8. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

25. januarja 2013

NASVETI ZA TEKMOVALCE

Naloge na tem šolskem tekmovanju pokrivajo širok razpon težavnosti, tako da ni nič hudega, če ne znaš rešiti vseh.

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 načeloma 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 (in varneje) uporabiti fgets ali fscanf; vendar pa za rešitev naših tekmovalnih nalog zadošča tudi gets.

- Program, ki bere s standardnega vhoda po znakih, jih prepisuje na standardni izhod, na koncu pa izpiše še število prebranih znakov (ne vštevši znakov za konec vrstice):

```
program BranjeZnakov;
var i: integer; c: char;
begin
  i := 0;
  while not Eof do begin
    while not Eoln do
      begin Read(c); Write(c); i := i + 1 end;
    if not Eof then begin ReadLn; WriteLn end;
  end; {while}
  WriteLn('Skupaj ', i, ' znakov.');
end. {BranjeZnakov}
```

```
#include <stdio.h>

int main() {
  int i = 0, c;
  while ((c = getchar()) != EOF) {
    putchar(c); if (i != '\n') i++;
  }
  printf("Skupaj %d znakov.\n", i);
  return 0;
}
```

Še isti trije primeri v pythonu:

```
# Branje dveh števil in izpis vsote:
import sys

a, b = sys.stdin.readline().split()
a = int(a); b = int(b)
print "%d + %d = %d" % (a, b, a + b)
```

```
# Branje standardnega vhoda po vrsticah:
import sys

i = d = 0
for s in sys.stdin:
  s = s.rstrip('\n') # odrežemo znak za konec vrstice
  i += 1; d += len(s)
  print "%d. vrstica: \"%s\"" % (i, s)
print "%d vrstic, %d znakov." % (i, d)
```

```
# Branje standardnega vhoda znak po znak:
import sys

i = 0
while True:
  c = sys.stdin.read(1)
  if c == "": break # EOF
  sys.stdout.write(c)
  if c != '\n': i += 1
print "Skupaj %d znakov." % i
```

Še isti trije primeri v javi:

```
// Branje dveh števil in izpis vsote:
import java.io.*;
import java.util.Scanner;

public class Primer1
{
   public static void main(String[] args) throws IOException
   {
      Scanner fi = new Scanner(System.in);
      int i = fi.nextInt(); int j = fi.nextInt();
      System.out.println(i + " + " + j + " = " + (i + j));
   }
}
```

```
// Branje standardnega vhoda po vrsticah:
import java.io.*;

public class Primer2
{
   public static void main(String[] args) throws IOException
   {
      BufferedReader fi = new BufferedReader(new InputStreamReader(System.in));
      int i = 0, d = 0; String s;
      while ((s = fi.readLine()) != null) {
         i++; d += s.length();
         System.out.println(i + ". vrstica: \"" + s + "\""); }
      System.out.println(i + " vrstic, " + d + " znakov.");
   }
}
```

```
// Branje standardnega vhoda znak po znak:
import java.io.*;

public class Primer3
{
   public static void main(String[] args) throws IOException
   {
      InputStreamReader fi = new InputStreamReader(System.in);
      int i = 0, c;
      while ((c = fi.read()) >= 0) {
         System.out.print((char) c); if (c != '\n' && c != '\r') i++; }
      System.out.println("Skupaj " + i + " znakov.");
   }
}
```

8. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

25. januarja 2013

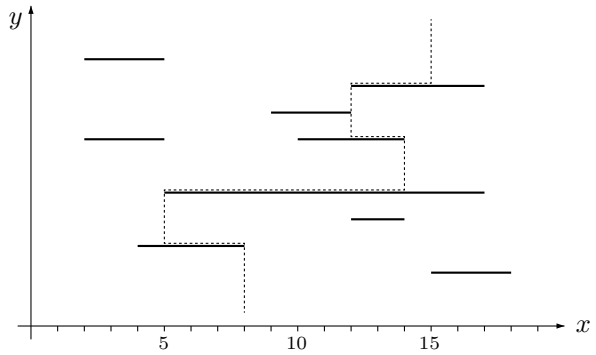
NALOGE ZA ŠOLSKO TEKMOVANJE

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.

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

1. Lemingi

Naloga temelji na spremenjeni verziji starejše igre Lemingi. Pri tej verziji igre je igralno polje sestavljeno iz vodoravnih daljic, ki predstavljajo ploščadi, po katerih se lahko leming premika levo ali desno. Če stopi čez rob, pade naravnost navzdol na naslednjo ploščad, če ta obstaja. Ko leming prileti na neko ploščad, se mu smer gibanja glede na prejšnjo obrne. Leming se pri padcu ne poškoduje, ne glede na to, s kakšne višine je padel. Če leming pri padanju zgolj oplazi krajišče ploščadi, se to ne šteje za padec na to ploščad. Ploščadi se med seboj ne prekrivajo in ne dotikajo. Lemingi so manjši od najmanjšega možnega višinskega razmaka med ploščadmi, torej se v ploščad ne morejo zadeti od strani.



Opiši postopek, ki glede na dano igralno polje ter dano začetno x-koordinato leminga in smer gibanja (leming vedno začne na takšni višini, da je nad vsemi ploščadmi) določi koordinato v x smeri, kjer leming nazadnje pade čez rob kakšne ploščadi (ko torej pod njim ne obstaja nobena druga ploščad več, na kateri bi pristal). Če pa leming ne pade na nobeno ploščad, naj tvoj postopek vrne kar začetno x-koordinato leminga.

Predpostavi, da so vhodni podatki že shranjeni v nekih spremenljivkah oz. tabelah: n je število ploščadi, x_i in y_i sta koordinati levega krajišča i-te ploščadi (za $i = 1, 2, \ldots, n$), d_i je dolžina i-te ploščadi, x_0 je začetna x-koordinata leminga, s_0 pa njegova začetna smer („`L`“ za levo ali „`D`“ za desno). Pri tem so ploščadi oštevilčene od višjih proti nižjim (z drugimi besedami, velja $y_1 \geq y_2 \geq \ldots \geq y_n$).

Gornja slika kaže primer, pri čemer debele črte predstavljajo ploščadi (teh je $n = 9$), črtkana črta pa pot leminga, če je $x_0 = 15$ in $s_0 =$ `D` (začetna smer leminga je v desno; ko prvič prileti na ploščad, se mu smer spremeni v levo, zato se po tej ploščadi premika v levo in tako naprej). Vidimo lahko, da je njegova končna x-koordinata (torej rezultat, ki ga mora poiskati tvoj postopek) v tem primeru $x = 8$.

2. 3-d šah

Predstavljajmo si trodimenzionalno šahovnico, v kateri posamezna polja niso kvadratki, ampak kockice, pa tudi celotna šahovnica je kocka, ki jo sestavlja $8 \times 8 \times 8$ polj. V to šahovnico postavimo tri kraljice. **Napiši program**, ki prebere položaj vseh treh kraljic in izpiše število polj, ki jih istočasno napadajo vse tri kraljice. Položaj vsake kraljice je opisan v svoji vrstici, v njej pa so tri cela števila od 1 do 8, ki podajajo koordinate kraljice. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `kraljice.txt` (kar ti je lažje).

Pravila za to, katera polja napada posamezna kraljica, so podobna kot pri običajnem šahu v dveh dimenzijah. Vsaka kraljica napada:

- 3 linije (v smeri vsake osi)
- 3×2 ravninski diagonali (v vsaki od ravnin, na katerih leži polje s kraljico)
- 4 prostorske diagonale.

To, ali neka kraljica napada neko polje ali ne, je neodvisno od tega, ali med to kraljico in tistim poljem stoji še kakšna druga kraljica ali ne (kraljice torej ne blokirajo napadov druga druge). Kraljica napada tudi polje, na katerem stoji.

3. Brisanje parov

Dan je niz, ki ga sestavljajo male in velike črke. Če sta mala in ustrezna velika črka druga za drugo, ju zbrišemo; to ponavljamo, dokler se da. Tako na primer iz `abbBAacCAdCcaBb` dobimo `abAdCcaBb`.

Napiši podprogram Okrajsaj(niz), ki izračuna ustrezno spremenjen (okrajšan) niz. Vseeno je, ali tvoj podprogram vrne okrajšani niz kot funkcijsko vrednost (npr. s stavkom **return**) ali pa kar spremeni vhodni parameter niz tako, da le-ta ob vrnitvi iz podprograma vsebuje skrajšano različico niza.

Predpostavi, da vemo, da so v nizu le male in velike črke (in nobenih drugih znakov) in da sta na voljo podprograma JeMalaCrka(znak) (ki vrne **true**, če je znak mala črka, sicer pa **false**) in DajVeliko(znak) (če je znak mala črka, vrne ustrezno veliko črko, drugače pa vrže izjemo). Tvoj podprogram naj bo čim bolj učinkovit, da bo deloval hitro tudi za zelo dolge vhodne nize; rešitev, ki ima časovno zahtevnost $O(n^2)$ namesto $O(n)$ (če je n dolžina vhodnega niza) lahko dobi pri tej nalogi največ 12 točk od 20).

4. Ta5nik

Pri pisanju kratkih sporočil si včasih prihranimo nekaj tipkanja tako, da kakšen podniz znakov v besedi nadomestimo s krajšim nizom po kakšnem preprostem ustaljenem pravilu. Tako lahko na primer namesto „`stric`“ zapišemo „`s3c`“, namesto „`tapetnik`“ pa „`ta5nik`“. V nekaterih nizih je mogoče napraviti celo več kot eno zamenjavo, na primer: `1kostra0en`, `3gonome3ja`, `cen35alen`, `pr15`, `u01`. Če se v nekem nizu več možnih zamenjav prekriva, se dogovorimo, da vedno uporabimo najbolj levo med njimi; tako na primer iz besede „`petrijevka`“ dobimo „`5rijevka`“ in ne „`pe3jevka`“.

Podan imamo naslednji seznam dovoljenih zamenjav:

`nič` → `0`	`ena` → `1`	`dve` → `2`	`tri` → `3`	`štiri` → `4`
`pet` → `5`	`šest` → `6`	`sedem` → `7`	`osem` → `8`	`devet` → `9`

Napiši podprogram `Ta5nik(s)`, ki v nizu `s` opravi vse možne zamenjave, kot to določa tabela možnih zamenjav, nato pa, če je bilo možno opraviti več kot eno zamenjavo, predelani niz izpiše, sicer pa ne izpiše ničesar.

Primer: pri klicu `Ta5nik("enakostraničen")` naj izpiše `1kostra0en`; če pa pokličemo `Ta5nik("tapetnik")`, naj ne izpiše ničesar.

5. Koalicije

V nekem parlamentu sedijo poslanci n različnih strank (vsak poslanec pripada natanko eni stranki), in sicer je iz i-te stranke natanko n_i poslancev (za $i = 1, 2, \ldots, n$). Radi bi sestavili čim večjo koalicijo (torej skupino strank, ki imajo skupaj čim več poslancev), vendar z naslednjo omejitvijo: dan je seznam parov strank, ki se med sabo ne morejo prenašati, in od vsakega takega para hočemo imeti v naši koaliciji natanko eno stranko (ne pa obeh ali nobene od njiju). **Opiši postopek**, ki na podlagi teh podatkov izračuna velikost (skupno število poslancev) največje možne koalicije, ki ustreza danim omejitvam (ali pa ugotovi, da taka koalicija sploh ne obstaja).

Primer: recimo, da imamo $n = 6$ strank s takšnim številom poslancev: $n_1 = 5$, $n_2 = 10$, $n_3 = 3$, $n_4 = 4$, $n_5 = 2$, $n_6 = 1$; in da so pari nasprotujočih si strank naslednji: $(1, 2)$, $(3, 2)$, $(2, 6)$ in $(4, 5)$. Potem se izkaže, da ima največja primerna koalicija 14 poslancev (sestavljata jo stranki 2 in 4).

8. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

25. januarja 2013

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Lemingi

Nalogo lahko rešimo tako, da simuliramo gibanje leminga. Njegovo trenutno x-koordinato hranimo v spremenljivki x, smer pa v s. Na začetku ju inicializiramo na x_0 oz. s_0 (začetni položaj in smer), nato pa pregledujemo ploščadi od višjih proti nižjim (besedilo naloge pravi, da so tako ali tako že podane v tem vrstnem redu) in pri vsaki pogledamo, ali bi leming padel nanjo ali ne; če pade na ploščad, spremenimo njegovo smer in izračunamo x-koordinato krajišča, pri kateri bo padel z nje. Ob koncu tega postopka imamo v spremenljivki x ravno zadnjo x-koordinato, ki jo leming na ta način doseže. S psevdokodo bi lahko naš postopek opisali takole:

$x := x_0$; $s := s_0$;
for $i := 1$ **to** n: (* *Pregledujmo ploščadi od višjih proti nižjim.* *)
  if $x_i < x$ **and** $x < x_i + d_i$: (* *Ali pade leming na i-to ploščad?* *)
    if $s =$ D: (* *Spremenimo smer iz desne v levo in pademo z levega krajišča.* *)
      $s :=$ L; $x := x_i$;
    else: (* *Spremenimo smer iz leve v desno in pademo z desnega krajišča.* *)
      $s :=$ D; $x := x_i + d_i$;
return x;

Zapišimo še rešitev v C-ju:

```
int Leming(int x0, char s0, int n, const int xi[], const int yi[], const int di[])
{
  int i, x = x0; char s = s0;
  for (i = 0; i < n; i++)
    if (xi[i] < x && x < xi[i] + di[i])
      if (s == 'D') s = 'L', x = xi[i];
      else s = 'D', x = xi[i] + di[i];
  return x;
}
```

2. 3-d šah

Šahovnico predstavimo s trodimenzionalno tabelo $8 \times 8 \times 8$ celih števil, v kateri vsaka celica predstavlja eno od polj šahovnice, vrednost te celice pa nam pove, koliko kraljic napada to polje. Na začetku torej postavimo vsa števila v tabeli na 0, nato pa se za vsako kraljico sprehodimo po vseh poljih, ki jih napada, in pripadajoče elemente tabele povečamo za 1. Na koncu moramo tako le še prešteti, koliko elementov tabele ima vrednost 3 (kar je znak, da tisto polje napadajo vse tri kraljice).

Ostane še vprašanje, kako za neko kraljico ugotoviti, katera polja napada (v spodnjem programu se s tem ukvarja podprogram ObdelajKraljico). Tega se lahko lotimo na več načinov, na primer takole. Smer napada kraljice lahko opišemo s trojico števil $(\triangle x, \triangle y, \triangle z)$, pri čemer je vsako od njih lahko -1, $+1$ ali 0 in nam pove, kako se v tej smeri spreminja ena od koordinat (torej $\triangle x = 1$ pomeni, da se x-koordinata povečuje, $\triangle y = -1$ pomeni, da se y-koordinata zmanjšuje in podobno). Ne glede na položaj kraljice in opazovano smer pa vemo, da kraljica ne bo napadala več kot sedem polj v tisti smeri, saj bi potem zagotovo že padli čez rob šahovnice. Naš podprogram lahko torej z nekaj zankami pregleda vse možne smeri napada in se v vsaki smeri sprehodi sedem polj daleč ter vsa tako dosežena polja označi kot napadena (v ta namen imamo še pomožni podprogram Oznaci, ki pred tem še preveri, ali ne leži zahtevano polje mogoče že zunaj šahovnice).

```
#include <stdio.h>

int a[8][8][8]; /* a[x][y][z] pove, koliko kraljic napada polje (x, y, z) */

/* Označi dano polje kot napadeno (če ne leži zunaj šahovnice). */
int Oznaci(int x, int y, int z) {
  if (0 <= x && x < 8 && 0 <= y && y < 8 && 0 <= z && z < 8) a[x][y][z]++; }

/* Obišče vsa polja, ki jih napada kraljica s položaja (x, y, z). */
int ObdelajKraljico(int x, int y, int z)
{
  int dx, dy, dz, d;
  Oznaci(x, y, z); /* Kraljica napada tudi polje, na katerem stoji. */
  /* Preglejmo vse možne smeri napada. */
  for (dx = -1; dx <= 1; dx++) for (dy = -1; dy <= 1; dy++)
  for (dz = -1; dz <= 1; dz++)
    /* Smer dx = dy = dz = 0 je neveljavna, saj se koordinate takrat ne spreminjajo. */
    if (dx != 0 || dy != 0 || dz != 0)
      /* Obiščimo 7 polj v tej smeri in jih označimo kot napadena. */
      for (d = 1; d <= 7; d++) Oznaci(x + dx * d, y + dy * d, z + dz * d);
}

int main()
{
  int x, y, z, n;
  /* Inicializirajmo vse celice tabele a na 0. */
  for (x = 0; x < 8; x++) for (y = 0; y < 8; y++) for (z = 0; z < 8; z++)
    a[x][y][z] = 0;
  /* Preberimo položaj vseh treh kraljic in označimo napadena polja. */
  for (n = 0; n < 3; n++) {
    scanf("%d %d %d", &x, &y, &z); ObdelajKraljico(x - 1, y - 1, z - 1); }
  /* Preštejmo, koliko polj napadajo vse tri kraljice. */
  for (n = 0, x = 0; x < 8; x++) for (y = 0; y < 8; y++) for (z = 0; z < 8; z++)
    if (a[x][y][z] == 3) n++;
  /* Izpišimo rezultat. */
  printf("%d\n", n); return 0;
}
```

Zelo elegantna pa je tudi naslednja rešitev. Recimo, da imamo kraljico na položaju (x_k, y_k, z_k) in da nas zanima, če ta kraljica napada polje (x, y, z). Izračunajmo absolutne vrednosti razlik: $\triangle x = |x_k - x|$, $\triangle y = |y_k - y|$ in $\triangle z = |z_k - z|$. Če pogledamo definicijo tega, katera polja kraljica napada, vidimo, da je polje napadeno natanko v naslednjih primerih: (1) če sta dva od $\triangle x$, $\triangle y$ in $\triangle z$ enaka 0 (tretji pa ima lahko v tem primeru poljubno vrednost) — to je napad v smeri, vzporedni z eno od koordinatnih osi; (2) če je eden od njih enak nič, druga dva pa imata poljubno vrednost, vendar oba enako — to je napad v smeri ene od ravninskih diagonal; (3) če imajo vsi trije enako vrednost — to je napad v smeri ene od diagonal kocke. Na podlagi tega razmisleka lahko za katerokoli polje šahovnice preprosto preverimo, ali ga posamezna kraljica napada ali ne; v spodnjem programu to počne funkcija Napadeno. Glavni del našega programa z nekaj gnezdenimi zankami sprehodi po vseh poljih in za vsako preveri, če ga napadajo vse tri kraljice.

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

bool Napadeno(int dx, int dy, int dz)
{
  int d = dx; if (d == 0) d = dy; else if (dy != 0 && dy != d) return false;
  return (d == 0 || dz == 0 || d == dz);
}
```

```
int main()
{
  int x, y, z, i, n, xk[3], yk[3], zk[3];
  /* Preberimo položaj vseh treh kraljic. */
  for (i = 0; i < 3; i++) scanf("%d %d %d", &xk[i], &yk[i], &zk[i]);
  /* Za vsako polje poglejmo, če ga napadajo vse kraljice. */
  for (n = 0, x = 1; x <= 8; x++) for (y = 1; y <= 8; y++) for (z = 1; z <= 8; z++) {
    i = 0; while (i < 3 && Napadeno(abs(x - xk[i]), abs(y - yk[i]), abs(z - zk[i]))) i++;
    if (i == 3) n++; }
  /* Izpišimo rezultat. */
  printf("%d\n", n); return 0;
}
```

3. Brisanje parov

Pri tej nalogi si je koristno pomagati s skladom. Na začetku imejmo prazen sklad. Vhodni niz berimo črko za črko od leve proti desni. ko naletimo na malo črko, jo odložimo na vrh sklada; pri veliki črki pa pogledamo, če je na vrhu sklada pripadajoča mala črka; če je, jo pobrišemo, sicer pa veliko črko dodamo na sklad. Na koncu tega postopka je vsebina sklada ravno okrajšani niz, po katerem sprašuje naloga. Oglejmo si delovanje tega postopka na primeru iz naloge: vhodni niz je torej `abbBAacCAdCcaBb`.

Prebrani znak	Stanje sklada (vrh je na desni)
a	a
b	ab
b	abb
B	ab
A	abA
a	abAa
c	abAac
C	abAa
A	abA

in tako naprej.

Ker na sklad po vsakem prebranem znaku vhodnega niza dodamo največ en znak, včasih pa s sklada tudi kaj pobrišemo, sledi, da je po k prebranih vhodnih znakih niz na skladu dolg največ k (ali pa še manj). Zato lahko naš spodnji podprogram za shranjevanje sklada uporablja kar isto tabelo niz, v kateri je dobil vhodni niz (kajti naslednji še neprebrani vhodni znak gotovo leži v tistem delu tabele, ki je sklad doslej še ni dosegel in povozil njene vsebine). V spremenljivki n vodimo podatek o trenutni dolžini sklada. Na koncu je tako v tabeli niz ravno okrajšana različica prvotnega vhodnega niza; le z ničelnim znakom ga moramo še zaključiti, pa nastane iz njega veljaven C/C++ovski niz.

```
void Okrajsaj(char *niz)
{
  int n = 0; char c, *p = niz;
  while (*p) {
    c = *p++;
    if (n > 0 && JeMalaCrka(niz[n - 1]) && c == DajVeliko(niz[n - 1])) n--;
    else niz[n++] = c; }
  niz[n] = 0;
}
```

Oglejmo si še rešitev v pythonu. Tu smo kot sklad uporabili kar običajen pythonov seznam (*list*) in ga na koncu z metodo join staknili v niz:

```
def Okrajsaj(niz):
  sklad = []
  for c in niz:
    if sklad and JeMalaCrka(sklad[-1]) and c == DajVeliko(sklad[-1]): sklad.pop()
    else: sklad.append(c)
  return "".join(sklad)
```

4. Ta5nik

Naloge se lahko lotimo na več načinov. Preprosta in za naše namene čisto dovolj dobra rešitev je, da se v zanki premikamo naprej po vhodnem nizu `s` in na vsakem mestu preverimo, ali se vsebina niza `s` od trenutnega položaja naprej ujema s kakšnim od vzorcev (nizov `nič`, `ena` in tako naprej). Če opazimo ujemanje, povečamo števec zamenjav in v izhodni niz (v spodnji rešitvi kaže spremenljivka izhod na začetek izhodnega niza, konec pa na njegov konec) zapišemo številko, s katero moramo zamenjati pravkar odkriti vzorec (npr. `nič` zamenjamo z `0` itd.). Če pa pregledamo vse vzorce, ne da bi pri katerem ugotovili, da se pojavlja na trenutnem položaju v nizu `s`, lahko trenutni znak niza `s` skopiramo v izhodni niz in se po nizu `s` premaknemo za eno mesto naprej. Ta postopek ponavljamo, dokler ne pridemo do konca niza `s`. Na koncu le še preverimo, če smo izvedli vsaj dve zamenjavi, in v tem primeru izpišemo dobljeni izhodni niz.

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

const char *vzorci[10] = { "nič", "ena", "dve", "tri", "štiri",
                           "pet", "šest", "sedem", "osem", "devet" };

void Ta5nik(const char *s)
{
  char *izhod = (char *) malloc(1 + strlen(s)), *konec; const char *S, *P;
  int stZamenjav = 0, d; konec = izhod;
  while (*s)
  {
    for (d = 0; d <= 9; d++) {
      /* Primerjajmo niz vzorci[d] z vhodnim nizom od trenutnega položaja naprej. */
      P = vzorci[d]; S = s; while (*P && *P == *S) P++, S++;
      /* Če smo prišli do konca vzorca, ne da bi opazili neujemanje,
         izvedimo zamenjavo. */
      if (! *P) { *konec++ = '0' + d; stZamenjav++; s = S; break; }}
    /* Če pri nobenem vzorcu nismo izvedli zamenjave, skopirajmo trenutni znak
       brez sprememb v izhodni niz. */
    if (d > 9) *konec++ = *s++;
  }
  /* Če smo izvedli več kot eno zamenjavo, izhodni niz izpišimo. */
  if (stZamenjav > 1) { *konec = 0; printf("%s\n", izhod); }
  free(izhod);
}
```

V gornji rešitvi smo to, ali se vzorec vzorci[d] pojavlja v trenutnem položaju vhodnega niza, preverjali z notranjo zanko **while**, ki je primerjala znake vzorca in znake niza `s`, dokler ne pride do konca vzorca ali pa zazna neujemanja. Namesto z zanko bi lahko to rešili tudi s funkcijo strncmp iz standardne knjižnice:

```
    if (0 == strncmp(s, vzorci[d], strlen(vzorci[d]))) {
      *konec++ = '0' + d; stZamenjav++; s += strlen(vzorci[d]); break; }}
```

V nekaterih programskih jezikih imamo pri roki kakšno standardno funkcijo za zamenjavo vseh pojavitev nekega podniza v danem nizu. V tem primeru nam lahko pride na misel, da bi to funkcijo preprosto klicali desetkrat: najprej bi zamenjali vse pojavitve podniza `nič` z nizom `0`, nato vse pojavitve podniza `ena` z nizom `1` in tako naprej. Težava te rešitve je, da bi na primer v nizu `petrijevka` najprej opravili zamenjavo `tri` → `3` in tako dobili `pe3jevka`, medtem ko naloga zahteva, da moramo v primeru prekrivanja vzorcev opraviti najbolj levo možno zamenjavo (`5rijevka`). Na srečo za konkretnih deset vzorcev, s katerimi delamo pri tej nalogi (nizi `nič`, `ena`, ..., `devet`) velja, da lahko pride do prekrivanja vzorcev — torej do primerov, ko se konec (sufiks) enega vzorca ujema z začetkom (prefiksom) drugega vzorca — le v naslednjih štirih primerih: `dvena`, `petri`, `šestri` in `devetri`. Vidimo torej, da bomo do najbolj leve zamenjave gotovo

prišli, če izvedemo `ena` → `1` za `dve` → `2` in če izvedemo `tri` → `3` za `pet` → `5`, `šest` → `6` in `devet` → `9`. Tako pridemo na primer do naslednje rešitve v pythonu:[1]

```
def Ta5nik(s):
  vzorci = ("nič", "ena", "dve", "tri", "štiri", "pet", "šest", "sedem", "osem", "devet")
  stZamenjav = 0
  for d in (0, 2, 4, 5, 6, 7, 8, 9, 1, 3):
    stZamenjav += s.count(vzorci[d])
    s = s.replace(vzorci[d], str(d))
  if stZamenjav >= 2: print s
```

Nekaj težav nam je povzročila še zahteva iz naloge, da moramo okrajšani niz izpisati le, če smo izvedli več kot eno zamenjavo. Metoda `replace` nam ne pove, koliko zamenjav je izvedla, zato pred zamenjavo pokličemo še `s.count(vzorci[d])`, ki nam prešteje, kolikokrat se v `s` pojavlja tisti vzorec.

5. Koalicije

Naloga zahteva, da mora biti od vsakega para sovražnih strank natanko ena v koaliciji; to nam močno omeji možno sestavo koalicij. Če sta stranki u in v sovražni in vzamemo u v koalicijo, potem vemo, da v ne smemo vzeti v koalicijo; in ker zdaj v ni v koaliciji, moramo v koalicijo vzeti vse tiste stranke, ki so sovražne stranki v; in tako naprej. Vsakič torej, ko za neko stranko odločimo, ali bo v koaliciji ali ne, vemo, da iz tega tudi za njene sovražnice enolično sledi, ali bodo morale biti v koaliciji ali zunaj nje. Pri tem si je koristno pomagati s seznamom (vrsto), v katerega dodajamo stranke, za katere smo že določili, ali bodo v koaliciji ali ne, nismo pa še pregledali njihovih sovražnic. V glavni zanki bomo jemali stranke iz vrste in pri vsaki od njih določili stanje njenih sovražnic. Ta postopek ponavljamo, dokler se nam vrsta strank, ki še čakajo na obdelavo, ne izprazni.

Med delom si bomo pomagali še s tabelo a, v kateri za vsako stranko u vodimo podatek o tem, ali je v koaliciji ($a_u = 1$) ali ne ($a_u = -1$) ali pa da zanjo še ne vemo, če bo v koaliciji ali ne ($a_u = 0$). Postopek je torej takšen:

for $u := 1$ **to** n **do** $a_u := 0$;
$Q :=$ prazna vrsta;
$a_1 := 1$; dodaj 1 v Q; (* *Recimo, da bo stranka 1 v koaliciji.* *)
while Q ni prazna:
  naj bo u poljubna stranka iz Q; pobriši u iz Q;
  za vsako v, ki je sovražna stranki u:
    if $a_v = 0$ **then**
      $a_v = -a_u$; dodaj v v vrsto Q;
    else if $a_v = a_u$ **then**
      dani vhodni primer je nerešljiv: iz dosedanjih omejitev za stranki
      u in v sledi, da bi morali biti obe v koaliciji ali pa obe zunaj nje,
      hkrati pa sta si sovražni, torej ne bomo mogli izpolniti zahteve, da je v
      koaliciji natanko ena od njiju;

Vidimo, da smo postopek začeli tako, da smo stranko 1 vzeli v koalicijo. Če bi namesto tega začeli s tem, da stranko 1 izključimo iz koalicije (postavimo $a_1 := -1$), bi se preostanek postopka odvrtel popolnoma enako, le da bi bili predznaki vseh elementov v tabeli a obrnjeni in v koalicijo bi dobili ravno tiste stranke, ki so bile prej (pri $a_1 = +1$) zunaj nje. Postopka nam torej pravzaprav ni treba izvajati še enkrat, ampak je dovolj že, če na koncu sešejemo število poslancev v koaliciji in število tistih zunaj nje ter vrnemo večjo od obeh vrednosti.

Dosedanji postopek še ni čisto pravilna rešitev naše naloge. Lahko se namreč zgodi, da ob koncu izvajanja glavne zanke za nekatere stranke še ni določil, ali so v koaliciji ali

[1] Slabost tega pristopa je, da ga ne moremo enostavno posplošiti na poljuben nabor vzorcev, saj pri nekaterih naborih vzorcev sploh ni nobenega fiksnega vrstnega reda zamenjav, ki bi nam zagotavljal, da bo vedno izvedel najbolj levo možno zamenjavo. Na primer, če bi morali izvajati zamenjavi `cdc` → `0` in `dcd` → `1`, potem bi pri nekaterih vhodnih nizih dobili pravilni rezultat tako, da bi najprej izvedli vse možne zamenjave `cdc` → `0` in nato vse `dcd` → `1`, pri nekaterih vhodnih nizih pa bi morali narediti ravno obratno (naprej vse `dcd` → `1` in nato vse `cdc` → `0`).

ne (z drugimi besedami, nekateri elementi tabele a so mogoče še vedno enaki 0). Če bi vzeli recimo primer iz besedila naloge in začeli pri stranki 1 (tako kot to stori naš gornji postopek), bi v nadaljevanju obdelali še stranke 2, 3 in 6, ne pa tudi strank 4 in 5, kajti tidve prek relacije sovražnosti nista niti posredno niti neposredno povezani s stranko 1. Relacija sovražnosti nam torej množico strank lahko razbije na več delov, ki so vsak znotraj sebe sicer povezani, niso pa povezani drug z drugim. Zgoraj opisani postopek je torej obdelal šele enega od teh delov, v nadaljevanju pa moramo z njim obdelati še ostale dele in na koncu rezultate sešteti.

for $u := 1$ **to** n **do** $a_u := 0$;
$M := 0$; (* *Velikost največje koalicije.* *)
for $w := 1$ **to** n **do**
  if $a_w \neq 0$ **then**
    continue; (* *Točko w smo nekoč že obdelali.* *)
  $Q :=$ prazna vrsta; dodaj w v vrsto Q;
  $m_1 = n_w$; $m_{-1} := 0$; (* *Št. poslancev v koaliciji in zunaj nje.* *)
  while Q ni prazna:
    naj bo u poljubna stranka iz Q; pobriši u iz Q;
    za vsako v, ki je sovražna stranki u:
      if $a_v = 0$ **then**
        $a_v = -a_u$; dodaj v v vrsto Q; $m_{a_v} := m_{a_v} + n_v$;
      else if $a_v = a_u$ **then**
        dani vhodni primer je nerešljiv;
  $M := M + \max\{m_1, m_{-1}\}$;
return M;

8. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

25. januarja 2013

NASVETI ZA MENTORJE O IZVEDBI TEKMOVANJA IN OCENJEVANJU

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). Pri reševanju si lahko tekmovalci pomagajo tudi z literaturo in/ali zapiski, ni pa mišljeno, da bi imeli med reševanjem dostop do interneta ali do kakšnih datotek, ki bi si jih pred tekmovanjem pripravili sami. Čas reševanja je omejen na 180 minut.

Nekatere naloge kot odgovor zahtevajo program ali podprogram v kakšnem konkretnem programskem jeziku, nekatere naloge pa so tipa „opiši postopek“. Pri slednjih je načeloma vseeno, v kakšni obliki je postopek opisan (naravni jezik, psevdokoda, diagram poteka, izvorna koda v kakšnem programskem jeziku, ipd.), samo da je ta opis dovolj jasen in podroben in je iz njega razvidno, da tekmovalec razume rešitev problema.

Glede tega, katere programske jezike tekmovalci uporabljajo, naše tekmovanje ne postavlja posebnih omejitev, niti pri nalogah, pri katerih je rešitev v nekaterih jezikih znatno krajša in enostavnejša kot v drugih (npr. uporaba perla ali pythona pri problemih na temo obdelave nizov).

Kjer se v tekmovalčevem odgovoru pojavlja izvorna koda, naj bo pri ocenjevanju poudarek predvsem na vsebinski pravilnosti, ne pa na sintaktični. Pri ocenjevanju na državnem tekmovanju zaradi manjkajočih podpičij in podobnih sintaktičnih napak odbijemo mogoče kvečjemu eno točko od dvajsetih; glavno vprašanje pri izvorni kodi je, ali se v njej skriva pravilen postopek za rešitev problema. Ravno tako ni nič hudega, če npr. tekmovalec v rešitvi v C-ju pozabi na začetku **#include**ati kakšnega od standardnih headerjev, ki bi jih sicer njegov program potreboval; ali pa če podprogram `main()` napiše tako, da vrača **void** namesto **int**.

Pri vsaki nalogi je možno doseči od 0 do 20 točk. Od rešitve pričakujemo predvsem to, da je pravilna (= da predlagani postopek ali podprogram vrača pravilne rezultate), poleg tega pa je zaželeno tudi, da je učinkovita (manj učinkovite rešitve dobijo manj točk).

Če tekmovalec pri neki nalogi ni uspel sestaviti cele rešitve, pač pa je prehodil vsaj del poti do nje in so v njegovem odgovoru razvidne vsaj nekatere od idej, ki jih rešitev tiste naloge potrebuje, naj vendarle dobi delež točk, ki je približno v skladu s tem, kolikšen delež rešitve je našel.

Če v besedilu naloge ni drugače navedeno, lahko tekmovalčeva rešitev vedno predpostavi, da so vhodni podatki, s katerimi dela, podani v takšni obliki in v okviru takšnih omejitev, kot jih zagotavlja naloga. Tekmovalcem torej načeloma ni treba pisati rešitev, ki bi bile odporne na razne napake v vhodnih podatkih.

V nadaljevanju podajamo še nekaj nasvetov za ocenjevanje pri posameznih nalogah.

1. Lemingi

- Naloga pravi, da če leming pri padanju zgolj oplazi krajišče ploščadi, se to ne šteje kot padec na ploščad. V naši rešitvi to na primer pomeni, da v pogoju „**if** $x_i < x$ **and** $x < x_i + d_i$“ nastopata znaka $<$ in ne $\leq$. Rešitvam, ki stik s krajiščem pomotoma štejejo kot padec na ploščad, naj se zaradi tega odbije dve točki.

- Naloga pravi, da se lemingu smer gibanja spremeni, ko prileti na ploščad. Iz tega na primer sledi, da če je ob začetku letenja obrnjen v desno ($s_0 = \texttt{D}$), se bo po prvi

ploščadi, na katero bo priletel, gibal v levo (kot vidimo tudi v primeru v besedilu naloge). Rešitvam, ki pomotoma predpostavljajo, da se leming po prvi ploščadi, na katero prileti, giblje v smeri s_0 namesto v nasprotni smeri, naj se zaradi tega odbije dve točki.

- Mišljeno je, da učinkovita rešitev pri tej nalogi porabi $O(n)$ časa, če je n število ploščadi. Rešitev, ki porabi $O(n^2)$ časa (npr. ker po vsakem padcu pregleda celoten seznam ploščadi, da ugotovi, katera je naslednja ploščad, na katero bo leming priletel), naj dobi največ 10 točk (če drugače daje pravilne rezultate).

2. 3-d šah

- V naših rešitvah smo predstavili dve različici rešitve: pri eni imamo tabelo $8 \times 8 \times 8$ celic in za vsako kraljico označimo polja, ki jih ta kraljica napada (in povečamo ustrezne elemente tabele), pri drugi pa nimamo tabele in za vsako polje zgolj z izračunom preverimo, ali ga posamezna kraljica napada ali ne. Čeprav je druga različica elegantnejša in porabi manj pomnilnika, naj se pri ocenjevanju obe vrsti rešitev šteje kot dovolj dobri. Tudi rešitev s tabelo lahko torej dobi vse točke, če daje pravilne rezultate.
- Če bi rešitev pomotoma predpostavila, da so koordinate kraljic v vhodnih podatkih podane v razponu 0..7 namesto 1..8, naj se ji zaradi tega odšteje dve točki.
- Če bi rešitev z označevanjem napadenih polj v tabeli pomotoma kakšno polje označila po večkrat pri isti kraljici (in zato pri kakšnem polju mislila, da ga napada več kraljic, kot jih v resnici), naj se ji zaradi tega odbije največ pet točk.
- Če bi rešitev z označevanjem napadenih polj v tabeli pomotoma poskušala dostopati do elementov na neveljavnih indeksih, naj se ji zaradi tega odšteje največ pet točk.

3. Brisanje parov

- Mišljeno je, da bi učinkovita rešitev pri tej nalogi porabila $O(n)$ časa. Rešitve, ki porabijo $O(n^2)$ časa, naj dobijo največ 12 točk (če so drugače pravilne).
- Nič ni narobe, če rešitev namesto funkcij JeMala in DajVeliko, ki ju omenja besedilo naloge, uporablja ekvivalentne funkcije iz standardne knjižnice svojega jezika (npr. islower in toupper v C/C++).
- Besedilo naloge pravi, da funkcija DajVeliko sproži izjemo, če ji kot parameter podamo znak, ki ni mala črka. Če rešitev ne pazi na to in včasih pokliče DajVeliko s parametrom, ki ni mala črka, naj se ji odbije tri točke.

4. Ta5nik

- Naloga pravi, da mora rešitev izpisati okrajšano različico niza le, če je pri krajšanju izvedla vsaj dve zamenjavi. Rešitvam, ki tega pogoja ne upoštevajo (in npr. okrajšano različico niza izpišejo v vsakem primeru, ali pa je nikoli ne izpišejo in jo zgolj vrnejo kot funkcijsko vrednost), naj se zaradi tega odbije štiri točke.
- Naloga pravi, da če je možno na nekem nizu izvesti več zamenjav, moramo niz okrajšati tako, kot da bi najprej izvedli najbolj levo možno zamenjavo (na primer `petrijevka` → `5rijevka` in ne `pe3jevka`). Če rešitev tega pogoja ne upošteva, naj se ji zaradi tega odbije pet točk.
- Drugače pa rešitev, ki zamenja vseh deset vzorcev v fiksnem vrstnem redu (npr. najprej zamenja vse pojavitve podniza `dve` z `2`, nato zamenja vse pojavitve podniza `ena` z `1` itd.), tudi lahko dobi vse točke, če je vrstni red izbran tako, da daje pravilne rezultate (torej `dve` → `2` pred `ena` → `1` ipd.).

5. Koalicije

- V tej nalogi se skriva problem barvanja točk grafa z dvema barvama (točke grafa predstavljajo stranke, povezave pa pare sovražnih strank), pri čemer točki ne smeta biti iste barve, če sta neposredno povezani s povezavo. Ta naloga je za šolsko tekmovanje precej težka, zato od rešitev pričakujemo predvsem opažanje, da ko določimo barvo ene točke, iz tega enolično izhaja barva njenih sosed, iz tega potem barva njihovih sosed in tako naprej. Ni pa nujno, da rešitev graf pregleduje učinkovito. Naša rešitev porabi $O(n + m)$ časa za graf z n točkami in m povezavami, dovolj dobra pa bi bila tudi rešitev s časovno zahtevnostjo $O(n^2)$ (tudi taka lahko dobi vseh 20 možnih točk, če je tudi sicer pravilna).
- Če rešitev ne upošteva, da je graf lahko sestavljen iz več ločenih komponent, ki med seboj niso povezane, in zato pobarva samo eno od njih, naj se ji odbije šest točk. Če rešitev sicer pobarva vse komponente, vendar rezultatov za posamezne komponente ne skombinira tako, da bi nastala največja možna koalicija za graf kot celoto (npr. ker ne upošteva, da mora pri vsaki komponenti od dveh možnih koalicij vzeti tisto, ki ima več poslancev), naj se ji odbije tri točke.
- Naloga pravi tudi, naj rešitev prepozna primere, ko sploh ne obstaja nobena koalicija, ki bi bila v skladu z omejitvami (do tega pride, ko je v grafu kakšen cikel lihe dolžine). Rešitvi, ki takih primerov ne zazna (in takrat npr. vrne neko neveljavno koalicijo), naj se odšteje štiri točke.

Težavnost nalog

Državno tekmovanje ACM v znanju računalništva poteka v treh težavnostnih skupinah (prva je najlažja, tretja pa najtežja); na tem šolskem tekmovanju pa je skupina ena sama, vendar naloge v njej pokrivajo razmeroma širok razpon zahtevnosti. Za občutek povejmo, s katero skupino državnega tekmovanja so po svoji težavnosti primerljive posamezne naloge letošnjega šolskega tekmovanja:

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM in IJS
1. Lemingi	lažja naloga v prvi skupini
2. 3-d šah	srednje težka naloga v prvi skupini
3. Brisanje parov	težja v prvi ali lahka naloga v drugi skupini
4. Ta5nik	težja v prvi ali lažja naloga v drugi skupini
5. Koalicije	srednje težka naloga v drugi ali lažja v tretji skupini

Če torej na primer nek tekmovalec reši le prvo nalogo in del druge, pri ostalih pa ne naredi (skoraj) ničesar, to še ne pomeni, da ni primeren za udeležbo na državnem tekmovanju; pač pa je najbrž pametno, če na državnem tekmovanju ne gre v drugo ali tretjo skupino, pač pa v prvo.

Podobno kot prejšnja leta si tudi letos želimo, da bi čim več tekmovalcev s šolskega tekmovanja prišlo tudi na državno tekmovanje in da bi bilo šolsko tekmovanje predvsem v pomoč tekmovalcem in mentorjem pri razmišljanju o tem, v kateri težavnostni skupini državnega tekmovanja naj kdo tekmuje.