traku. Ta funkcija se vrne iz klica šele, ko je vrečka popolnoma odstranjena.

(Za te funkcije torej predpostavi, da že obstajajo in jih lahko kličeš iz svojega programa.) **Napiši program**, ki se bo izvajal v neskončni zanki in nadzoroval obnašanje tega dela tekočega traku. Predpostaviš lahko, da vrečke potujejo izredno počasi v primerjavi s hitrostjo delovanja računalnika; da ko vrečka potuje po traku, se teža, ki jo zaznava tehtnica, najprej nekaj časa le povečuje, nato se nekaj časa le zmanjšuje, nato pa je nekaj časa enaka 0 (ko se je dosedanja vrečka že odpeljala mimo tehtnice in nanjo še ni prišla naslednja vrečka); in da je črtna koda vrečke vidna ves čas, ko je ta vrečka na tehtnici.

Še deklaracije v drugih jezikih:

```
{ v pascalu: }
var MinTeza, MaxTeza: array [1..n] of integer;
function CrtnaKoda: integer;
function Tehtaj: integer;
procedure OdstraniVrecko;

# v pythonu:
MinTeza = [...]; MaxTeza = [...]
def CrtnaKoda: ... # vrne int
def Tehtaj: ... # vrne int
def OdstraniVrecko: ...
```

3. Majevska števila

Maji so ljudstvo, živeče v južni Mehiki in severni Srednji Ameriki s tritisočletno zgodovino. Majevska pisava je bila v rabi vse do prihoda Evropejcev in je dolgo predstavljala veliko zagonetko.

Manj zagoneten pa je njihov sistem zapisa števil. Poznali so ničlo in podobno kot mi (za razliko od rimskih številk) uporabljali mestni zapis števil, vendar s številsko osnovo 20 (namesto naše bolj običajne 10). To jim je omogočalo zapisati zelo velika števila, kar je prišlo prav pri obvladovanju astronomije in koledarja.

Posamezne številke torej predstavljajo števila med 0 in 19, zapisana pa so kot skupek pik in črt, pri čemer vsaka pika predstavlja 1 in vsaka črta predstavlja vrednost 5. Maji so sicer običajno pisali črte ležeče in pike naložene na njih, vendar so lahko črte tudi pokončne in pike lahko ležijo tudi pred ali za njimi — da dobimo vrednost, moramo le pošteti vse črte in pike. Za potrebe te naloge bomo zapisali črto (vrednost 5) kot znak „|“, eno piko kot znak „.“, dve piki pa zaradi lepšega izgleda lahko zapišemo kot dvopičje „:“ ali pa kot dve piki „..“. Številko 0 so Maji narisali kot školjko, mi pa si bomo pomagali kar z našo ničlo „0“.

Nekaj primerov, kako lahko zapišemo posamezne številke:

0 = 0
1 = .
2 = .. ali :
3 = ... ali :. ali .:
5 = | ali ::. ali :.: ali .:: ali ..... } ali še z nekaj drugimi kombinacijami znakov „|“, „:“ in „.“, ki tu niso naštete
8 = .:| ali :|.
12 = :||
19 = ::|||

Številke v mestnem zapisu številke ločimo med seboj s presledkom. Osnova je 20, na zadnjem mestu je torej faktor 1, na predzadnjem 20, na naslednjem z zadnje strani $20 \cdot 20 = 400$, na naslednjem $20 \cdot 20 \cdot 20 = 8000$, itd. Torej povsem enako, kot smo navajeni zapisovati v desetiškem sistemu, le da je faktor 20 namesto 10.

Nekaj primerov večjih števil:

Majevsko število	Številke	Vrednost
. 0	1 0	$1 \cdot 20 + 0 \cdot 1 = 20$
: ..	2 2	$2 \cdot 20 + 2 \cdot 1 = 42$
| 0 |:|	5 0 12	$5 \cdot 20 \cdot 20 + 0 \cdot 20 + 12 \cdot 1 = 2012$
. | :| |	1 5 7 5	$1 \cdot 20 \cdot 20 \cdot 20 + 5 \cdot 20 \cdot 20 + 7 \cdot 20 + 5 \cdot 1 = 10145$

Napiši program, ki bo prebral eno majevsko število, zapisano v eni vrstici, kot smo opisali zgoraj (pike, dvopičja in pokončne paličice, presledek loči številke) in izpisal vrednost števila, kot smo navajeni.

4. Kemijske formule

Podana je strukturna formula kemijske spojine (npr.: C_2H_5OH, H_2O, C_{60}). Elementi so podani z eno veliko tiskano črko angleške abecede, formule ne vsebujejo oklepajev. **Napiši podprogram** `void IzpisiAtome(char* s)`, ki bo izpisal, koliko atomov katere vrste je v spojini (vsak element naj izpiše v svoji vrstici, ni pa važno, v kakšnem vrstnem redu jih izpiše).

Primer: če pokličemo

```
IzpisiAtome("C2H5OH");
IzpisiAtome("H2O");
IzpisiAtome("C60");
```

naj se izpiše:

```
C 2
H 6
O 1

H 2
O 1

C 60
```

Še deklaracije v drugih jezikih:

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

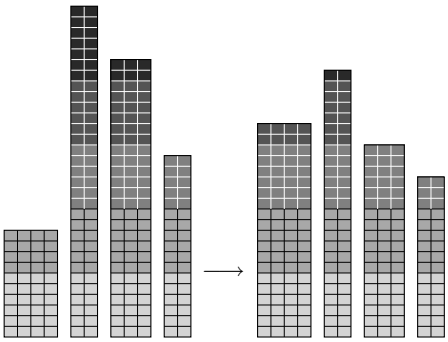
5. Okoljevarstveni ukrepi

V Butalah je začelo primanjkovati elektrike. Da bi vaščane motiviral k varčevanju z njo, je župan odredil, da bo elektrika za tiste, ki je porabijo veliko, dražja kot za tiste, ki je porabijo manj. Na koncu meseca pošljejo vsakemu vaščanu račun za elektriko, ki ga izračunajo po naslednjih pravilih:

- Recimo, da je ta vaščan porabil v tem mesecu x sodčkov elektrike[1] (x bo vedno celo število) in da ima ta mesec d dni.
- Za potrebe izračuna se predpostavi, kot da je vsak dan porabil natanko x/d sodčkov elektrike (ne glede na to, kako je bila v resnici razporejena poraba po mesecu).
- Za vsak dan se določi ceno elektrike takole: prvih 6 sodčkov tisti dan stane po 10 dinarjev vsak; naslednjih 6 sodčkov stane po 11 dinarjev vsak; naslednjih 6 sodčkov stane po 12 dinarjev vsak; naslednjih 6 sodčkov stane po 13 dinarjev vsak; vsa preostala poraba se zaračuna po 18 dinarjev na sodček.
- Ceno na dan potem pomnožimo s številom dni v mesecu (torej d) in tako dobimo znesek, ki ga bo moral ta vaščan ta mesec plačati za elektriko.

Primer: recimo, da ima mesec $d = 30$ dni in da smo porabili $x = 819$ sodčkov elektrike. Povprečna dnevna poraba je torej $x/d = 819/30 = 27{,}3$ sodčkov. Prvih 6 sodčkov na dan stane po 10 dinarjev, naslednjih 6 po 11, naslednjih 6 po 12, naslednjih 6 po 13, preostalih 3,3 sodčkov na dan (ker je $27{,}3 - 4 \cdot 6 = 3{,}3$) pa stane po 18 dinarjev. Cena na dan je torej $6 \cdot 10 + 6 \cdot 11 + 6 \cdot 12 + 6 \cdot 13 + 3{,}3 \cdot 18 = 335{,}4$ dinarjev, za ves mesec pa $335{,}4 \cdot d = 10\,062$ dinarjev.

V praksi porabijo Butalci pozimi več elektrike kot poleti in jo zato v mesecih, ko je porabijo veliko, tudi dražje plačujejo. Toda Cefizelj se je domislil, da bi lahko prihranil kar nekaj denarja, če bi porabo prerazporedil po mesecih (pri čemer bi skupna poraba ostala enaka).



Leva slika kaže primer porabe v štirimesečnem obdobju; sodčki, ki jih plačamo po višji ceni, so predstavljeni s temnejšimi odtenki sive. Širina stolpcev ponazarja dolžino meseca (v praksi seveda razlike v dolžini mesecev niso tako dramatične kot na naši sliki). Desna slika kaže, kako bi lahko isto skupno porabo razporedili med te štiri mesece tako, da bi bila skupna cena manjša (čeprav ne nujno tudi najmanjša možna).

Napiši program, ki prebere s standardnega vhoda 12 celih števil, ki predstavljajo porabo elektrike po mesecih od januarja do decembra, in

(a) izpiše skupno ceno elektrike, ki bi jo v tem letu pri takšni porabi po mesecih plačali; in

(b) izpiše, kako bi morali elektriko razporediti po mesecih, da bi bila skupna letna poraba enaka kot v vhodnih podatkih, skupna cena elektrike pa najmanjša možna. Izpiše naj tudi skupno ceno elektrike za celo leto pri tako preprazporejeni porabi. Pri tem pa upoštevaj omejitev, da mora biti mesečna poraba za vsak mesec še vedno celo število sodčkov. Če je možnih več enako dobrih rešitev, je vseeno, katero izpišeš.

Program, ki reši le podnalogo (a), dobi največ 8 točk (od 20 možnih). Pri vseh izračunih lahko predpostaviš, da ima februar 28 dni.

[1] Vsi vemo, da se električno energijo meri v kilovatnih urah, vendar je mestni starešina protestiral, češ da elektrike že ne morejo računati v urah, ker ima mesec za vse vaščane enako ur, pa jo odtlej računajo v sodčkih.

6. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2011

NALOGE ZA DRUGO SKUPINO

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

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev samodejno shranjuje in na zaslonu ti bo pisalo, kdaj se bo shranjevanje naslednjič zgodilo. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) **Zaradi varnosti priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku** (npr. kopiraj in prilepi v Notepad in shrani v datoteko).

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite (bolj učinkovite rešitve dobijo več točk). **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. Ve,jic,e

Tomija je učiteljica v šoli čisto zastrašila z vejicami: ker so mu v spisih kar naprej manjkale, jih zdaj za vsak slučaj raje napiše malo več. Da jih ja ne bi bilo premalo, postavi kakšno celo sredi besede. Primer njegovega spisa:[2]

```
Na t,em vl,a,ku stre,žejo so,uv,laki, ne pa tudi hra,ne, bogate
z vla,kni,n,ami. Vp,liv lako,,te je, hud!
```

Ko napiše tak spis v *Mikromehki Besedi* ali drugem urejevalniku besedil, je zoprno, da ne more enostavno zamenjati vseh pojavitev nekega niza w_1 z drugim (enako dolgim) nizom w_2. Če se recimo želi prikupiti svoji učiteljici z Notranjskega in vse pojavitve niza `vlak` zamenjati z `wlah`, bi moral na koncu dobiti takšno besedilo:

```
Na t,em wl,a,hu stre,žejo so,uw,lahi, ne pa tudi hra,ne, bogate
z wla,hni,n,ami. Vp,liv lako,,te je, hud!
```

Njegov urejevalnik besedila pa tega ne zna, saj zgago delajo vejice, razmetane znotraj w_1.

Pomagaj ubogemu Tomiju. **Napiši podprogram** Zamenjaj(s, w1, w2), ki izpiše besedilo, kakršno nastane, če v besedilu `s` zamenjamo vse pojavitve niza `w1` z enako dolgim nizom `w2`. Pri teh zamenjavah mora ohraniti vse originalne položaje vejic, kot je tudi prikazano v zgornjem primeru. Tvoj podprogram naj bo takšne oblike:

[2] Souvlaki je grška različica ražnjičev.

```
void Zamenjaj(char *s, char *w1, char *w2);                    /* v C/C++ */
void Zamenjaj(string s, string w1, string w2);                 // v C++
public static void zamenjaj(String s, String w1, String w2);  // v javi
procedure Zamenjaj(s, w1, w2: string);                         { v pascalu }
def Zamenjaj(s, w1, w2): ...                                   # v pythonu
```

2. (Opomba)

Pri pisanju besedila uporabimo oklepaje, če želimo zapisati kakšno opombo (ali kaj drugega manj pomembnega). Včasih tudi znotraj opombe povemo kaj še manj pomembnega (torej manj pomembnega od opombe (ki že sama ni pomembna), a še vedno zanimivega); v takem primeru lahko uporabimo gnezdene oklepaje.

Gnezdenje oklepajev lahko načeloma neomejeno stopnjujemo — možno je imeti oklepaje v oklepajih v oklepajih v oklepajih ... v oklepajih. Če se nam pri branju takšnega teksta mudi, lahko preskočimo vse dele besedila, ki so gnezdeni vsaj k korakov globoko, pri čemer je vrednost k odvisna od tega, kako hudo se nam mudi.

Napiši podprogram SamoPomembno(s, k), ki sprejme niz `s` s tovrstnim besedilom in naravno število k, nato pa na standardni izhod izpiše to, kar ostane od besedila `s`, če iz njega pobrišemo vso vsebino, ki je vgnezdena več kot k nivojev globoko. Iz besedila naj bodo pri tem pobrisani tudi prazni oklepajski pari, t.j. takšni, ki ne vsebujejo med oklepajem in zaklepajem nobenega drugega znaka, niti presledka. Pozor, oklepajski par je lahko prazen tudi zato, ker smo iz njegove notranjosti brisali druge prazne ali pregloboko gnezdene oklepajske pare.

Predpostaviš lahko, da so oklepaji v besedilu smiselno postavljeni; tako na primer niz „`foo)bar(baz)qux`“ ni veljaven vhodni podatek.

Primer: ob klicu

```
SamoPomembno("ja (ja(ja(a b)(tri)) r((c d)(ef ghi))tk() ena) nič", 2);
```

se mora izpisati:

```
ja (ja(ja) rtk ena) nič
```

Tvoj podprogram naj bo takšne oblike:

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

Če ti je ta naloga pretežka, lahko rešiš naslednjo lažjo različico: pobriši vso vsebino, zapisano v k ali več oklepajih, ne briši pa praznih oklepajskih parov. Rešitev te lažje različice dobi 10 točk (namesto 20).

3. Kozarci

Smo v proizvodnji večstranih kozarcev (imajo n stranic, pri čemer je n najmanj 4), kjer moramo nadzorovati stroj za barvanje stranic teh kozarcev. Na voljo imamo več funkcij za nadzorovanje tega stroja:

- **void** Naslednji() — vzame naslednji kozarec s tekočega traku (nekatere stranice na njem so lahko že pobarvane). Tega, kako je kozarec na začetku obrnjen, ne vemo.
- **void** Koncaj() — odloži ravnokar pobarvan kozarec na tekoči trak
- **void** Zavrzi() — odloži kozarec v zaboj s pokvarjenimi oziroma nepravilno pobarvanimi kozarci
- **void** Obrni() — obrne kozarec okoli vertikalne osi za $1/n$ obrata (torej za eno stranico naprej)
- **int** Preveri() — preveri, katera barva je na trenutno sprednji stranici; če trenutna stranica ni pobarvana, funkcija vrne 0
- **void** Pobarvaj(**int** barva) — pobarva trenutno sprednjo stranico z navedeno barvo. To funkcijo smeš poklicati le, če trenutna stranica še ni pobarvana.

Zahtevani vrstni red barv po stranicah kozarca je podan v tabeli **int** Barve[n]. (Število stranic, n, je tudi podano in je za vse kozarce enako.) **Napiši program**, ki v neskončni zanki jemlje kozarce s tekočega traku, poskrbi, da so pravilno pobarvani (pri tem pregleda že pobarvane stranice in pobarva še nepobarvane stranice), in jih odlaga na trak; če pa ugotovi, da kozarca ni mogoče pravilno pobarvati, naj ga odvrže.

4. Telefonske številke

Imamo n telefonskih številk, $s_1, \ldots, s_n$ (vse so k-mestne in v njih nastopajo številke od 0 do 9), ki jih pogosto kličemo. Katerikoli dve od njih se razlikujeta v vsaj štirih istoležnih številkah. Pri klicanju se pogosto zmotimo v eni številki (in torej pokličemo napačno številko), nikoli pa v več kot eni. Zdaj smo dobili seznam številk, ki smo jih dejansko poklicali, recimo $t_1, \ldots, t_m$ (tudi to so k-mestne številke, niso pa nujno vse različne), pri čemer je m precej večji od n. Originalnih $s_1, \ldots, s_n$ nimamo več, vemo pa, da se vsaka od njih pojavi vsaj enkrat nespremenjena v zaporedju $t_1, \ldots, t_m$. Iz zaporedja $t_1, \ldots, t_m$ torej ne moremo nujno ugotoviti, katere so originalne številke $s_1, \ldots, s_n$, lahko pa ugotovimo, katere skupine t-jev se nanašajo na isto originalno številko; **opiši postopek**, ki ugotovi velikost največje take skupine.

Primer: če imamo naslednje zaporedje 15 štirimestnih številk $t_1, \ldots, t_m$ (torej je $k = 4$ in $m = 15$):

1234, 2234, 4322, 3234, 2121, 1334, 4352, 1214, 5545,
2123, 4312, 4512, 5445, 4445, 5444, 5145, 5345

se dá ugotoviti, da so bile originalne številke štiri (torej $n = 4$) in da se na posamezne originalne številke nanašajo naslednje klicane številke:

- 1234, 2234, 3234, 1334, 1214 so vse nastale iz iste originalne številke;
- 4322, 4352, 4312, 4512 so vse nastale iz iste originalne številke;
- 5445, 5545, 5145, 5345, 4445, 5444 so vse nastale iz iste originalne številke;
- 2121, 2123 sta obe nastali iz iste originalne številke.

Za prve tri od teh štirih skupin lahko celo ugotovimo, kakšne so bile originalne številke s_i: to so bile 1234, 4322 in 5445; pri četrti skupini pa ne moremo ugotoviti, ali je bila originalna številka 2121 ali 2123. Kakorkoli že, vidimo lahko, da ima največja od teh skupin šest številk, tako da bi moral tvoj postopek pri tem primeru kot rezultat vrniti število 6.

5. Assembler

Pri tej nalogi bomo namesto običajnih programskih jezikov, kot so C/C++, pascal, java in podobni, uporabljali zelo preprost zbirni jezik (assembler) za nek izmišljen, zelo preprost procesor.

Program je zaporedje ukazov (vsi možni ukazi so opisani spodaj); pred vsakim ukazom lahko stoji tudi oznaka (labela), na katero se lahko sklicujemo pri pogojnih skokih. Razen v primeru pogojnih skokov pa se ukazi izvajajo po vrsti.

Program lahko uporablja poljubno število celoštevilskih spremenljivk (kot tip **int** oz. integer).

Poleg tega obstajata še dve logični spremenljivki, E in L, ki ju ne moremo spreminjati ali brati neposredno, pač pa z njima delajo nekatere inštrukcije (CMP nastavi njuno vrednost, JE in JL pa njuno vrednost bereta).

Poleg tega obstaja tudi sklad, ki lahko hrani poljubno dolgo zaporedje (seznam) celoštevilskih vrednosti. Do vsebine sklada ne moremo dostopati drugače kot prek ukazov PUSH (ki doda nov element na vrh sklada) in POP (ki pobere element z vrha sklada).

(*Besedilo se nadaljuje na naslednji strani.*)

Oglejmo si zdaj seznam vseh ukazov, ki jih podpira naš namišljeni procesor:

- ADD x, y — prišteje vrednost spremenljivke y spremenljivki x (rezultat shrani v x);
- SUB x, y — odšteje vrednost spremenljivke y od spremenljivke x (rezultat shrani v x);
- CMP x, y — primerja vrednosti spremenljivk x in y, ter nastavi logični vrednosti (zastavici) E in L: E dobi vrednost **true**, če je $x = y$, sicer dobi vrednost **false**; L dobi vrednost **true**, če je $x < y$, sicer dobi vrednost **false**;
- SET x, c — nastavi spremenljivko x na vrednost c (ki mora biti neka celoštevilska konstanta);
- JE lbl — nadaljuje izvajanje za oznako lbl, če ima E vrednost **true**;
- JL lbl — nadaljuje izvajanje za oznako lbl, če ima L vrednost **true**;
- PUSH x — doda vrednost spremenljivke x na vrh sklada;
- POP x — pobere vrednost z vrha sklada in jo vpiše v spremenljivko x; če je bil sklad prazen, se procesor sesuje.

V tem zbirnem jeziku **napiši zaporedje ukazov**, ki izračuna zmnožek števil v spremenljivkah x in y; ob koncu izvajanja mora biti ta zmnožek shranjen v spremenljivki x. Predpostavi, da sta na začetku izvajanja vrednosti spremenljivk x in y večji ali enaki 0 in da sta dovolj majhni, da pri množenju ne bo prišlo do težav zaradi prekoračitve obsega celih števil ali česa podobnega. Poleg spremenljivk x in y lahko uporabiš še poljubno mnogo svojih pomožnih spremenljivk, poleg tega pa lahko uporabljaš tudi sklad, vendar mora biti sklad ob koncu izvajanja tvojega zaporedja ukazov vedno v enakem stanju kot na začetku izvajanja.

Zaželeno je, da je tvoja rešitev čim bolj učinkovita, torej da ob računanju zmnožka $x \cdot y$ izvede čim manj ukazov.

Za primer si oglejmo naslednje zaporedje ukazov, ki rešuje malo drugačen problem — s sklada pobere najprej n in nato še n števil ter na koncu v spremenljivki vsota izračuna vsoto tistih n števil.

```
   POP n
   SET vsota, 0
zacetek:
   SET temp, 0
   CMP n, temp
   JE konec
   POP x
   ADD vsota, x
   SET temp, 1
   SUB n, temp
   CMP temp, temp
   JE zacetek
konec:
```

6. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2011

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

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

Delovno okolje

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

Na vsakem računalniku imaš na voljo imenik `U:\_Osebno`, v katerem lahko kreiraš svoje datoteke. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C# ali java, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz JDK 1.6 in s prevajalnikom za C# iz Visual Studia 2008. Za delo lahko uporabiš `FP` oz. `ppc386` (Free Pascal), `GCC/G++` (GNU C/C++ — command line compiler), `javac` (za javo 1.6), Visual Studio 2005/2010 in druga orodja.

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

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

```
rtk -name JNovak
```

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

Za oddajo rešitve uporabi enega od naslednjih ukazov:

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

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

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

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

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk. (Izjema je prva naloga, kjer je testnih primerov 20 in za pravilen odgovor pri posameznem testnem primeru dobiš 5 točk.) Nato se točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N - 1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

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

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

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

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

Primer vhodne datoteke:

```
123 456
```

Ustrezna izhodna datoteka:

```
5790
```

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

- V pascalu:

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

- V C-ju:

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

- V C++:

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

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

- V javi:

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

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

- V C#:

```
using System.IO;

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

6. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2011

NALOGE ZA TRETJO SKUPINO

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

1. Reklama (`reklama.in`, `reklama.out`)

Stanovanjsko naselje je razdeljeno na pravokotno mrežo gospodinjstev s h vrsticami in w stolpci; v vsaki celici mreže je eno gospodinjstvo. Oglaševalska agencija želi razširiti novico o novem izdelku. V preteklosti je bila opravljena anketa, s pomočjo katere so določili k potencialnih kupcev. Ker bi obiskovanje vsakega gospodinjstva posebej vzelo preveč časa, so se odločili izvesti predstavitev izdelka samo v enem gospodinjstvu (ne nujno v gospodinjstvu potencialnega kupca!). Vedo namreč, da se bo novica o izdelku hitro razširila po naselju. Iz gospodinjstva, ki je že prejelo novico (neposredno s predstavitvijo ali posredno preko svojih sosedov), se v enem dnevu novica razširi na vsa štiri sosednja gospodinjstva, ki imajo z njim v mreži skupno eno stranico. Širjenje reklame je neodvisno od zainteresiranosti članov gospodinjstva za nakup izdelka. **Napiši program**, ki ugotovi, v najmanj kolikšnem času lahko oglaševalci obvestijo vseh k potencialnih kupcev o novem izdelku, če izvedejo predstavitev v najbolj primernem gospodinjstvu.

Vhodna datoteka: v prvi vrstici so podana cela števila h, w in k, ločena s po enim presledkom. Sledi seznam lokacij potencialnih kupcev, ki jih je potrebno obvestiti o izdelku. Vsak kupec je opisan v svoji vrstici s številko vrstice in stolpca gospodinjstva. Veljalo bo $1 \leq h \leq 1000$, $1 \leq w \leq 1000$ in $1 \leq k \leq 1000$. Vrstice so oštevilčene od 1 do h, stolpci pa od 1 do w.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število dni, v katerem lahko novica doseže vseh k potencialnih kupcev, če izvedemo začetno predstavitev v najbolj primerno izbranem gospodinjstvu.

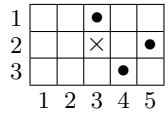
Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
2
```

Slika tega primera:



Pike • označujejo potencialne kupce. Če izvedemo predstavitev v drugi vrstici in tretjem stolpcu (kjer je znak ×), bo novica dosegla vse potencialne kupce v dveh dneh (tistega v vrstici 1 bo dosegla že prvi dan, ostala dva pa drugi dan).

2. Kompresija slike (`slika.in`, `slika.out`)

Da bi prihranili na količini prostora, potrebnega za zapis sivinske slike, želimo zmanjšati število sivinskih nivojev na največ k odtenkov. Izbrati je torej treba največ k odtenkov in vsako točko na sliki pobarvati z enim izmed k izbranih odtenkov. Vsi sivinski odtenki morajo biti cela števila med vključno 0 in 1000. Napako tako zapisane slike definiramo kot vsoto absolutnih vrednosti razlik med vrednostmi slikovnih točk na začetni in končni sliki. **Napiši program**, ki ugotovi, kolikšna je najmanjša napaka, ki jo lahko dosežemo pri taki kompresiji slike.

Vhodna datoteka: v prvi vrstici vhodnih podatkov so podana cela števila h, w in k, ločena s po enim presledkom. Sledi opis sivinske slike v obliki tabele s h vrsticami in w stolpci. Vrednosti slikovnih elementov so ločene s presledki in so po velikosti med vključno 0 in 1000. Veljalo bo $1 \leq w \leq 100$, $1 \leq h \leq 100$ in $1 \leq k \leq 20$.

Pri 60 % testnih primerov so vrednosti vseh slikovnih elementov med vključno 0 in 50.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer minimalno napako pri kompresiji slike na k sivinskih odtenkov.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
3
```

Možen končni videz te slike po kompresiji bi bil takšen (tega ne piši v izhodno datoteko — tu je podan le kot ilustracija):

```
1 1 1 1
3 3 1 1
```

3. Konstrukcija grafa (`graf.in`, `graf.out`)

Urbanist se ukvarja s posebno idejo ureditve glavnega mesta. Raziskave so pokazale, da večina turistov prispe v mesto preko letališča in nadaljuje svojo pot z železniške postaje na drugem koncu mesta. V času med prihodom letala in odhodom vlaka pa si krajšajo čas z ogledom nekaterih izmed 30 mestnih znamenitosti. Letališče in železniška postaja sta izjemna arhitekturna dosežka in se uvrščata v omenjenih 30 znamenitosti. Cilj urbanista je narediti mesto čim privlačnejše za turiste. V ta namen bo izdelal načrt enosmernih ulic, ki bodo povezovale znamenitosti. Zanimivost in prepoznavnost mesta pa želi poudariti s tem, da bo možno priti z letališča do železniške postaje na točno n različnih načinov (pri tem so dovoljene tudi take poti, ki obiščejo kakšno znamenitost po večkrat; niso pa dovoljene poti, ki bi šle po kakšni ulici v napačno smer). Župan mu je prepovedal uporabo vzporednih ulic, torej lahko neposredno od znamenitosti A do znamenitosti B vodi največ ena ulica. **Napiši program**, ki prebere n in izpiše načrt ulic, ki ustreza tem zahtevam.

Vhodna datoteka: vsebuje eno samo celo število, n, ki predstavlja želeno število različnih poti od letališča do železniške postaje. Veljalo bo $n \leq 10^8$, v 70 % testnih primerov pa bo veljalo tudi $n \leq 5000$.

Izhodna datoteka: za vsako enosmerno ulico izpiši po eno vrstico, v njej pa naj bosta števili A in B (ločeni z enim presledkom), ki povesta, da ulica pelje od znamenitosti A do znamenitosti B. Znamenitosti so oštevilčene s števili od 1 do 30. Letališče je označeno s številom 1, železniška postaja pa z 2.

Če obstaja več različnih pravilnih rešitev, je vseeno, katero od njih izpišeš.

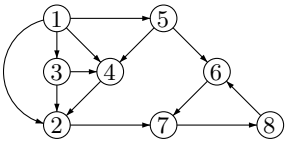
Primer vhodne datoteke:

```
5
```

Ena od možnih pripadajočih izhodnih datotek:

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

Slika tako dobljenega omrežja ulic:



4. Mušji drekci (`musji.in`, `musji.out`)

V predalu sameva kuverta; nanjo se sčasoma poserje n mušic. Končno jo iz predala potegne tajnica, ki se ji grozno mudi oddati pošto. Kuverta je edina, ki jo ima, po drugi strani pa mora z njo odposlati strašno ugleden dopis, zato s kuverto nikakor ne želi razkazovati prebave lokalnih mušic. Edina možna rešitev je, da neomiko prikrije z dvema znamkama.

Napiši program, ki prebere opis drekcev na kuverti ter velikost znamk in pove, ali je možno z dvema znamkama prekriti vso mušjo nesnago. Drekce obravnavamo kot točke. Znamki sta enako veliki, imata kvadratno obliko ter ravne robove in morata biti nalepljeni vzporedno s stranicami kuverte. Znamki se lahko prekrivata in lahko gledata čez rob kuverte. Drekec, ki leži točno pod robom znamke, šteje kot pokrit.

Vhodna datoteka: vhodni podatki so sestavljeni iz več testnih primerov. Na vhodu bo zato v prvi vrstici napisano število testnih primerov c (to je celo število in zanj velja $1 \leq c \leq 10$). Sledi c blokov podatkov; njihovo strukturo opisuje naslednji odstavek.

Vsak posamezen blok se začne s prazno vrstico. Sledi ji vrstica z n ($0 \leq n \leq 10^5$), številom drekcev. V naslednji vrstici se nahaja celo število a, dolžina stranice znamke v mikrometrih. Vsaka od naslednjih n vrstic vsebuje dve s presledkom ločeni celi števili, x_i in y_i; to sta koordinati i-tega drekca, prav tako v mikrometrih. Vsa števila a, x_i in y_i so večja ali enaka 1 in manjša ali enaka 10^9.

V 50 % testnih primerov bo dodatno veljalo $n \leq 5000$; v 20 % testnih primerov bo veljalo $n \leq 300$.

Izhodna datoteka: za vsakega od testnih primerov izpiši v samostojni vrstici `DA`, če je drekce mogoče prekriti z znamkama na opisani način, sicer `NE`.

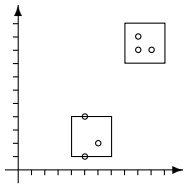
Primer vhodne datoteke:

```
2

6
3
5 1
5 4
6 2
9 10
9 9
10 9

3
10
1 1
100 1
200 1
```

Pripadajoča izhodna datoteka:

```
DA
NE
```

Komentar: v prvem primeru lahko z eno znamko pokrijemo prve tri naštete drekce in z drugo preostale tri, na primer takole:



5. Podajanje žoge (`podaje.in`, `podaje.out`)

Otroci, razdeljeni v dve ekipi, se postavijo na križišča kariraste mreže. Med seboj si podajajo žogo po naslednjih pravilih: en otrok lahko poda žogo drugemu le, če oba pripadata isti ekipi in stojita v isti vrstici ali stolpcu in med njima ni nobenega drugega otroka. **Napiši program**, ki prebere koordinate vseh otrok in izračuna najmanjše potrebno število podaj, v katerem lahko pride žoga od določenega otroka do določenega drugega otroka.

Vhodna datoteka: v prvi vrstici je število otrok, n. Sledi n vrstic, ki po vrsti opisujejo posamezne otroke. Vsaka od teh vrstic vsebuje tri cela števila, ločena s po enim presledkom: najprej x-koordinato tega otroka, nato y-koordinato tega otroka in končno še številko ekipe, ki ji ta otrok pripada (1 ali 2). Nikoli se ne zgodi, da bi dva ali več otrok stalo na isti točki.

Na koncu pride še vrstica z dvema različnima celima številoma, s in t, ločenima z enim presledkom. To sta številki otrok (velja torej $1 \leq s \leq n$ in $1 \leq t \leq n$) in sta pri vseh testnih primerih izbrani tako, da otroka pripadata isti ekipi.

Veljalo bo $1 \leq n \leq 100\,000$, vse koordinate otrok pa so večje ali enake 0 in manjše ali enake 10^6. Pri 50 % testnih primerov bo $n \leq 1000$, koordinate otrok pa bodo manjše ali enake 1000.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število podaj, v katerih je mogoče žogo spraviti od otroka s do otroka t (ob upoštevanju prej navedenih pravil podajanja). Če žoga od s do t sploh ne more priti, pa izpiši `-1`.

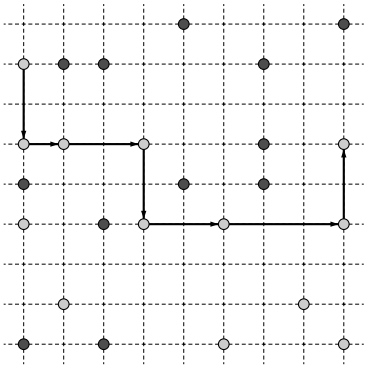
Primer vhodne datoteke:

```
25
0 0 1
2 0 1
5 3 2
8 3 2
0 5 2
4 4 1
6 4 1
6 5 1
1 7 1
2 7 1
8 0 2
1 1 2
2 3 1
0 4 1
0 7 2
0 3 2
3 3 2
7 1 2
6 7 1
4 8 1
8 8 1
5 0 2
8 5 2
1 5 2
3 5 2
15 23
```

Pripadajoča izhodna datoteka:

```
7
```

Slika tega primera:



6. tekmovanje ACM v znanju računalništva za srednješolce

26. marca 2011

REŠITVE NALOG ZA PRVO SKUPINO

1. Vsota

Premikajmo se po zaporedju z dvema indeksoma: i kaže na prvi člen našega podzaporedja, j pa na zadnji člen podzaporedja. Na začetku postavimo i na 1 in se z j zapeljimo tako daleč, da bo vsota podzaporedja presegla m. Nato počasi povečujmo i. Vsakič, ko i povečamo za 1, izpade en člen iz vsote; če je ta zdaj manjša ali enaka m, moramo dodati v podzaporedje nekaj novih členov, torej povečajmo j (in popravimo vsoto; to ponavljajmo, dokler vsota ne preseže m).

```
void Vsota(int a[], int n, int m)
{
  int z, d = n + 1, i = 0, j = -1, vsota = 0;
  while (i < n) {
    /* Dodajajmo člene v podzaporedje (in povečujmo j),
       dokler njegova vsota ne preseže m. */
    while (j + 1 < n && vsota <= m)
      vsota += a[++j];
    /* Če smo našli najkrajše zaporedje doslej, si ga zapomnimo. */
    if (vsota > m && j - i + 1 < d)
      z = i, d = j - i + 1;
    /* Premaknimo levi rob zaporedja za en člen naprej. */
    vsota -= a[i++]; }
  printf("%d %d\n", z, d);
}
```

2. Lego kocke

Spodnja rešitev hrani zadnje tri izmerjene teže v spremenljivkah t1, t2 in t3. Ker naloga pravi, da teža najprej nekaj časa le narašča, nato pa nekaj časa le pada, bomo maksimalno težo prepoznali po tem, da je t1 < t2 in t2 > t3. (Če bi se lahko zgodilo, da bi bila teža pri več zaporednih meritvah enaka, bi se to preverjanje malo zapletlo.) Takrat lahko pogledamo črtno kodo vrečke in preverimo, če njena teža leži na predpisanem območju.

```
#include <stdbool.h>
int main()
{
  int t1 = 0, t2 = 0, t3 = 0, koda;
  while (true)
  {
    t1 = t2; t2 = t3; t3 = Tehtaj();
    if (t1 < t2 && t2 > t3)
    {
      /* Našli smo lokalni maksimum teže. Preberimo črtno kodo
         vrečke in preverimo, če teža ustreza omejitvam. */
      koda = CrtnaKoda();
      if (t2 < MinTeza[koda] || t2 > MaxTeza[koda])
        OdstraniVrecko();
    }
  }
}
```

3. Majevska števila

Spodnja rešitev bere vhodne podatke znak po znak, vrednost trenutne števke računa v spremenljivki stevka, vrednost vseh predhodnih števk pa v n. Če je trenutno prebrani znak še del števke (torej pika, dvopičje, črta ali 0), prištejemo njegovo vrednost k trenutni števki; ko pa pridemo do konca števke (presledek, konec vrstice ali konec datoteke), pomnožimo dosedanji n z 20, mu prištejemo novo števko in postavimo spremenljivko stevka na 0, da bo pripravljena za naslednjo števko.

```
#include <stdio.h>
int main()
{
  int n = 0, stevka = 0, c;
  do {
    c = fgetc(stdin);
    if (c == '.') stevka += 1;
    else if (c == '0') stevka = 0;
    else if (c == ':') stevka += 2;
    else if (c == '|') stevka += 5;
    else n = 20 * n + stevka, stevka = 0;
  } while (c != EOF && c != '\n');
  printf("%d\n", n);
}
```

Slabost te rešitve je, da ne deluje pravilno, če sta dve zaporedni števki ločeni z več kot enim presledkom (ali pa če so odvečni presledki na koncu vrstice ipd.); obnašala bi se, kot da bi bila med dvema sosednjima presledkoma števka z vrednostjo 0. Temu se lahko izognemo z dodatno spremenljivko, ki pove, ali se je nova števka sploh že začela:

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
  int n = 0, stevka = 0, c; bool jeStevka = false;
  do {
    c = fgetc(stdin);
    if (c == '.') stevka += 1, jeStevka = true;
    else if (c == '0') stevka = 0, jeStevka = true;
    else if (c == ':') stevka += 2, jeStevka = true;
    else if (c == '|') stevka += 5, jeStevka = true;
    else {
      if (jeStevka) n = 20 * n + stevka;
      stevka = 0; jeStevka = false; }
  } while (c != EOF && c != '\n');
  printf("%d\n", n);
}
```

4. Kemijske formule

Naloga pravi, da je vsak element predstavljen z eno veliko tiskano črko angleške abecede, tako da je možnih kvečjemu 26 različnih elementov. Imejmo torej tabelo koliko s 26 celicami, ki za vsak element pove število doslej videnih atomov tega elementa. Na začetku postavimo vse vrednosti v tabeli na 0. Niz s pregledujmo od leve proti desni; prvi znak je črka, ki predstavlja element, nato pa pride nič ali več števk, ki skupaj tvorijo število atomov tega elementa. To število počasi računamo v spremenljivki n, na koncu (ko pregledamo že vse števke in pridemo do znaka, ki ni števka) pa za n povečamo ustrezno celico tabele koliko. Posebej moramo paziti na primer, ko za oznako elementa ni nobene števke, kajti to pomeni en atom tega elementa in ne 0. Na koncu se moramo le še sprehoditi po tabeli koliko in izpisati rezultate.

```
#include <stdio.h>
void IzpisiAtome(char *s)
```

```
{
  int koliko[26], element = -1, n;
  /* Inicializirajmo tabelo. */
  for (element = 0; element < 26; element++) koliko[element] = 0;
  while (*s)
  {
    /* Prvi naslednji znak mora biti črka, ki predstavlja nek element. */
    element = *s++ - 'A'; n = 0;
    /* Preberimo število, ki ga tvorijo števke za simbolom elementa. */
    while ('0' <= *s && *s <= '9') n = 10 * n + (*s++ - '0');
    /* Če števk sploh ni bilo (in je n = 0), imamo 1 atom tega elementa.
       To bi sicer dalo napačne rezultate pri formuli C0, ampak saj taka
       formula tako ali tako nima smisla. */
    if (n == 0) n = 1;
    /* Zabeležimo teh n atomov v tabeli koliko. */
    koliko[element] += n;
  }
  /* Izpišimo rezultate. */
  for (element = 0; element < 26; element++)
    if (koliko[element]) printf("%c %d\n", 'A' + element, koliko[element]);
  printf("\n");
}
```

5. Okoljevarstveni ukrepi

Najprej si pripravimo podprogram Cena, ki izračuna ceno elektrike za en mesec v odvisnosti od porabe in števila dni v mesecu. Če ima mesec d dni, nas bo prvih $6 \cdot d$ sodčkov stalo po 10 dinarjev, naslednjih $6 \cdot d$ sodčkov po 11 dinarjev, naslednjih $6 \cdot d$ sodčkov po 12 dinarjev, naslednjih $6 \cdot d$ sodčkov po 13 dinarjev in vsi preostali sodčki po 18 dinarjev. V spremenljivki cena se nam nabira skupna cena doslej obdelanih sodčkov, spremenljivko poraba pa počasi zmanjšujemo za število že obdelanih sodčkov.

```
int Cena(int poraba, int dni)
{
  int c, p, cena = 0;
  for (c = 10; c <= 13; c++) {
    p = 6 * dni; if (d > poraba) p = poraba;
    cena += p * c; poraba -= p; }
  cena += 18 * poraba; return cena;
}
```

Glavni blok programa najprej prebere porabo po mesecih, izračuna ceno po mesecih (tu kliče podprogram Cena) ter skupno ceno in porabo za celo leto. S tem smo že rešili podnalogo (a).

Malo več dela je s podnalogo (b). Elektrika bo najcenejša, če bo povprečna dnevna poraba po vseh mesecih približno enaka (če si predstavljamo porabo ilustrirano s stolpci, tako kot v besedilu naloge, je najugodnejši razpored porabe po mesecih takrat, ko so vsi stolpci enako visoki). Povprečno dnevno porabo za celo leto dobimo tako, da skupno letno porabo delimo s številom dni v letu; toda kaj če se deljenje ne izide? Če imamo na primer 10 000 sodčkov letne porabe in 365 dni v letu, je povprečje približno 27,4. Če vsak dan porabimo 27 sodčkov, bo to naneslo v celem letu 9 855 sodčkov; do 10000 nam jih manjka še 145. Primeren razpored je torej ta, da 145 dni v letu porabimo en sodček več, torej 28 sodčkov, ostale dni pa po 27 sodčkov. Načeloma je vseeno, kako te dneve z višjo porabo razporedimo po letu; spodnja rešitev jih stlači vse v prve mesece leta.

```
#include <stdio.h>
int main()
{
  const int DniVMesecu[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  int cena = 0, poraba = 0, mesec, povp, p, dni = 0;
```

```
  /* Preberimo porabo po mesecih; za vsak mesec izračunajmo ceno;
     v spremenljivkah cena in poraba hranimo skupno ceno in porabo. */
  for (mesec = 0; mesec < 12; mesec++) {
    scanf("%d", &p);
    poraba += p;
    cena += Cena(p, DniVMesecu[mesec]);
    dni += DniVMesecu[mesec]; }
  printf("Prvotna cena: %d\n", cena);

  printf("Poraba po mesecih, ki da najnižjo ceno:\n");
  /* Izračunajmo povprečno dnevno porabo čez celo leto (zaokroženo navzdol). */
  povp = poraba / dni;
  poraba -= povp * dni;
  /* Zdaj moramo v prvih 'poraba' dneh porabiti po povp + 1 sodčkov, v ostalih
     dneh pa po povp sodčkov. */
  for (mesec = 0, cena = 0; mesec < 12; mesec++) {
    p = DniVMesecu[mesec]; if (p > poraba) p = poraba;
    /* V tem mesecu bomo imeli p dni z večjo porabo (povp + 1),
       ostale dni pa bomo porabili po povp sodčkov. */
    poraba -= p;
    /* Izračunajmo zdaj skupno porabo (in ceno) za ta mesec in jo izpišimo. */
    p += DniVMesecu[mesec] * povp;
    printf("%d\n", p); cena += Cena(p, DniVMesecu[mesec]); }
  printf("Cena pri tej porabi: %d\n", cena); return 0;
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Ve,jic,e

S kazalcem p se premikajmo po nizu s in pri vsakem položaju p-ja preverimo, ali se na tem mestu v s začne pojavitev niza w1. V ta namen se z dvema kazalcema hkrati premikajmo po nizih — s pp gremo po s od položaja p naprej, z ww pa gremo po w1 od začetka naprej. Istoležne znake primerjamo in se ustavimo, ko bodisi opazimo neujemanje bodisi pridemo do konca niza w1. Razlika v primerjavi z običajnim primerjanjem nizov je le ta, da se po s ne premaknemo vedno le za en znak naprej, ampak še preskočimo vse vejice. Če smo na ta način prišli do konca niza w1, ne da bi opazili kakšno neujemanje, potem vemo, da smo našli pojavitev w1 v s. To pojavitev moramo zdaj zamenjati z w2, torej le skopiramo znake iz w2 v s (spet od položaja p) naprej, pri tem pa v s-ju preskočimo vejice, tako da bodo ostale tam, kjer so bile.

```
#include <stdio.h>
void Zamenjaj(char *s, char *w1, char *w2)
{
  char *p, *pp, *ww;
  for (p = s; *p; )
  {
    for (pp = p, ww = w1; *ww && *ww == *pp; ww++)
      do { pp++; } while (*pp == ',');
    if (*ww) { p++; continue; }
    for (ww = w2; *ww; p++)
      if (*p != ',') *p = *ww++;
  }
  printf("%s\n", s);
}
```

2. (Opomba)

Za začetek si oglejmo rešitev lažje različice naloge, pri kateri moramo zgolj pobrisati tiste dele niza, ki so vgnezdeni vsaj k nivojev globoko. Niz s glejmo od leve proti desni in si

v spremenljivki globina zapomnimo trenutno globino gnezdenja oklepajev (torej koliko oklepajev je trenutno odprtih); ko naletimo na znak (, globino povečamo za 1, ko pa naletimo na znak), jo zmanjšamo za 1. Znake na globini, manjši od k, izpišemo. Paziti moramo le na to, da globino povečamo še pred izpisom znaka (, zmanjšamo pa jo po izpisu znaka), tako da se pri k-tem vgnezdenem oklepajskem paru ne bo izpisal niti oklepaj niti zaklepaj.

```
void SamoPomembno(char *s, int k)
{
  int globina = 0;
  for ( ; *s; s++)
  {
    if (*s == '(') globina++;
    if (globina <= k) fputc(*s, stdout);
    if (*s == ')') globina--;
  }
}
```

Če hočemo zdaj v to rešitev dodati še brisanje praznih oklepajev, se spremeni predvsem to, da ko naletimo na nek znak (, še ni takoj jasno, ali ga bo sploh treba izpisati ali ne (torej ali bo ta oklepaj na koncu zaradi brisanja praznih oklepajskih parov izpadel iz niza). Zato poleg spremenljivke globina vpeljimo še eno s spremenljivko, globIzpisa, ki pove, do katere globine smo oklepaje že izpisali. Tisti na globinah od globIzpisa + 1 do globina pa še čakajo na to, da ugotovimo, ali jih bo treba izpisati ali pa bodo mogoče izpadli zaradi brisanja oklepajskih parov. Ko naletimo na znak, ki ni niti oklepaj niti zaklepaj in ki je na globini manj kot k, vemo, da ga bo treba izpisati, in ob tej priliki najprej izpišemo še vse doslej neizpisane oklepaje (za globine od globIzpisa + 1 do globina. Zaklepaj izpišemo le, če je bil izpisan njegov pripadajoči oklepaj (torej če je globIzpisa == globina); če ni bil, je to znak, da gre za prazen oklepajski par in torej tudi zaklepaja ne smemo izpisati.

```
void SamoPomembno2(char *s, int k)
{
  int globina = 0, globIzpisa = 0;
  for ( ; *s; s++)
  {
    if (*s == '(') globina++;
    else if (*s == ')') {
      if (globIzpisa == globina) {
        /* Pripadajoči oklepaj smo očitno izpisali, torej izpišimo tudi zaklepaj. */
        fputc(*s, stdout); globIzpisa--; }
      globina--; }
    else {
      if (globina > k) continue;
      while (globIzpisa < globina) /* Najprej izpišimo še neizpisane odprte oklepaje. */
        fputc('(', stdout), globIzpisa++;
      fputc(*s, stdout); }
  }
}
```

3. Kozarci

Naloga pravi, da ne vemo, kako je kozarec zasukan, ko ga poberemo s tekočega traku. Torej ne vemo, ali bi bilo treba prvo stranico pobarvati z barvo Barve[0] ali z Barve[1] ali Barve[2] itd. Če bi bil kozarec še popolnoma nepobarvan, bi bilo to tako ali tako vseeno; težava pa je, da so nekatere stranice mogoče že pobarvane. Ugotoviti moramo torej, za koliko stranic bi bilo treba kozarec zasukati, da bi se ga dalo od tam naprej barvati z barvami od Barve[0] naprej. Zasuk za j stranic pride v poštev, če je prva stranica kozarca ravno barve Barve[j] (ali pa nepobarvana), druga stranica barve Barve[j + 1] (ali pa nepobarvana) in tako naprej. Možni zasuki so od 0 do $n - 1$ in da ne bomo za vsakega od njih obračali kozarca za poln krog, lahko vse zasuke preverjamo istočano: v tabeli kand vodimo podatke o tem, kateri zasuki sploh še pridejo v poštev (torej: kateri

so konsistentni s tem, kar smo doslej videli o trenutnem kozarcu); ko kozarec zasukamo za eno stranico naprej, pogledamo, če je ta stranica pobarvana; če je pobarvana, se sprehodimo po tabeli kand in če bi kateri od zasukov v njej zahteval za trenutno stranico neko drugo barvo namesto te, ki smo jo na kozarcu dejansko opazili, potem vemo, da tisti zasuk ni pravi. Na koncu vzamemo poljubnega od zasukov, ki je prestal to testiranje, in kozarec pobarvamo v skladu z njim; če pa ni primernega zasuka, moramo kozarec zavreči.

```
#include <stdbool.h>
int main()
{
  bool kand[n]; int i, j, barva;
  while (true)
  {
    Naslednji();
    for (i = 0; i < n; i++) kand[i] = true;
    for (i = 0; i < n; i++, Obrni()) {
      barva = Preveri();
      if (barva == 0) continue;
      /* Trenutna, i-ta stranica kozarca je pobarvana z barvo 'barva'.
         Poglejmo, če je kateri od možnih zasukov nekonsistenten s tem. */
      for (j = 0; j < n; j++)
        if (kand[j] && Barve[(i + j) % n] != barva) kand[j] = false; }
    /* Poiščimo prvi primerni zasuk; če ni nobenega, kozarec zavržemo. */
    for (i = 0; i < n && ! kand[i]; i++) ;
    if (i >= n) { Zavrzi(); continue; }
    /* Kozarec pobarvamo v skladu z izbranim zasukom i. */
    for (j = 0; j < n; j++, Obrni())
      if (Preveri() == 0) /* Že pobarvane stranice pustimo pri miru. */
        Pobarvaj(Barve[(i + j) % n]);
    Koncaj();
  }
}
```

4. Telefonske številke

Nalogo lahko elegantno rešimo tako, da za vsako t_j pogledamo, koliko je v zaporedju $t_1, \ldots, t_m$ takih številk, ki se od t_j razlikujejo v eni števki, v vseh ostalih pa se ujemajo z njo. Recimo, da je $n(j, r)$ takih številk, ki se od t_j razlikujejo le v r-ti števki, v ostalih pa se ujemajo z njo.

Če je t_j ravno ena od originalnih številk s_i (naloga pravi, da se tudi vse t_i pojavljajo vsaj enkrat v $t_1, \ldots, t_m$), bomo na ta način prešteli vse številke, ki smo jih dobili, ko smo poskušali klicati s_i (pa smo se mogoče zmotili v eni števki). Vsota $1 + \sum_{r=1}^{k} n(j, r)$ je tedaj ravno velikost cele skupine (1 smo prišteli zato, ker je tudi t_j oz. s_i sama del te skupine).

Če pa t_j ni enaka nobeni od originalnih številk, je morala nastati tako, da smo poskušali klicati neko s_i in smo se pri tem zmotili v eni števki, recimo na r-tem mestu. Potem sledi, da je $n(j, r') = 0$ za vsak $r' \neq r$. Zakaj? Če bi obstajal neka $t_{j'}$, ki bi se od t_j razlikovala le na r'-tem mestu, bi se ta $t_{j'}$ razlikovala od s_i že na dveh mestih (r in r'), torej ni mogla nastati pri poskusu klica s_i; torej je nastala pri poskusu klica neke druge originalne številke, recimo $s_{i'}$; toda ker se $s_{i'}$ (kot pravi naloga) razlikuje od s_i v vsaj štirih mestih, se $t_{j'}$ razlikuje od $s_{i'}$ v vsaj dveh mestih, torej vendarle ni mogel nastati pri poskusu klica številke $s_{i'}$. Tako smo prišli v protislovje, torej vidimo, da je res $n(j, r') = 0$ za vse $r' \neq r$. Če torej za takšno t_j izračunamo vsoto $1 + \sum_{r=1}^{k} n(j, r)$, smo s tem prešteli le tiste številke, ki so nastale ob poskusu klica s_i in se od nje razlikujejo na r-tem mestu (ali pa še tam ne); prešteli smo torej nek manjši del skupine številk, ki je nastala ob poskusih klica številke s_i.

Če torej izračunamo vsoto $1 + \sum_{r=1}^{k} n(j, r)$ za vse j in vzamemo največjo izmed tako dobljenih vsot, bomo dobili ravno velikost največje skupine številk, ki so nastale iz kakšne s_i, ravno to pa je rezultat, po katerem sprašuje naloga.

Razmislimo še malo podrobneje o tem, kako za vsako t_j prešteti, koliko drugih številk v zaporedju $t_1, \ldots, t_m$ se od nje razlikuje na r-tem mestu in se z njo ujema povsod drugod. To lahko elegantno naredimo tako, da vse številke $t_1, \ldots, t_m$ leksikografsko uredimo, le da pri primerjanju številk preskočimo r-to številko. Tako bodo prišle skupaj številke, ki se ujemajo povsod razen na r-tem mestu. Še ena možnost je, da številke $t_1, \ldots, t_m$ (brez r-te številke) zložimo v razpršeno tabelo (*hash table*) ali pa v drevo (*trie*), pri čemer bodo spet prišle skupaj tiste, ki se ujemajo povsod razen na r-tem mestu.

Zapišimo postopek še s psevdokodo:

(* v v_j *se bo nabirala vsota* $1 + \sum_{r=1}^{k} n(j,r)$ *)*
for $j := 1$ **to** m **do** $v_j := 1$;

for $r := 1$ **to** k:
  pripravi skupine številk (iz $t_1, \ldots, t_m$), ki se ujemajo
    povsod razen mogoče v r-ti številki (kot smo videli zgoraj, lahko
    to naredimo z urejanjem, razpršeno tabelo, drevesom ipd.);
  za vsako tako skupino:
    naj bo N število številk v tej skupini;
    za vsako t_j iz te skupine:
      (* *vemo, da je* $n(j,r) = N - 1$ *)*
      $v_j := v_j + N - 1$;

vrni $\max\{v_1, v_2, \ldots, v_m\}$;

5. Assembler

Preprosta, a neučinkovita rešitev je, da zmnožek $x \cdot y$ računamo kot $x + x + \ldots + x + x$, torej vsoto y členov z vrednostjo x. Spodnji program to vsoto računa v spremenljivki z, ki jo v vsaki iteraciji zanke poveča za x; števec zanke pa je spremenljivka yy, ki se začne pri y in se v vsaki iteraciji zmanjša za 1, dokler ne pade na 0. Na koncu imamo torej v z ravno vrednost $x \cdot y$ in jo moramo le še skopirati v x (saj naloga pravi, da mora biti zmnožek na koncu v x).

```
  SET yy, 0
  ADD yy, y          /* yy = y */
  SET z, 0           /* z = 0 */
zacetek:
  SET temp, 0
  CMP yy, temp       /* ali je yy == 0? */
  JE konec           /* če da, končajmo */
  ADD z, x           /* z = z + x */
  SET temp, 1
  SUB yy, temp       /* yy = yy - 1 */
  CMP temp, temp
  JE zacetek         /* skočimo nazaj na začetek zanke */
konec:
  SET x, 0
  ADD x, z           /* x = z */
```

Učinkovitejša ideja je tale: y lahko vsekakor zapišemo v dvojiškem zapisu, torej kot vsoto potenc števila 2, na primer

$$1234 = 1024 + 128 + 64 + 16 + 2 = 2^{10} + 2^7 + 2^6 + 2^4 + 2^1.$$

Iz tega sledi tudi

$$x \cdot 1234 = 2^{10}x + 2^7x + 2^6x + 2^4x + 2^1x.$$

Torej, če bi znali y nekako razbiti na potence števila 2 (z drugimi besedami, če bi znali preverjati, kateri biti v dvojiškem zapisu števila y so prižgani) in izračunati še x-kratnike teh potenc, bi morali potem le še sešteti te x-kratnike. Težava te ideje je, da zbirni jezik naše naloge ne ponuja tradicionalnih operacij za delo z biti (SHL, AND in podobne). Pomagamo pa si lahko z naslednjim dejstvom: recimo, da leži y na območju

$2^k \leq y < 2^{k+1}$. Potem vemo, da je bit k v dvojiškem zapisu y gotovo prižgan; vsi nižji biti skupaj imajo namreč vrednost le $2^{k-1} + \ldots + 4 + 2 + 1 = 2^k - 1$, kar je manj od y, torej bomo do y lahko prišli le, če je bit k prižgan. Ker torej vemo, da je bit k prižgan, lahko k našemu zmnožku (ki ga bomo spet računali v spremenljivki z) prištejemo $2^k x$, nato pa bit k v y ugasnemo preprosto tako, da od y odštejemo 2^k. Zdaj imamo pred sabo neko manjše število, ki ga obdelujemo naprej na enak način z manjšimi potencami števila 2.

Z operacijami, ki jih imamo na voljo, ni težko računati potenc števila 2 v naraščajočem vrstnem redu: $1 + 1 = 2$, $2 + 2 = 4$, $4 + 4 = 8$, $8 + 8 = 16$ in tako naprej. Enako velja tudi za x-kratnike teh potenc. Toda postopek, ki smo ga pravkar zasnovali, bi želel pregledovati potence v padajočem vrstnem redu. Ker nimamo pri roki inštrukcije za deljenje z 2 ali kaj podobnega, si pomagamo s tem, da števila 2^k in $2^k x$ sproti, ko jih računamo, odlagamo na sklad. Tako bodo nižje potence pri dnu sklada, višje pa pri vrhu večje potence pri vrhu sklada in ko jih bomo kasneje pobirali s sklada, jih bomo dobivali v padajočem vrstnem redu, torej točno tako, kot jih potrebujemo.

Oglejmo si zdaj najprej zanko, ki izračuna dovolj potenc števila 2 (in njihovih x-kratnikov) ter jih odloži na sklad:

```
   SET p, 1          /* p = 1 */
   SET px, 0
   ADD px, x         /* px = x */
zacetek1:
   PUSH p            /* Zdaj je p neka potenca števila 2 in px = p * x. */
   PUSH px           /* Odložimo ju na sklad. */
   CMP y, p
   JL konec1         /* Če je p > y, končajmo. */
   ADD p, p          /* Podvojimo p in px. */
   ADD px, px
   CMP y, y
   JE zacetek1
konec1:
```

V resnici bi se lahko ustavili že, čim je $p \geq y$, ne šele pri $p > y$; vendar je pogoj $p > y$ malo lažje preverjati. Kakorkoli že, zdaj pride na vrsto glavna zanka, ki ugaša bite v y in računa zmnožek $x \cdot y$:

```
   SET z, 0          /* z = 0 */
   SET yy, 0
   ADD yy, y         /* yy = y */
zacetek2:
   POP px            /* Poberimo s sklada naslednjo potenco */
   POP p             /* števila 2 in njen x-kratnik. */
   CMP yy, p
   JL preskok        /* Če je yy < p, ta bit ni prižgan. */
   SUB yy, p         /* Sicer ugasnimo bit p v yy. */
   ADD z, px         /* z = z + p * x */
preskok:
   SET tmp, 1
   CMP p, tmp
   JG zacetek2       /* Če je p > 1, moramo še nadaljevati. */
   SET x, 0          /* Zdaj je z = x * y; skopirajmo */
   ADD x, z          /* ta zmnožek v x. */
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Reklama

Označimo položaj, na katerem smo izvedli predstavitev, z (x_0, y_0). Iz besedila naloge sledi, da po t dneh izvejo za naš izdelek vsa gospodinjstva (x, y), katerih koordinate ustrezajo pogoju

$$|x - x_0| + |y - y_0| \leq t.$$

Upoštevajmo, da je $|a| = \max\{a, -a\}$; gornji pogoj je zato enakovreden naslednjemu:

$$\max\{x - x_0, x_0 - x\} + \max\{y - y_0, y_0 - y\} \le t,$$

tega pa lahko razbijemo na štiri dele, odvisno od predznaka števil $x - x_0$ in $y - y_0$:

$$\begin{aligned} x - x_0 + y - y_0 &\le t \\ x - x_0 - y + y_0 &\le t \\ x_0 - x + y - y_0 &\le t \\ x_0 - x + y_0 - y &\le t \end{aligned}$$

Vidimo, da tu pogosto nastopajo vsote in razlike x- in y-koordinat; da si bomo te reči lažje predstavljali, pišimo $u = x + y$, $v = x - y$ in podobno za u_0 in v_0. Tako dobimo:

$$\begin{aligned} u - u_0 &\le t \\ v - v_0 &\le t \\ v_0 - v &\le t \\ u_0 - u &\le t, \end{aligned}$$

kar je isto kot

$$|u - u_0| \le t \quad \text{in} \quad |v - v_0| \le t.$$

Mi bi seveda radi izbrali u_0, v_0 in t tako, da bodo ti pogoji izpolnjeni za vsa gospodinjstva naših potencialnih kupcev (in da bo t pri tem čim manjši). Naj bo (x_i, y_i) položaj i-tega izmed teh gospodinjstev in naj bo $u_i = x_i + y_i$, $v_i = x_i - y_i$. Ko si enkrat izberemo u_0 in v_0 (torej položaj predstavitve), lahko najmanjši primeren t dobimo tako, da poiščemo $d_u(u_0) := \max_i |u_i - u_0|$ in $d_v(v_0) := \max_i |v_i - v_0|$ in vzamemo večjega od njiju: $t(u_0, v_0) := \max\{d_u(u_0), d_v(v_0)\}$.

Načeloma bi lahko najmanjšo vrednost $d_u(u_0)$ dobili, če poiščemo najmanjši in največji u_i (recimo jima u_{min} in u_{max}) ter postavimo u_0 ravno na pol poti med njima, torej $u_0 = (u_{min} + u_{max})/2$. Zanj je $d_u(u_0)$ enako ravno $(u_{max} - u_{min})/2$. Če tako dobljeni u_0 ni celo število, je vseeno, ali ga zaokrožimo navzdol ali navzgor, saj bomo v vsakem primeru dobili $d_u(u_0) = \lceil (u_{max} - u_{min})/2 \rceil$. (Primer: če imamo $u_{min} = 5$, $u_{max} = 10$, lahko $(5+10)/2 = 7{,}5$ zaokrožimo bodisi na 7 bodisi na 8, pa bomo v vsakem primeru do enega od krajišč, bodisi u_{min} bodisi u_{max}, oddaljeni za $\lceil (10 - 5)/2 \rceil = 3$.) Enak razmislek lahko seveda ponovimo tudi za v-koordinate in tako dobimo še v_0.

Težava pa je, da je naša prvotna mreža (x, y) diskretna, koordinate x in y so cela števila, zato ni vsak par koordinat (u, v) veljaven. Iz $u - v = (x + y) - (x - y) = 2y$ sledi, da je razlika $u - v$ vedno soda, torej sta u in v enake parnosti. Pri naši ideji, da bi vzeli $u_0 = (u_{min} + u_{max})/2$ in $v_0 = (v_{min} + v_{max})/2$, pa se čisto lahko zgodi, da dobimo u_0 in v_0 različne parnosti. V prvotnem koordinatnem sistemu ustreza taki točki par ne-celih koordinat (x, y). (Na primer: če je $u_{min} = 3$, $u_{max} = 7$, $v_{min} = 0$, $v_{max} = 4$ dobimo $u_0 = 5$, $v_0 = 2$, kar ustreza $x_0 = 3{,}5$, $y_0 = 1{,}5$.)

V primerih, ko smo imeli necelo\v{s}tevilski $(u_{min} + u_{max})/2$, se tej težavi zlahka izognemo: odvisno od tega, ali to vrednost zaokrožimo navzdol ali navzgor, dobimo enkrat sodo število, enkrat pa liho (obe izbiri pa nam dasta enako $d_u(u_0)$, torej sta v tem smislu obe enako dobri); od njiju torej za u_0 vzemimo tisto, ki se po parnosti ujema z v_0. Podobno lahko razmišljamo, če imamo takšno možnost izbire glede zaokrožanja pri v_0 namesto pri u_0.

Ostane nam še primer, ko sta tako $u_0 = (u_{min} + u_{max})/2$ kot $v_0 = (v_{min} + v_{max})/2$ že celi števili, vendar različne parnosti. V tem primeru nam ne ostane drugega, kot da enega od njiju povečamo ali zmanjšamo za 1. Recimo, da to naredimo z u_0; njegova vrednost $d_u(u_0)$ zdaj ni več enaka $(u_{max} - u_{min})/2$, pač pa $(u_{max} - u_{min})/2 + 1$. Podobno je, če za 1 povečamo ali zmanjšamo vrednost v_0. Ker bi radi, da je večja od vrednosti $d_u(u_0)$ in $d_v(v_0)$ čim manjša (to je namreč $t(u_0, v_0)$ — število dni, v katerem bodo obveščeni vsi potencialni kupci), popravimo raje tisto koordinato, pri kateri je bila ta vrednost doslej manjša; torej, če je bilo $d_u(u_0) < d_v(v_0)$, popravimo raje u_0, koordinato v_0 pa pustimo pri miru; tako bo $d_v(v_0)$ po novem še vedno $\ge d_u(u_0 \pm 1)$, torej bo $t(u_0 \pm 1, v_0) = \max\{d_u(u_0 \pm 1), d_v(v_0)\} = d_v(v_0) = \max\{d_u(u_0), d_v(v_0)\} = t(u_0, v_0)$; z drugimi besedami, rešitev se ne bo zaradi tega popravka celo nič poslabšala. Podobno

lahko razmišljamo, če je bilo $d_u(u_0) > d_v(v_0)$, le da takrat pač raje premaknemo v_0 namesto u_0. Če pa je veljalo $d_u(u_0) = d_v(v_0)$, je zdaj vseeno, ali premaknemo u_0 ali v_0, saj se bo $t(u_0, v_0)$ v vsakem primeru povečal za 1.

```
#include <stdio.h>
int main()
{
  int h, w, k, x, y, u, v, uMin, uMax, vMin, vMax, i, t;
  /* Preberimo velikost mreže in število točk. */
  FILE *f = fopen("reklama.in", "rt");
  fscanf(f, "%d %d %d", &h, &w, &k);
  /* Preberimo koordinate točk, jih preračunajmo v koordinatni sistem (u, v)
     in poiščimo največjo in najmanjšo koordinato na vsaki osi. */
  for (i = 0; i < k; i++)
  {
    fscanf(f, "%d %d", &y, &x);
    u = x + y; v = x - y;
    if (i == 0) uMin = u, uMax = u, vMin = v, vMax = v;
    if (u < uMin) uMin = u; if (u > uMax) uMax = u;
    if (v < vMin) vMin = v; if (v > vMax) vMax = v;
  }
  fclose(f);
  /* Čas obveščanja je načeloma ceil(max(uMax - uMin, vMax - vMin)/2), ki je dosežen,
     če predstavitev izvedemo v u0 = (uMax - uMin) / 2, v0 = (vMax - vMin) / 2. */
  t = uMax - uMin; if (vMax - vMin > t) t = vMax - vMin;
  /* Posebej pa moramo paziti na primere, ko tako dobljen par (u0, v0) ni veljaven,
     vsi njegovi veljavni sosednje pa dajo za 1 dan daljši čas obveščanja. */
  if (uMax - uMin == vMax - vMin &&
      abs(uMax - uMin) % 2 == 0 && abs(uMin - vMin) % 2 == 1) t++;
  /* Izpišimo rezultat. */
  f = fopen("reklama.out", "wt");
  fprintf(f, "%d\n", (t + 1) / 2); fclose(f); return 0;
}
```

2. Kompresija slike

Recimo, da smo si naših k odtenkov sive že nekako izbrali in jih oštevilčili v naraščajočem vrstnem redu: $0 \le b_1 < b_2 < \ldots < b_k < B$, pri čemer je $B = 1001$ število vseh odtenkov v našem barvnem prostoru. Naloga pravi, da moramo minimizirati vsoto absolutnih vrednosti napak, torej bo najbolje, če barvo vsakega piksla zamenjamo z najbližjo b_i. Če leži neka barva b točno na pol poti med b_i in b_{i+1}, je vseeno, ali piksle barve b zamenjamo z b_i ali z b_{i+1}; lahko bi celo zamenjali nekatere z eno in nekatere z drugo, ampak od tega ne bi bilo nobene posebne koristi; dogovorimo se na primer, da bomo v takem primeru vedno uporabili temnejšega od obeh odtenkov, torej b_i.

Vidimo torej, da se lahko pri naši kompresiji omejimo tako, da (za vsak b) vse piksle barve b prebarvamo z istim barvnim odtenkom, namreč s tistim b_k, ki minimizira $|b-b_i|$. Ker se vsi piksli barve b preslikajo v isti odtenek b_i, je za potrebe našega razmisleka pravzaprav vseeno, kateri piksli so to oz. kje na sliki ležijo; pomembno je le to, koliko jih je. Iz vhodne slike je zato koristno izračunati histogram h — to je tabela z B celicami, v kateri element h_b pove, koliko pikslov na sliki je barve b.

Iz dosedanjega razmisleka tudi vidimo, da bo v komprimirani sliki vsak odtenek b_i pokril nek strnjen interval barv iz prvotne slike (načeloma vse tiste b, za katere je $(b_{i-1}+b_i)/2 < b \le (b_i+b_{i+1})/2$; posebej moramo paziti le na spodnjo mejo pri b_1 in na zgornjo mejo pri b_k). Označimo te meje med intervali s c_i; imamo torej $0 = c_0 \le c_1 \le c_2 \le \ldots \le c_{k-1} \le c_k = B$ in ta števila nam povedo, da se pri kompresiji v odtenek b_i preslikajo vsi piksli, katerih barva b leži na intervalu $c_{i-1} \le b < c_i$.

Namesto da iščemo najboljši nabor odtenkov $(b_1, \ldots, b_k)$, bi lahko torej rekli, da iščemo najboljše razbitje intervala $0, \ldots, B-1$ na k podintervalov; oz. z drugimi besedami, da iščemo najboljše zaporedje mej $(c_0, c_1, \ldots, c_{k-1}, c_k)$. Med vsemi temi razbitji

na podintervale je gotovo tudi tisto, ki ustreza najboljšemu možnemu naboru odtenkov, tako da, če bomo našli najboljše razbitje, bomo s tem dobili najboljšo rešitev naloge.

Recimo zdaj, da imamo v mislih neko konkretno razbitje na podintervale, in si oglejmo v njem nek konkreten interval, na primer kar tistega najsvetlejšega, od c_{k-1} do $c_k - 1$ (ki se bo pri kompresiji preslikal v odtenek b_k). Za vsako barvo b s tega intervala (torej $c_{k-1} \le b < c_k$) imamo na sliki h_b pikslov te barve in vsak od njih prispeva k napaki naše komprimirane predstavitve slike vrednost $|b - b_k|$; skupaj to nanese $\sum_{b=c_{k-1}}^{c_k-1} h_b|b-b_k|$. Vidimo torej, da na napako pri teh pikslih čisto nič ne vplivajo ostali odtenki $(b_1, \ldots, b_{k-1})$; še več, na napako pri teh pikslih ne vpliva niti to, kako smo na podintervale razbili preostale barve (tiste od 0 do $c_{k-1}-1$). Podobno tudi izbor odtenka b_k nič ne vpliva na napako pri pikslih barv od 0 do $c_{k-1} - 1$; z drugimi besedami, ko si enkrat izberemo c_{k-1} (drugo krajišče pa je tako ali tako fiksirano: $c_k = B$), lahko izberemo zanj najboljši b_k neodvisno od tega, kako bomo izbrali $c_1, \ldots, c_{k-2}$. Podobno tudi na napako pri pikslih barv od 0 do $c_{k-1} - 1$ nič ne vpliva to, kaj smo si mi izbrali za b_k. Ker vnaprej ne vemo, kateri c_{k-1} bo pripeljal do najmanjše napake, bomo morali pač preizkusiti vse možnosti. Tako smo prišli do naslednje rekurzivne rešitve:

algoritem NajboljšeRazbitje(c_k, k):
(* *Izračuna napako najboljšega razbitja intervala* $0, \ldots, c_k - 1$ *na* k *podintervalov.* *)
1 za vsak c_{k-1} od 0 do $c_k - 1$:
2 $E[c_{k-1}] := \infty$;
3 za vsak b_k od c_{k-1} do $c_k - 1$:
4 izračunaj napako $E[c_{k-1}, b_k] := \sum_{b=c_{k-1}}^{c_k-1} h_b|b - b_k|$;
5 če je ta napaka manjša od $E[c_{k-1}]$, jo shrani v $E[c_{k-1}]$;
6 $E[c_{k-1}] := E[c_{k-1}] +$ NajboljšeRazbitje($c_k, k-1$);
7 med vsemi tako dobljenimi $E[c_{k-1}]$ vrni najmanjšo;

Posebej bi morali paziti še na robni primer: ko je $k = 1$, torej ko hočemo en sam interval, ne res razbitja na več podintervalov, pride v poštev le $c_{k-1} = 0$, saj takrat ni nobenih drugih podintervalov, ki bi jim lahko prepustili barve pod c_{k-1}.

Ta rešitev torej deluje tako, da preizkusi vse možnosti glede levega krajišča zadnjega (najsvetlejšega) podintervala, torej c_{k-1}; pri vsaki c_{k-1} poišče najmanjšo napako za dobljeni podinterval (v ta namen preizkusi vse možne b_k in vzame tistega z najmanjšo napako) in najmanjšo napako za preostale barve (od 0 do $c_{k-1} - 1$, če se jih razbije na $k - 1$ podintervalov; v ta namen uporabimo rekurzivni klic); na koncu pa uporabi tisto c_{k-1}, ki je dala najmanjšo skupno napako.

Opisana rešitev je pravilna, vendar neučinkovita; na srečo pa je ni pretežko predelati v učinkovito rešitev. Za začetek opazimo, da če se naš podprogram kliče večkrat za isti par (B, k), bo vračal vedno enake rezultate, torej bi si bilo koristno že izračunane rezultate shranjevati v neki tabeli, da ne bomo računali istih stvari po večkrat. Opazimo lahko celo, da ko računamo rezultate za nek k, potrebujemo le rezultate za $k - 1$, ne pa tudi tistih za $k - 2$, $k - 3$ in tako naprej — tiste lahko torej sproti pozabljamo in tako prihranimo še nekaj pomnilnika. Tako imamo zdaj rešitev, ki izvede izračun za vsak par (c_k, k) le enkrat, pri tem pa ima $O((c_k)^3)$ dela (dve gnezdeni zanki v vrsticah 1 in 3 ter še tretja, ki je skrita v vsoti v vrstici 4); ker gre lahko c_k do B, je skupna časovna zahtevnost te rešitve $O(B^4 k)$.

Naslednji pomembni prihranek pa je povezan z iskanjem najboljšega b_k. Recimo, da je na našem intervalu od c_{k-1} do $c_k - 1$ skupaj N pikslov, od česar jih je n svetlejših od b_k in n' temnejših od b_k. Imamo torej $N = \sum_{b=c_{k-1}}^{c_k-1} h_b$ in $n = \sum_{b=b_k+1}^{c_k-1} h_b$ in seveda $n' = N - n - h_{b_k}$. Kaj se zgodi z napako $\sum_b h_b|b - b_k|$, če povečamo b_k za 1? Pri vseh pikslih, ki so svetlejši od (stare) b_k, se napaka zmanjša za 1; pri vseh, ki so bili prej barve b_k ali temnejše, pa se poveča za 1. Napaka se torej skupno zmanjša za $n - (N - n) = 2n - N$; če je to > 0, je nova vrednost b_k boljša od prejšnje. Podoben razmislek pokaže, da če b_k zmanjšamo za 1, se napaka skupno zmanjša za $n' - (N - n') = 2n' - N$. Najboljši možni b_k bomo seveda prepoznali po tem, da noben od teh dveh premikov ne zmanjša njegove napake, saj je imel že prej najmanjšo možno napako. Za najboljši b_k torej velja, da je $2n - N \le 0$, pa tudi $2n' - N \le 0$. Iz prvega dobimo $n \le N/2$, iz drugega pa $n' \le N/2$, kar je (če vstavimo $n' = N - n - h_{b_k}$) isto kot $n + h_{b_k} \ge N/2$. Z drugimi besedami,

najboljši b_k je ravno mediana barve vseh pikslov z intervala od c_{k-1} do $c_k - 1$: kvečjemu polovica teh pikslov sme biti svetlejša od b_k, kvečjemu polovica sme biti temnejša od b_k.

Zanko v vrsticah 3 in 4 gornjega postopka (pravzaprav dve zanki, ker se ena skriva še v izračunu vsote v vrstici 4) bi torej lahko nadomestili s preprostejšo in učinkovitejšo zanko, ki bi poiskala mediano. Lepo pri tem je, da če se na primer b_k premakne za 1, se n in n' le malo spremenita (če se b_k zmanjša za 1, se n poveča za h_b), tako da ju ni treba računati vsakič znova. Vrstice 2–5 moramo zamenjati z nečim takšnim:

$N := 0$; za vsak b od c_{k-1} do $c_k - 1$: $N := N + h_b$;
$b_k := c_k - 1$; $n := 0$;
while $b_k \geq c_{k-1}$:
  $n := n + h_{b_k}$;
  if $n + h_{b_k} \geq N/2$ then (* *našli smo mediano* *) break;
  $b_k := b_k - 1$;
(* *Zdaj poznamo mediano b_k in lahko izračunamo njeno napako.* *)
$E[c_{k-1}] := \sum_{b=c_{k-1}}^{c_k-1} h_b|b - b_k|$;

Naša nova zanka ima časovno zahtevnost $O(B)$; zunanja zanka (tista iz vrstice 1) pa ostane in postopek kot celota (ko ga izvedemo za vse pare (c_k, k)) ima zdaj časovno zahtevnost $O(B^3k)$.

Dobljeno rešitev lahko še malo izboljšamo. Spomnimo se, da v zunanji zanki pregledamo vse možne c_{k-1}, od 0 do $c_k - 1$. Pri vsakem izračunamo b_k kot mediano barve vseh pikslov z intervala $c_{k-1}, \ldots, c_k - 1$. Toda mediane, ki jih dobimo pri različnih c_{k-1}, so si med seboj tesno povezane. Recimo, da že poznamo pravo mediano pri neki vrednosti c_{k-1}; kaj se zgodi, če c_{k-1} zmanjšamo za 1? Z drugimi besedami, kaj se zgodi, če v interval barv, ki naj bi ga pokril odtenek b_k, dodamo še eno malo temnejšo barvo? Nova mediana po tem seveda ne more biti svetlejša od stare — lahko je kvečjemu enaka ali temnejša. Vsota N se poveča na $N' := N + h_{c_{k-1}-1}$, vsota n pa se nič ne spremeni (če ne spremenimo b_k). Ker je bila b_k prej prava mediana, velja zanjo $n + h_{b_k} \geq N/2$ in $n \leq N/2$; zdaj bi želeli, da bi veljalo isto za N' namesto N. Ker je $n \leq N/2$ in $N' \geq N$, velja tudi $n \leq N'/2$. Za $n + h_{b_k}$ pa ni nujno, da je $\geq N'/2$; in če ni, moramo b_k malo zmanjšati: pri tem se bo n malo povečal in prej ali slej bo pogoj $n + h_{b_k} \geq N'/2$ izpolnjen. Tako smo dobili naslednji postopek:

$b_k := c_k$; $n := 0$; $N := 0$;
za vsak c_{k-1} od $c_k - 1$ do 0 (v padajočem vrstnem redu):
  $N := N + h_{c_{k-1}}$;
  while $2(n + h_{b_k}) < N$:
    $n := n + h_{b_k}$; $b_k := b_k - 1$;
  (* *Zdaj je b_k mediana barve vseh pikslov z intervala od c_{k-1} do $c_k - 1$.* *)

Obe zanki — zunanja, ki zmanjšuje c_{k-1}, in notranja, ki zmanjšuje b_k, sta zdaj prepleteni; ko zunanja opravi vseh c_k iteracij, opravi tudi notranja največ toliko iteracij, saj se b_k pri njej ves čas zmanjšuje (od začetne vrednosti c_k do neke končne vrednosti, ki je nekje od 0 do c_k). Zato imata obe zanki skupaj časovno zahtevnost le $O(c_k)$.

Toda račun mediane nam še ni dovolj, mi bi radi pri vsakem c_{k-1} izračunali tudi vsoto absolutnih vrednosti napak, ki jo dobimo, če vse barve z intervala $c_{k-1}, \ldots, c_k - 1$ predstavimo z njihovo mediano. Če bomo za to pri vsakem c_{k-1} izvedli še eno vgnezdeno zanko, ki bo računala vsoto $E[c_{k-1}] = \sum_{b=c_{k-1}}^{c_k-1} h_b|b - b_k|$, bomo pristali pri enaki časovni zahtevnosti kot prej: $O(B^2)$ za vsak par (c_k, k) in zato $O(B^3k)$ za celoten postopek. Na srečo lahko tudi to počnemo učinkoviteje. Recimo, da že poznamo napako za neko konkretno vrednost c_{k-1} in b_k; v našem postopku se ti dve spremenljivki občasno zmanjšata za 1 in ugotoviti moramo, kako takrat čim ceneje izračunati novo napako. Če se c_{k-1} zmanjša za 1 (njegovi novi vrednosti recimo c'_{k-1}, dobi ta vsota nov člen z vrednostjo $h_{c'_{k-1}}|c'_{k-1} - b_k|$. Če pa se b_k zmanjša za 1, se zgodi naslednje: napaka pri vseh n piksli, ki so bili že prej svetlejši od mediane, se zdaj poveča za 1 (ker je nova mediana še temnejša od prejšnje); napaka pri vseh n' pikslih, ki so bili prej temnejši od mediane, pa se zdaj zmanjša za 1. Napaki moramo torej preprosto prišteti $n - n'$. Tako torej vidimo, da nam računanje nove napake po vsaki spremembi c_{k-1} in b_k vzame le

$O(1)$ časa; pri vsakem paru (c_k, k) porabimo torej le $O(c_k)$ časa in postopek kot celota reši nalogo v $O(B^2k)$ časa.

Zapišimo dobljeni postopek še v C-ju:

```
#include <stdio.h>
#define B 1000
int main()
{
  int w, h, k, b, bb, i, kk, N, n, m, d, kand;
  int hist[B + 1], ff[2][B + 1], *f0, *f1;
  /* Preberimo vhodno datoteko. */
  FILE *f = fopen("slika.in", "rt"); fscanf(f, "%d %d %d", &h, &w, &k);
  for (b = 0; b <= B; b++) hist[b] = 0;
  for (i = 0; i < w * h; i++) { fscanf(f, "%d", &b); hist[b]++; }
  fclose(f);
  for (b = 0; b <= B; b++) ff[0][b] = 0;

  for (kk = 1; kk <= k; kk++) {
    f0 = &ff[(kk - 1) % 2][0]; f1 = &ff[kk % 2][0];
    /* Zdaj bomo reševali podprobleme s kk odtenki. Rezultate bomo
       shranjevali v tabelo f1, rešitve podproblemov s kk - 1 odtenki
       pa imamo že izračunane v tabeli f0. */
    for (b = 0; b <= B; b++)
    {
      /* Podproblem: radi bi predstavili barve 0, ..., b s samo kk odtenki.
         Pri tem bo odtenek kk pokril barve bb, ..., b. */
      m = b; /* Mediana barv pikslov z območja bb, ..., b. */
      N = 0; /* Število pikslov na območju od bb, ..., b. */
      n = 0; /* Število pikslov na območju m + 1, ..., b. */
      d = 0; /* Vsota absolutnih vrednosti napak za območje od bb do b. */
      f1[b] = (w + 1) * (h + 1) * (B + 1);
      for (bb = b; bb >= 0; bb--)
      {
        N += hist[bb]; d += (m - bb) * hist[bb];
        while (2 * (n + hist[m]) < N) { /* Popravimo mediano. */
          n += hist[m]; m--;
          d = d - (N - n) + n; }
        if (bb > 0 && kk == 1) continue;
        kand = (bb == 0 ? 0 : f0[bb - 1]) + d;
        if (kand < f1[b]) f1[b] = kand;
      } /* for bb */
    } /* for b */
  } /* for kk */

  /* Izpišimo rezultat. */
  f = fopen("slika.out", "wt"); fprintf(f, "%d\n", ff[k % 2][B]); fclose(f); return 0;
}
```

3. Konstrukcija grafa

Radi bi sestavili usmerjen graf, v katerem je od točke s do točke t natanko n različnih sprehodov. Za začetek se prepričajmo, da ni nobene koristi od tega, da bi imeli v grafu kakšne cikle. Če je v grafu kakšen tak cikel, ki vsebuje kakšno točko, ki leži na neki poti od s do t, lahko v to pot vključimo še enega ali več obhodov po tem ciklu in tako dobimo neskončno različnih sprehodov od s do t; naloga pa zahteva le n takih sprehodov. Če pa v grafu sicer je nek cikel, vendar nobena njegova točka ne leži na kakšni poti od s do t, potem je z vidika našega problema vseeno, če tega cikla sploh ne bi bilo (pravzaprav lahko iz grafa pobrišemo vse točke, ki niso dosegljive iz s ali pa iz njih ni dosegljiva t).

Omejimo se torej na aciklične grafe. Naj bo $f(u)$ število različnih poti od u do t; očitno je $f(t) = 1$ (pot dolžine 0 od t do t) in $f(u) = \sum_v f(v)$, pri čemer gre vsota po vseh takih v, za katere obstaja neposredna povezava $u \to v$.

Največ poti bomo torej dobili, če vključimo v graf kar vse možne povezave. Poleg točke t imejmo še zaporedje točk $u_0, u_1, u_2, \ldots, u_k$ in povezave $(u_i \to u_j)$ za vsak par točk, pri katerem je $0 \le j < i \le k$. Poleg tega imejmo še povezavo $(u_i \to t)$ za vsak $i = 0, \ldots, k$.

Tako smo dobili graf s $k+2$ točkami in hitro se lahko prepričamo, da je v njem natanko 2^i poti od točke u_i do točke t. Število n lahko zapišemo kot vsoto nekaj različnih potenc števila 2 — to je pravzaprav isto, kot če bi ga pretvorili v dvojiški zapis. Na primer:

$$1234 = 1024 + 128 + 64 + 16 + 2 = 2^{10} + 2^7 + 2^6 + 2^4 + 2^1.$$

Torej, če ustanovimo novo točko s in iz nje potegnemo neposredne povezave do u_{10}, u_7, u_6, u_4 in u_1, bo od s do t natanko 1234 različnih poti. Pri naši nalogi gre lahko n do 10^8, kar je med 2^{26} in 2^{27}, torej nam bodo zadoščale potence do 2^{26}. Tako dobimo graf s točkami $t, u_0, u_1, \ldots, u_{26}$ in s; to je skupaj 29 točk, kar je še v okviru predpisanih omejitev.

Paziti moramo še na številčenje točk; naloga predpisuje, da mora biti $s = 1$ in $t = 2$. Za ostale točke je vseeno, v kakšnem vrstnem redu jih oštevilčimo; spodnji program točki u_i pripiše številko $3 + i$.

```
#include <stdio.h>
#define MaxN 100000000
#define m 26 /* floor(log2 MaxN) */
int main()
{
  int n, i, j;
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("graf.in"); fscanf(f, "%d", &n); fclose(f);
  /* Dodajmo povezave ui → t in ui → uj za i > j. */
  f = fopen("graf.out", "wt");
  for (i = 3; i <= 3 + m; i++) for (j = 2; j < i; j++) fprintf(f, "%d %d\n", i, j);
  /* Zdaj je od ui = 3 + i do t = 2 možnih natanko 2^i poti. */
  if (n > 0) {
    /* Sestavimo n iz potenc števila 2 in dodajmo povezave od s = 1 do ustreznih ui. */
    for (i = 0; i <= m; i++) if (n & (1 << i)) fprintf(f, "%d %d\n", 1, 3 + i); }
  fclose(f); return 0;
}
```

4. Mušji drekci

Imamo torej n točk in radi bi jih pokrili z dvema kvadratoma $a \times a$, katerih stranice so vzporedne koordinatnima osema. Poiščimo med temi točkami najbolj levo in ji recimo L (s koordinatama (x_L, y_L); če je več najbolj levih, torej takih z minimalno x-koordinato, je vseeno, katero od njih vzamemo za L); podobno poiščimo tudi najbolj desno točko D, najbolj spodnjo S in najbolj zgornjo Z.

Če se našo množico točk dá pokriti z dvema kvadratoma, mora eden od teh kvadratov seveda pokriti tudi točko L; recimo mu *prvi kvadrat*. (Če oba kvadrata pokrivata L, si pač izberimo za prvi kvadrat poljubnega od njiju.)

(1) Mogoče pokrije prvi kvadrat tudi točko S. Nobene koristi ni od tega, da bi bil levi rob kvadrata levo od x_L (ker tam ni nobene točke) ali da bi bil spodnji rob kvadrata nižje od y_S (ker tudi tam ni nobene točke). Torej se lahko omejimo na primere, ko je levi rob prvega kvadrata natanko na x_L, njegov spodnji rob pa natanko na y_S. S tem je položaj prvega kvadrata že popolnoma natančno določen in ni težko preveriti, katere točke bi tak kvadrat dejansko pokril. Ob tem bomo tudi videli, katerih točk ne pokrije; med temi lahko spet poiščemo minimalno in maksimalno x- in y-koordinato in vidimo, če bi se dalo vse te preostale točke pokriti z enim samim kvadratom (veljati mora $x_{max} - x_{min} \le a$ in $y_{max} - y_{min} \le a$).

(2) Mogoče prvi kvadrat ne pokrije točke S, pokrije pa točko Z. Zdaj lahko razmišljamo analogno kot v prejšnjem odstavku; zgornji rob prvega kvadrata postavimo na y_D, pogledamo, katere točke pokrije, in preverimo, če je mogoče vse preostale točke pokriti z enim samim kvadratom.

(3) Mogoče prvi kvadrat ne pokrije niti S niti Z, pokrije pa točko D. Torej mora točki S in Z pokriti drugi kvadrat. Ker prvi kvadrat pokrije L in D, mora biti $x_D - x_L \leq a$; in ker drugi kvadrat pokrije S in Z, mora biti $y_Z - y_S \leq a$; in ker so x-koordinate vseh točk na intervalu $[x_L, x_D]$, njihove y-koordinate pa na intervalu $[y_S, y_Z]$, sledi, da bi se dalo vse te točke pokriti že z enim samim kvadratom. Torej lahko primer (3) ignoriramo, ker se bo dalo takšne razporede točk pokriti tudi tako, da bo prvi kvadrat pokril L in S (in sploh vse točke), kar smo že obravnavali pod (1).

(4) Mogoče prvi kvadrat ne pokrije niti S niti Z niti D. Torej mora te tri točke pokriti drugi kvadrat. S podobnim razmislekom kot pri (1) vidimo, da lahko postavimo desni rob drugega kvadrata na x_D, njegov spodnji rob na y_S in nato preverimo, katere točke pokrije in ali je mogoče vse preostale točke pokriti z enim samim kvadratom.

Tako torej vidimo, da če je našo množico točk sploh mogoče pokriti z dvema kvadratoma, jo je mogoče pokriti na enega od načinov (1), (2) ali (4), in videli smo tudi, kako lahko za vsakega od teh treh načinov preverimo, ali je pri dani množici točk res mogoč ali ne. Če nam na nobenega od teh treh načinov ne uspe pokriti vseh točk, lahko zaključimo, da jih s samo dvema kvadratoma ni mogoče pokriti.

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

#define MaxN 1000000
#define MaxKoord 1000000000

int n, a, xs[MaxN], ys[MaxN];

/* Preveri, ali je mogoče vse točke pokriti z dvema kvadratoma,
   pri čemer ima prvi kvadrat levi rob x0 in spodnji rob y0. */
bool Test(int x0, int y0)
{
  int x1 = MaxKoord, y1 = MaxKoord, x2 = 0, y2 = 0, i;
  if (x0 < 0) x0 = 0; if (y0 < 0) y0 = 0;
  for (i = 0; i < n; i++) {
    /* Če točko i pokrije že prvi kvadrat, jo preskočimo. */
    if (x0 <= xs[i] && xs[i] - x0 <= a)
      if (y0 <= ys[i] && ys[i] - y0 <= a) continue;
    /* Ostanejo točke, ki jih bo moral pokriti drugi kvadrat.
       Zapomnimo si najmanjšo in največjo x- in y-koordinato. */
    if (xs[i] < x1) x1 = xs[i]; if (xs[i] > x2) x2 = xs[i];
    if (ys[i] < y1) y1 = ys[i]; if (ys[i] > y2) y2 = ys[i]; }
  /* Preverimo, ali je mogoče vse preostale točke pokriti z enim kvadratom. */
  return (y2 - y1 <= a && x2 - x1 <= a);
}
int main()
{
  int c, i, L, D, S, Z;
  FILE *f = fopen("musji.in", "rt"), *g = fopen("musji.out", "wt");
  fscanf(f, "%d", &c);
  while (c-- > 0)
  {
    fscanf(f, "%d", &n); fscanf(f, "%d", &a);
    /* Preberimo koordinate točk in si zapomnimo indeks najbolj
       leve, najbolj desne, najbolj spodnje in najbolj zgornje. */
    for (i = 0, L = 0, D = 0, S = 0, Z = 0; i < n; i++) {
      fscanf(f, "%d %d", &xs[i], &ys[i]);
      if (xs[i] < xs[L]) L = i; if (xs[i] > xs[D]) D = i;
      if (ys[i] < ys[S]) S = i; if (ys[i] > ys[Z]) Z = i; }
    /* Preverimo, če se da vse točke pokriti z dvema kvadratoma. */
    fprintf(g, "%s\n", Test(xs[L], ys[S]) || Test(xs[L], ys[Z] - a) ||
      Test(xs[D] - a, ys[S]) ? "DA" : "NE");
  }
  fclose(f); fclose(g); return 0;
}
```

5. Podajanje žoge

Uredimo otroke po naraščajoči x-koordinati, tiste z enako x-koordinato pa po naraščajoči y-koordinati. Tako pridejo skupaj tisti, ki bi utegnili podajati drug drugemu žogo v vodoravni smeri; za vsak par dveh zaporednih otrok v tem vrstnem redu moramo le preveriti, če imata res enako x-koordinato in če pripadata isti ekipi: če to dvoje drži, potem vemo, da lahko tadva otroka podajata žogo drug drugemu.

Za podaje v navpični smeri uporabimo enak razmislek, le vloga x- in y-koordinat je zamenjana.

Tako smo za vsakega otroka ugotovili, katerim drugim otrokom (največ štirim) lahko podaja; te podatke si zapomnimo v neki tabeli (v spodnjem programu je to sosedje). Dobili smo graf vseh možnih podaj, v njem pa nas zanima najkrajša pot od s do t; poiščemo jo lahko na primer z iskanjem v širino.

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

#define MaxN 100000

int xs[MaxN], ys[MaxN], ekipa[MaxN], is[MaxN], n, s, t, *k1, *k2;
int sosedje[MaxN][4], d[MaxN];

int Primerjaj(const void* a, const void* b)
{
  int u = *(int *) a, v = *(int *) b;
  int d = k1[u] - k1[v]; return d ? d : k2[u] - k2[v];
}

int main()
{
  int u, v, i, j, k, head, tail;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen(argc > 1 ? argv[1] : "podaje.in", "rt");
  fscanf(f, "%d", &n);
  for (u = 0; u < n; u++) {
    fscanf(f, "%d %d %d", &xs[u], &ys[u], &ekipa[u]);
    for (i = 0; i < 4; i++) sosedje[u][i] = -1;
    d[u] = -1; }
  fscanf(f, "%d %d", &s, &t); s--; t--; fclose(f);

  for (k = 0; k < 2; k++) {
    /* V tabeli is pripravimo indekse otrok (od 0 do n - 1), kazalca
       k1 in k2 pa naj kažeta na koordinate, po katerih bomo urejali
       (pri k = 0 urejamo najprej po x in nato po y, pri k = 1 pa ravno obratno. */
    for (u = 0; u < n; u++) is[u] = u;
    if (k == 0) k1 = xs, k2 = ys; else k1 = ys, k2 = xs;
    qsort(is, n, sizeof(is[0]), &Primerjaj);

    /* V dobljenem vrstnem redu za vsak par sosednjih otrok poglejmo,
       če se ujemata v prvi koordinati in pripadata isti ekipi. Če je
       to dvoje res, si lahko podajata žogo, kar si bomo zapisali v tabelo sosedje. */
    for (i = 0; i < n; i++) for (j = 0, u = is[i]; j < 2; j++) {
      v = i + j * 2 - 1; if (v < 0 || v >= n) continue; else v = is[v];
      if (ekipa[v] == ekipa[u] && (xs[v] == xs[u] || ys[v] == ys[u])) sosedje[u][2 * k + j] = v; }}

  /* Z iskanjem v širino poiščimo najkrajšo pot od s do t.
     Tabelo is uporabljamo kot vrsto. */
  head = 0; tail = 0; is[tail++] = s; d[s] = 0;
  while (head < tail && d[t] < 0)
    /* Vzemimo naslednjo točko iz vrste; recimo ji u. Preglejmo njene sosede. */
    for (u = is[head++], i = 0; i < 4; i++) {

      /* Vemo, da je mogoče do u-ja priti v d[u] korakih; torej je mogoče do
         u-jevega soseda, v, priti v d[u] + 1 korakih. Če do v-ja še ne poznamo
         kakšne boljše poti, si to zdaj zapomnimo in ga dodajmo v vrsto. */
      v = sosedje[u][i];
      if (v < 0 || d[v] >= 0) continue;
```

```
        d[v] = d[u] + 1; is[tail++] = v; }

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

Viri nalog: lego kocke, kozarci, assembler — Boris Gašperin; reklama, kompresija slike, konstrukcija grafa — Tomaž Hočevar; kemijske formule, podajanje žoge — Jurij Kodre; majevska števila — Mark Martinec; okoljevarstveni ukrepi — Mojca Miklavec; (Opomba) — Klemen Simonič in Mitja Trampuš; ve,jic,e, mušji drekci — Mitja Trampuš; vsota, telefonske številke — Janez Brank;

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