16. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

22. januarja 2021

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.");
   }
}
```

16. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

22. januarja 2021

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. Križci in krožci

Dva igralca sta se igrala križce in krožce na karirasti mreži nenavadne oblike: sestavlja jo ena sama vrstica, v njej pa je n polj. Stanje mreže lahko zato opišemo z nizom n znakov, v katerem črka `'x'` predstavlja križec, črka `'o'` pa krožec. **Napiši podprogram** (funkcijo) `Izenaceno(s)`, ki kot vhodni podatek dobi niz `s` in preveri, če ta niz predstavlja takšno stanje mreže, v katerem se je igra končala z izenačenim izidom. Z drugimi besedami, preveriti je treba, če veljajo vsi naslednji pogoji:

- Na vsakem od polj mora biti ali križec ali krožec, drugih znakov v nizu ne sme biti.
- Število križcev mora biti enako številu krožcev.
- Nikjer se ne smejo pojavljati po trije (ali več) enaki znaki skupaj.

Tvoja rešitev naj bo učinkovita, tako da bo delovala tudi za velike n (dolge vhodne nize).

Nekaj primerov: niza `xxooxo` in `oxox` predstavljata izenačene izide, nizi `xxooox`, `xxcoox` in `ooxooxxo` pa ne.

2. Kovanci

Janezek rad obiskuje dedka. Ne le, da dedek ve toliko zanimivih zgodb, vsakič, ko ga obišče, se njegov hranilnik-prašiček odebeli. Zadnjič pa ga je čakalo presenečenje. Dedek mu je na mizo postavil več kupčkov kovancev in mu naročil: vzameš lahko, kolikor želiš kupčkov, samo nikoli ne smeš vzeti dveh sosednjih. Če so torej na mizi kupčki z vrednostmi

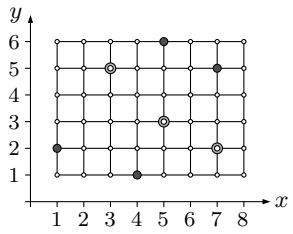
2, 4, 1, 3, 4,

lahko Janezek izbere prvi, tretji in peti kupček, lahko vzame drugega in petega, lahko drugega in četrtega itd. — možnih je še nekaj drugih kombinacij. Janezkov prašiček je seveda lačen, zato bi rad Janezek pobral z mize karseda veliko kovancev. Pomagaj mu in **opiši postopek** (ali napiši podprogram oz. funkcijo, če ti je lažje), ki za dano tabelo vrednosti kupčkov vrne največjo vsoto, ki jo je mogoče na ta način doseči. Pri zgornjem primeru je pravilni rezultat 8 (to vsoto dobimo, če poberemo drugi in peti kupček).

Pozor: čeprav je v zgornjem primeru pet kupčkov, naj tvoja rešitev deluje za poljubno število kupčkov, tudi če jih je npr. več tisoč.

3. Taksi

V nekem mestu ima cestno omrežje obliko pravokotne kariraste mreže. Položaj vsake točke (križišča) lahko zato opišemo s parom celoštevilskih koordinat (x, y), kot kaže naslednja slika:



Ker potekajo ceste samo vodoravno in navpično, je dolžina poti med dvema točkama, recimo (x_1, y_1) in (x_2, y_2), pri tem cestnem omrežju enaka $|x_1 - x_2| + |y_1 - y_2|$.

Taksist, ki vozi za n stalnih strank, bi rad na eni od točk omrežja postavil svojo centralo, pri čemer se mora odločiti med m možnimi položaji centrale. Med raziskavo profitabilnosti je že določil koordinate (x, y) vseh n strank in vseh m možnih položajev centrale. **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki ugotovi, na katerega izmed možnih položajev naj postavi centralo, da bo vsota razdalj od centrale do strank čim manjša. (Če obstaja več enako dobrih najboljših položajev, je vseeno, katerega od njih vrne tvoja rešitev.) Kot vhodne podatke tvoj postopek dobi n, m ter koordinate vseh n stalnih strank in vseh m možnih položajev centrale. Tvoj postopek naj ne predpostavi, da je mreža majhna, kot na primer tista na gornji sliki — deluje naj tudi za velike mreže.

Primer: zgornja slika kaže mrežo, na kateri so štiri stalne stranke (črne pike) in trije možni položaji centrale (dvojni krogi). Med temi tremi položaji je najboljši tisti na $(5, 3)$, pri katerem je vsota razdalj do stalnih strank enaka (če gledamo stranke od leve proti desni) $5 + 3 + 3 + 4 = 15$.

4. Preusmerjanje

Spletni strežnik ima možnost, da ko odjemalec od njega zahteva vsebino z nekega naslova (URLja), te vsebine odjemalcu ne pošlje, pač pa ga obvesti, da se ta vsebina zdaj nahaja na nekem drugem naslovu — temu pravimo z drugimi besedami, da ga je *preusmeril*. Takšne preusmeritve lahko tvorijo verige: z enega naslova nas preusmerijo na drugega, s tega na tretjega in tako naprej; v najslabšem primeru pa se lahko taka veriga preusmeritev celo zacikla (na primer: s tretjega naslova nas preusmerijo nazaj na drugega).

Napiši podprogram oz. funkcijo, ki za dani začetni naslov z sledi verigi preusmeritev in vrne naslov, pri katerem se ta veriga konča, oz. ugotovi, če se veriga zacikla. (Mogoče je seveda tudi, da se veriga konča kar pri z-ju samem, če s tega naslova ni nobene preusmeritve.) Da bo naša naloga lažja, bomo naslove namesto z nizi znakov predstavili kar z naravnimi števili od 1 do n. Kot vhodne podatke dobi tvoj podprogram števili n in z ter seznam parov (s_i, t_i), ki povedo, da obstaja z naslova s_i preusmeritev na naslov t_i. Posamezna stran lahko nastopa kot prvi element v največ enem takem paru — z drugimi besedami, ne more se zgoditi, da bi z neke strani s obstajali preusmeritvi na dve ali več drugih strani.

Primer: če imamo $n = 6$ in preusmeritve $(1, 2)$, $(2, 4)$, $(3, 1)$, $(6, 5)$, je pri $z = 1$ pravilni odgovor 4.

5. Odstranjevanje črk

Angleška beseda *splatters* ima zanimivo lastnost. Če iz nje črke brišemo v pravem vrstnem redu, bomo imeli do konca na vsakem koraku pred seboj neko angleško besedo: *splatters* → *splatter* → *platter* → *latter* → *later* → *late* → *ate* → *at* → *a*.

Napiši podprogram oz. funkcijo, ki kot vhodni podatek dobi seznam nizov in v njem poišče najdaljši niz z zgoraj opisano lastnostjo, torej najdaljši tak niz, v katerem bi se dalo enega po enega brisati znake (če na vsakem koraku primerno izberemo, kateri znak pobrišemo) tako dolgo, da bi ostal niz dolžine 1, pri čemer bi po vsakem brisanju imeli niz, ki je tudi prisoten v vhodnem seznamu. Če obstaja več enako dolgih najdaljših nizov, je vseeno, katerega od njih vrne tvoja rešitev.

Predpostavi, da so nizi sestavljeni le iz malih črk angleške abecede (od `a` do `z`) in dolgi vsaj 1 ter kvečjemu 30 znakov; nize dobiš v neki tabeli, vektorju, seznamu ali čem podobnem, torej se ti ni treba ukvarjati z branjem nizov iz datoteke. Nizov v vhodnem seznamu je lahko veliko, zato naj bo tvoja rešitev učinkovita.

16. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

22. januarja 2021

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Križci in krožci

Nalogo lahko rešimo z zanko, ki pregleduje znake niza enega po enega. Pri vsakem znaku preverimo, če je križec ali krožec; če ni nič od tega dvojega, lahko takoj zaključimo, da je niz neveljaven. Podobno tudi preverimo, če je trenutni znak slučajno enak prejšnjima dvema; če je, imamo tri zaporedne enake znake in vemo, da je niz neveljaven. Na koncu moramo preveriti še, če je število križcev enako številu krožcev; spodnja rešitev to počne tako, da med pregledovanjem niza sproti računa razliko med številom doslej prebranih križcev in številom doslej prebranih krožcev. Če je ta razlika na koncu enaka 0, je bilo križcev in krožcev enako veliko, sicer pa ne.

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

bool Izenaceno(const string &s)
{
  int razlika = 0; // Razlika v dosedanjem številu križcev in krožcev.
  char c1 = ' ', c2 = ' '; // Prejšnja dva znaka.

  // Pojdimo po znakih niza.
  for (char c : s)
  {
    // Ali so vsi znaki križci in krožci?
    if (c != 'x' && c != 'o') return false;

    // Ali so kdaj trije zaporedni znaki enaki?
    if (c == c1 && c1 == c2) return false;

    // Popravimo razliko med številom križcev in krožcev.
    razlika += (c == 'x') ? 1 : -1;

    // Prejšnja dva znaka si zapomnimo.
    c2 = c1; c1 = c;
  }

  // Na koncu še preverimo, če je križcev in krožcev enako mnogo.
  return razlika == 0;
}
```

Zapišimo to rešitev še v pythonu:

```
def Izenaceno(s):
  razlika = 0 # Razlika v dosedanjem številu križcev in krožcev.
  c1 = ' '; c2 = ' ' # Prejšnja dva znaka.

  # Pojdimo po znakih niza.
  for c in s:

    # Ali so vsi znaki križci in krožci?
    if c != 'x' and c != 'o': return False

    # Ali so kdaj trije zaporedni znaki enaki?
    if c == c1 and c1 == c2: return False

    # Popravimo razliko med številom križcev in krožcev.
    razlika += 1 if c == 'x' else -1

    # Prejšnja dva znaka si zapomnimo.
    c2 = c1; c1 = c

  # Na koncu še preverimo, če je križcev in krožcev enako mnogo.
  return razlika == 0
```

Tej rešitvi se pozna, da smo jo najprej napisali v C++ in nato prevedli v python. Zapišimo jo še bolj pythonično:

```
def Izenaceno2(s):
  krizci = s.count('x'); krozci = s.count('o')
  return krizci == krozci and krizci + krozci == len(s) and "xxx" not in s and "ooo" not in s
```

2. Kovanci

Recimo, da je na mizi n kupčkov, pri čemer na i-tem kupčku leži a_i kovancev (za $i = 1, 2, \ldots, n$). Razmišljamo lahko od konca proti začetku. Pri zadnjem kupčku imamo dve možnosti: ali ga vzamemo ali pa ne. Če zadnji kupček vzamemo, potem predzadnjega ne smemo (saj naloga pravi, da ne smemo vzeti dveh sosednjih kupčkov), kar pomeni, da nam ostane le še vprašanje, kako sestaviti čim večjo vsoto iz prvih $n - 2$ kupčkov. Po drugi strani pa, če zadnjega kupčka ne vzamemo, potem predzadnjega smemo vzeti, kar pomeni, da imamo zdaj pred seboj vprašanje, kako sestaviti čim večjo vsoto iz prvih $n - 1$ kupčkov.

V obeh primerih smo torej prišli do problema, ki je enake oblike kot prvotni, le da namesto vseh kupčkov gledamo samo prvih nekaj (natančneje prvih $n - 2$ ali $n - 1$). Ker vnaprej ne moremo vedeti, katera od obeh možnosti bo dala boljšo rešitev, moramo preizkusiti obe in uporabiti boljšo od njiju.

Tega ni težko zapisati z rekurzivno zvezo. Naj bo $f(k)$ največja vsota, ki jo lahko sestavimo z izbiranjem izmed prvih k kupčkov. Končni rezultat, po katerem sprašuje naloga, je potem $f(n)$. Dosedanji razmislek pa lahko strnemo v formulo:

$$f(k) = \max\{a_k + f(k-2), f(k-1)\}.$$

Robni primer nastopi na začetku zaporedja, kjer si lahko mislimo $f(k) = 0$ za $k \leq 0$ (če sploh ni nobenega kupčka kovancev, bo tudi naš pobrani znesek lahko zgolj 0).

Iz te formule lahko vidimo, da ko računamo $f(k)$, je koristno, če takrat že poznamo $f(k-1)$ in $f(k-2)$. Torej je pametno računati te vrednosti od leve proti desni, po naraščajočih k. Že izračunane vrednosti funkcije f bi lahko shranjevali v tabelo, vendar pravzaprav v vsakem trenutku potrebujemo le prejšnji dve vrednosti: ko računamo $f(k)$, potrebujemo $f(k-1)$ in $f(k-2)$, ne pa več tudi $f(k-3)$ in tako naprej. Zato je dovolj, če vedno hranimo le zadnji dve vrednosti funkcije.

```
#include <vector>
#include <algorithm>
using namespace std;

int NajvecjaVsota(const vector<int> &kupi)
{
  int f = 0, fPrej = 0; // Rešitvi za 0 kupov.
  // Pregledujmo kupe od leve proti desni.
  for (int kup : kupi)
  {
    // V spremenljivki f je zdaj najboljša rešitev za vse kupe do
    // vključno prejšnjega, v fPrej pa za vse kupe pred prejšnjim.
    // Izračunajmo najboljšo rešitev za vse kupe vključno s trenutnim.
    int fNova = max(fPrej + kup, f);
    // Zadnji dve rešitvi si zapomnimo.
    fPrej = f; f = fNova;
  }
  return f; // Vrnimo rešitev za vse kupe.
}
```

Zapišimo to rešitev še v pythonu:

```
def NajvecjaVsota(kupi):
  f = 0; fPrej = 0 # Rešitvi za 0 kupov.
  # Pregledujmo kupe od leve proti desni.
```

```
  for kup in kupi:
    # V spremenljivki f je zdaj najboljša rešitev za vse kupe do
    # vključno prejšnjega, v fPrej pa za vse kupe pred prejšnjim.
    # Izračunajmo najboljšo rešitev za vse kupe vključno s trenutnim.
    fNova = max(fPrej + kup, f)
    # Zadnji dve rešitvi si zapomnimo.
    fPrej = f; f = fNova
  # Vrnimo rešitev za vse kupe.
  return f
```

3. Taksi

Preprosta, vendar manj učinkovita rešitev je z dvema gnezdenima zankama. Z zunanjo zanko pojdimo po vseh možnih položajih centrale; pri vsakem položaju centrale pojdimo potem z notranjo zanko po strankah, računajmo razdaljo od centrale do posamezne stranke in te razdalje seštevajmo. Ko za trenutni položaj centrale poznamo vsoto razdalj do vseh strank, lahko pogledamo, če je to najmanjša vsota doslej, in če je, si jo zapomnimo (in tudi to, pri kateri centrali smo jo dobili). Na koncu zunanje zanke bomo tako poznali najmanjšo vsoto sploh in tudi položaj centrale, pri katerem smo jo dosegli.

Zapišimo to rešitev v C++:

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

struct Tocka { int x, y; };

// Vrne indeks izbrane centrale.
int IzberiCentralo1(const vector<Tocka>& stranke, const vector<Tocka>& centrale)
{
  int najVsota = 0, najCentrala = -1;
  // Preglejmo vse možne položaje centrale.
  for (int i = 0; i < centrale.size(); ++i)
  {
    // Izračunajmo vsoto razdalj od i-te centrale do vseh strank.
    int vsota = 0; Tocka C = centrale[i];
    for (const auto & S : stranke) vsota += abs(S.x - C.x) + abs(S.y - C.y);
    // Če je to najboljši rezultat doslej, si ga zapomnimo.
    if (najCentrala < 0 || vsota < najVsota)
      najVsota = vsota, najCentrala = i;
  }
  return najCentrala;
}
```

Zapišimo to rešitev še v pythonu. Tu ne bomo definirali svoje strukture oz. razreda za točko, pač pa bomo predpostavili, da dobimo točke kot urejene pare (pythonov tip tuple):

```
def IzberiCentralo1(stranke, centrale):
  najVsota = 0; najCentrala = -1
  # Preglejmo vse možne položaje centrale.
  for i, (cx, cy) in enumerate(centrale):
    # Izračunajmo vsoto razdalj od i-te centrale do vseh strank.
    vsota = 0
    for (sx, sy) in stranke: vsota += abs(sx - cx) + abs(sy - cy)
    # Če je to najboljši rezultat doslej, si ga zapomnimo.
    if najCentrala < 0 or vsota < najVsota:
      najVsota = vsota; najCentrala = i
  return najCentrala # Vrnimo najboljšo rešitev.
```

Ali, krajše:

```
def IzberiCentralo1b(stranke, centrale):
  return min((sum(abs(sx − cx) + abs(sy − cy) for (sx, sy) in stranke), i)
    for (i, (cx, cy)) in enumerate(centrale))[1]
```

Časovna zahtevnost te rešitve je $O(n \cdot m)$, če imamo n strank in m možnih položajev centrale.

Do hitrejše rešitve lahko pridemo, če si pomagamo z naslednjim opažanjem: pri takšni meri razdalje, kot jo uporabljamo pri naši nalogi (torej manhattanska razdalja namesto bolj znane evklidske razdalje), je tisto, kar k razdalji med dvema točkama prispeva razlika njunih x-koordinat, popolnoma neodvisno od tistega, kar k isti razdalji prispeva razlika njunih y-koordinat. Ta dva prispevka lahko računamo ločeno in ju nato seštejemo. Razmislimo, kaj to pomeni za našo nalogo. Naj bo $\mathbf{s}_i = (x_i, y_i)$ položaj i-te stranke (za $i = 1, \ldots, n$); in recimo, da razmišljamo o tem, da bi centralo postavili na točko $\mathbf{c} = (\tilde{x}, \tilde{y})$. Da ocenimo to možnost, moramo izračunati vsoto razdalj od $\mathbf{c}$ do vseh strank $\mathbf{s}_i$:

$$\begin{aligned} J(\mathbf{c}) &= \textstyle\sum_{i=1}^{n} \big(|x_i - \tilde{x}| + |y_i - \tilde{y}|\big) \\ &= \textstyle\sum_{i=1}^{n} |x_i - \tilde{x}| + \sum_{i=1}^{n} |y_i - \tilde{y}|. \end{aligned}$$

Vsoto smo torej razdelili na dve vsoti, eno za x-koordinate in eno za y-koordinate; recimo jima $J_x(\tilde{x})$ in $J_y(\tilde{y})$. Oglejmo si pobliže prvo od njiju; za drugo bo razmislek podoben. V mislih uredimo stranke naraščajoče po x-koordinati in jih v tem vrstnem redu oštevilčimo. Poglejmo zdaj, koliko jih leži levo od $\tilde{x}$, torej da zanje velja $x_i < \tilde{x}$; recimo, da je to prvih k v tem vrstnem redu, preostalih $n - k$ pa ima $x_i \geq \tilde{x}$. Videli smo, da vsaka stranka k vsoti prispeva člen $|x_i - \tilde{x}|$; pri desnih strankah je to kar enako $x_i - \tilde{x}$, pri levih pa je ta prispevek enak $\tilde{x} - x_i$. Našo vsoto lahko zdaj razdelimo na dve, eno po levih in eno po desnih strankah:

$$\begin{aligned} J_x(\tilde{x}) &= \textstyle\sum_{i=1}^{n} |x_i - \tilde{x}| \\ &= \textstyle\sum_{i=1}^{k} (\tilde{x} - x_i) + \sum_{i=k+1}^{n} (x_i - \tilde{x}). \end{aligned}$$

V vsakem seštevancu se pojavlja $\tilde{x}$, ki pa ni odvisen od i, zato ga lahko nesemo ven in ga enostavno pomnožimo s številom takih seštevancev:

$$J_x(\tilde{x}) = k\tilde{x} - \big(\textstyle\sum_{i=1}^{k} x_i\big) + \big(\sum_{i=k+1}^{n} x_i\big) - (n-k)\tilde{x}.$$

Vsoti $\sum_{i=1}^{k} x_i$ recimo S_k; to ni nič drugega kot vsota x-koordinat najbolj levih k točk. Druga vsota, $\sum_{i=k+1}^{n} x_i$, je potem enaka $S_n - S_k$. Našo formulo lahko zdaj zapišemo kot

$$\begin{aligned} J_x(\tilde{x}) &= k\tilde{x} - S_k + (S_n - S_k) - (n-k)\tilde{x} \\ &= (2k - n)\tilde{x} + S_n - 2S_k. \end{aligned}$$

Za potrebe iskanja najboljšega položaja centrale lahko v tej zadnji formuli člen S_n tudi izpustimo, saj je pri vseh $\mathbf{c}$ enak in torej nič ne vpliva na to, kateri položaj $\mathbf{c}$ bo imel najmanjšo vsoto $J(\mathbf{c})$.

Po tej formuli lahko $J_x(\tilde{x})$ računamo zelo poceni in enostavno. Ko imamo stranke enkrat urejene po x-koordinatah, lahko z bisekcijo hitro ugotovimo, koliko jih leži levo od $\tilde{x}$; tako dobimo k. Vsote S_k za vse k od 0 do n si lahko izračunamo vnaprej in jih hranimo v tabeli. Tako bomo imeli z izračunom vsake $J_x(\tilde{x})$ le $O(\log n)$ dela (zaradi bisekcije), pred tem pa še $O(n \log n)$ dela za urejanje strank po x-koordinatah. Z enakim razmislekom lahko seveda obdelamo tudi y-koordinate. Skupaj je torej časovna zahtevnost te rešitve $O((n + m) \log n)$. Zapišimo to rešitev v C++:

```
#include <algorithm>

int IzberiCentralo2(const vector<Tocka>& stranke, const vector<Tocka>& centrale)
{
  // Pripravimo urejeni tabeli x- in y-koordinat vseh strank.
  int n = stranke.size();
  vector<int> xi(n), yi(n);
  for (int i = 0; i < n; i++) xi[i] = stranke[i].x, yi[i] = stranke[i].y;
  sort(xi.begin(), xi.end()); sort(yi.begin(), yi.end());

  // Pripravimo delne vsote obeh tabel.
```

```
  vector<int> sxi(n + 1), syi(n + 1); sxi[0] = 0; syi[0] = 0;
  for (int i = 0; i < n; i++) sxi[i + 1] = sxi[i] + xi[i], syi[i + 1] = syi[i] + yi[i];
  // Preglejmo vse možne položaje centrale.
  int najVsota = 0, najCentrala = -1;
  for (int i = 0; i < centrale.size(); ++i)
  {
    // Izračunajmo vsoto razdalj od i-te centrale do vseh strank.
    Tocka C = centrale[i];
    int kx = lower_bound(xi.begin(), xi.end(), C.x) - xi.begin();
    int ky = lower_bound(yi.begin(), yi.end(), C.y) - yi.begin();
    int Jx = (2 * kx - n) * C.x - 2 * sxi[kx];
    int Jy = (2 * ky - n) * C.y - 2 * syi[ky];
    int vsota = Jx + Jy;
    // Če je to najboljši rezultat doslej, si ga zapomnimo.
    if (najCentrala < 0 || vsota < najVsota)
      najVsota = vsota, najCentrala = i;
  }
  return najCentrala;
}
```

In v pythonu:

```
import bisect

def IzberiCentralo2(stranke, centrale):
  n = len(stranke)
  # Pripravimo urejeni tabeli x- in y-koordinat vseh strank.
  xi = [x for (x, y) in stranke]; xi.sort()
  yi = [y for (x, y) in stranke]; yi.sort()
  # Pripravimo delne vsote obeh tabel.
  sxi = [0]; syi = [0]
  for x in xi: sxi.append(sxi[-1] + x)
  for y in yi: syi.append(syi[-1] + y)
  # Preglejmo vse možne položaje centrale.
  najVsota = 0; najCentrala = -1
  for i, (cx, cy) in enumerate(centrale):
    # Izračunajmo vsoto razdalj od i-te centrale do vseh strank.
    kx = bisect.bisect_left(xi, cx)
    ky = bisect.bisect_left(yi, cy)
    Jx = (2 * kx - n) * cx - 2 * sxi[kx]
    Jy = (2 * ky - n) * cy - 2 * syi[ky]
    vsota = Jx + Jy
    # Če je to najboljši rezultat doslej, si ga zapomnimo.
    if najCentrala < 0 or vsota < najVsota:
      najVsota = vsota; najCentrala = i
  return najCentrala # Vrnimo najboljšo rešitev.
```

Še ena možnost je, da možne položaje central uredimo po x-koordinati in nato istočasno pregledujemo po naraščajočih x-koordinatah tako seznam strank kot seznam položajev centrale. S tem postopkom, torej neke vrste zlivanjem dveh urejenih seznamov, bomo lahko za vsak položaj centrale določili, koliko strank je levo od nje, od tam naprej pa lahko razmišljamo enako kot zgoraj, da izračunamo J_x tega položaja centrale. Nato podobno naredimo še za y-koordinate. Porabimo torej $O(n \log n)$ časa za urejanje strank, $O(m \log m)$ časa za urejanje central in nato $O(n + m)$ za zlivanje obeh seznamov ter izračun vseh $J_x(\tilde{x})$ in $J_y(\tilde{y})$. Časovna zahtevnost te rešitve je $O(n \log n + m \log m)$, kar je bolje od prejšnje, če je $m < n$.

```
int IzberiCentralo2b(const vector<Tocka>& stranke, const vector<Tocka>& centrale)
{
  int n = stranke.size(), m = centrale.size();
```

```
  vector<int> vsote(m, 0);
  for (int os = 0; os < 2; os++) // os 0 = x, os 1 = y
  {
    // Uredimo stranke po koordinati.
    vector<int> ti(n);
    for (int i = 0; i < n; i++) ti[i] = (os == 0) ? stranke[i].x : stranke[i].y;
    sort(ti.begin(), ti.end());

    // Uredimo centrale po koordinati.
    vector<pair<int, int>> ci(m);
    for (int i = 0; i < m; i++) ci[i] = {(os == 0) ? centrale[i].x : centrale[i].y, i};
    sort(ci.begin(), ci.end());

    // Za vsako centralo izračunajmo vsoto razdalj do strank v trenutni smeri.
    int k = 0, sk = 0;
    for (auto [tc, j] : ci[i]) // centrale[j] ima koordinato tc
    {
      // Premaknimo se mimo strank, ki imajo manjšo koordinato kot trenutna centrala.
      while (k < n && ti[k] < tc) sk += ti[k++];

      // Zdaj so ti[0..k - 1] manjši od tc (in sk je njihova vsota),
      // ti[k..n - 1] pa so večji ali enaki tc.
      // Prištejmo prispevek razdalj v trenutni smeri k oceni j-tega
      // možnega položaja centrale.
      vsote[j] += (2 * k - n) * tc - 2 * sk;
    }
  }
  // Poiščimo najboljši položaj centrale.
  int najVsota = 0, najCentrala = -1;
  for (int i = 0; i < m; ++i)
    if (najCentrala < 0 || vsote[i] < najVsota) najVsota = vsote[i], najCentrala = i;
  return najCentrala;
}
```

Zapišimo to rešitev še v pythonu:

```
def IzberiCentralo2b(stranke, centrale):
  n = len(stranke); m = len(centrale)
  vsote = [0] * m

  for os in range(2): # os 0 = x, os 1 = y

    # Uredimo stranke po koordinati.
    ti = [s[os] for s in stranke]; ti.sort()

    # Uredimo centrale po koordinati.
    ci = [(c[os], i) for i, c in enumerate(centrale)]; ci.sort()

    # Za vsako centralo izračunajmo vsoto razdalj do strank v trenutni smeri.
    k = 0; sk = 0
    for (tc, j) in ci: # centrale[j][os] == tc

      # Premaknimo se mimo strank, ki imajo manjšo koordinato kot trenutna centrala.
      while k < n and ti[k] < tc: sk += ti[k]; k += 1
      # Zdaj so ti[0..k - 1] manjši od tc (in sk je njihova vsota),
      # ti[k..n - 1] pa so večji ali enaki tc.
      # Prištejmo prispevek razdalj v trenutni smeri k oceni j-tega
      # možnega položaja centrale.
      vsote[j] += (2 * k - n) * tc - 2 * sk

  # Poiščimo najboljši položaj centrale in ga vrnimo.
  najVsota = 0; najCentrala = -1
  for i in range(m):
    if najCentrala < 0 or vsote[i] < najVsota: najVsota = vsote[i]; najCentrala = i
  return najCentrala
```

4. Preusmerjanje

Ko sledimo verigi preusmeritev, je koristno, če znamo za trenutno stran hitro ugotoviti, ali obstaja z nje preusmeritev in kam. Naloga pravi, da kot vhod dobimo seznam

parov (s_i, t_i), kar za ta namen ni najbolj prikladno — morali bi se sprehoditi po celem seznamu in za vsak par preverjati, ali je njegova s_i ravno tista stran, s katero se trenutno ukvarjamo. Bolje bo, če ta seznam predelamo v tabelo: naloga pravi, da so strani oštevilčene od 1 do n, zato imamo lahko tabelo z n elementi, v kateri številko strani uporabimo kot indeks; vrednost posameznega elementa pa naj nam pove, kam nas tista stran preusmeri (če nikamor, lahko tja zapišemo na primer -1).

Zdaj znamo hitro in preprosto slediti verigi preusmeritev; paziti pa moramo še na možnost, da se veriga zacikla. Ena možnost, kako odkriti tak cikel, je, da si nekje shranjujemo podatke o tem, katere strani smo med našim sledenjem verigi že obiskali. Preden sledimo preusmeritvi na neko novo stran, moramo potem preveriti, ali smo tisto stran kdaj prej že obiskali; če da, potem vemo, da smo se znašli na ciklu. Tudi za podatke obiskanosti lahko uporabimo tabelo n logičnih vrednosti, ki za vsako stran povedo, ali smo jo že obiskali ali ne.

Druga možnost je, da se ne zmenimo za to, ali smo neko stran že obiskali ali ne, pač pa štejemo, koliko korakov smo naredili pri sledenju verigi. Če naredimo n korakov, ne da bi se veriga končala, potem vemo, da se vsaj ena stran na verigi pojavi več kot enkrat (ker bi drugače morali imeti $n+1$ različnih strani, v resnici pa jih je samo n), torej na verigi obstaja cikel. Ta rešitev porabi manj pomnilnika kot prejšnja (ker ne potrebuje tabele s podatki o obiskanosti), pač pa več časa (ker naredi n korakov, četudi se cikel mogoče pojavi že veliko prej).

Oglejmo si implementacijo prve od teh dveh možnosti v C++:

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

struct Preusmeritev { int s, na; };

int KonecVerige(int n, int z, const vector<Preusmeritev> &preusmeritve)
{
  // Pripravimo si tabelo, ki za vsako stran pove, kam nas
  // od tam preusmerijo; če nikamor, bo tam −1.
  vector<int> kam(n + 1, −1);
  for (const auto &p : preusmeritve) kam[p.s] = p.na;

  // Pripravimo si tabelo za označevanje že obiskanih strani.
  vector<bool> obiskana(n + 1, false);

  // Sledimo verigi preusmeritev od z naprej.
  while (kam[z] >= 0 && ! obiskana[z])
  {
    obiskana[z] = true; // Označimo trenutno stran za obiskano
    z = kam[z];         // in se premaknimo na naslednjo.
  }

  // Če smo se ustavili zato, ker smo prišli na neko že obiskano stran,
  // to pomeni, da se je veriga zaciklala; sicer pa vrnimo stran, pri kateri
  // se je veriga končala.
  return obiskana[z] ? −1 : z;
}
```

Vektorja kam in obiskana smo inicializirali na velikost $n+1$ elementov, da lahko kot indekse uporabljamo števila od 1 do n, kot se pojavljajo v vhodnih podatkih — elementov na indeksu 0 tako nikoli ne uporabimo. Če nas ta majhna potrata pomnilnika moti, bi morali pač paziti na to, da indekse v vhodnih podatkih pred obdelavo zmanjšamo za 1, na koncu pa številko strani, pri kateri se je veriga končala, spet povečamo za 1, preden jo vrnemo.

Namesto tabel lahko za kam in obiskana uporabimo razpršeni tabeli; s tem lahko potencialno prihranimo nekaj časa in pomnilnika, kajti v zgornji rešitvi imata obe tabeli vedno po n elementov, četudi je preusmeritev mogoče malo in četudi mogoče obiščemo le majhno število strani; pri razpršeni tabeli pa ima lahko kam le toliko elementov, kolikor je preusmeritev v vhodnem seznamu, obiskana pa le toliko elementov, kolikor strani obiščemo pri sledenju verigi. V C++ si lahko pomagamo z razredoma unordered_map in unordered_set. Še ena prednost tega pristopa je, da ga lahko skoraj brez sprememb

uporabimo tudi, če so strani namesto z zaporednimi številkami od 1 do n predstavljene z nizi (npr. z naslovi, torej URLji). Oglejmo si še to rešitev:

```
#include <unordered_set>
#include <unordered_map>

int KonecVerige2(int n, int z, const vector<Preusmeritev> &preusmeritve)
{
  // Pripravimo si slovar preusmeritev.
  unordered_map<int, int> kam;
  for (const auto &p : preusmeritve) kam.emplace(p.s, p.na);

  // Pripravimo si množico že obiskanih strani.
  unordered_set<int> obiskana;

  // Sledimo verigi preusmeritev od z naprej.
  while (obiskana.find(z) == obiskana.end())
  {
    // Poglejmo, ali obstaja s strani „z“ preusmeritev kam drugam.
    auto it = kam.find(z);

    // Če ne obstaja, se veriga tu konča.
    if (it == kam.end()) return z;

    obiskana.emplace(z); // Označimo trenutno stran za obiskano
    z = it->second;      // in se premaknimo na naslednjo.
  }
  // Tu vemo, da se je veriga zaciklala.
  return -1;
}
```

Zapišimo obe različici rešitve še v pythonu. Tu predpostavimo, da so v vhodnih podatkih predstavitve kar urejeni pari (tip tuple v pythonu):

```
def KonecVerige(n, z, preusmeritve):
  # Pripravimo si tabelo, ki za vsako stran pove, kam nas
  # od tam preusmerijo; če nikamor, bo tam -1.
  kam = [-1] * (n + 1)
  for (s, na) in preusmeritve: kam[s] = na

  # Pripravimo si tabelo za označevanje že obiskanih strani.
  obiskana = [False] * (n + 1)

  # Sledimo verigi preusmeritev od z naprej.
  while kam[z] >= 0 and not obiskana[z]:
    obiskana[z] = True # Označimo trenutno stran za obiskano
    z = kam[z]         # in se premaknimo na naslednjo.

  # Če smo se ustavili zato, ker smo prišli na neko že obiskano stran,
  # to pomeni, da se je veriga zaciklala; sicer pa vrnimo stran, pri kateri
  # se je veriga končala.
  return -1 if obiskana[z] else z

def KonecVerige2(n, z, preusmeritve):
  # Pripravimo si slovar preusmeritev.
  kam = {s: na for (s, na) in preusmeritve}

  # Pripravimo si množico že obiskanih strani.
  obiskana = set()

  # Sledimo verigi preusmeritev od z naprej.
  while z not in obiskana:

    # Poglejmo, ali obstaja s strani „z“ preusmeritev kam drugam.
    # Če ne obstaja, se veriga tu konča.
    if z not in kam: return z

    obiskana.add(z); # Sicer označimo trenutno stran za obiskano
    z = kam[z]       # in se premaknimo na naslednjo.

  # Tu vemo, da se je veriga zaciklala.
  return -1
```

5. Odstranjevanje črk

Besedam, ki imajo v nalogi opisano lastnost, bomo rekli, da so *ugodne*. Iz definicije v besedilu naloge vidimo, da je beseda ugodna, če je dolga eno samo črko ali pa če lahko iz nje z brisanjem ene črke dobimo kakšno drugo ugodno besedo; sicer pa ni ugodna. Z brisanjem ene črke seveda nastane malo krajša beseda; ko se na primer ukvarjamo z neko besedo dolžine k znakov, bomo z brisanjem dobili besede dolžine $k-1$ znakov. Koristno bo torej, če bomo takrat za tiste krajše besede že vedeli, ali so ugodne ali ne, saj je od tega potem odvisno, ali je tudi daljša beseda ugodna. Naš vhodni seznam besed je torej pametno za začetek urediti naraščajoče po dolžini in jih potem pregledovati v tem vrstnem redu.

Pri vsaki besedi moramo na vse možne načine pobrisati eno črko in preveriti, če je kakšna od tako dobljenih krajših besed že znana kot ugodna. V ta namen je koristno, če ugodne besede shranjujemo v razpršeno tabelo ali kakšno podobno podatkovno strukturo, pri kateri bomo lahko poceni preverili, ali vsebuje neko besedo ali ne. V C++ lahko uporabimo na primer razred unordered_set, v pythonu pa set.

```
#include <string>
#include <unordered_set>
#include <vector>
#include <algorithm>
using namespace std;

string Splatters(vector<string> besede)
{
  // Uredimo besede po naraščajoči dolžini.
  sort(besede.begin(), besede.end(), [] (const string &s, const string& t) {
    return s.length() < t.length(); });
  // Pripravimo razpršeno tabelo, v katero bomo shranjevali ugodne besede.
  unordered_set<string> ugodne;
  int n = besede.size(), zadnjaUgodna = -1;
  // Preglejmo besede po naraščajoči dolžini.
  for (int i = 0; i < n; i++)
  {
    const string &s = besede[i];
    // Enočrkovne besede so ugodne že same po sebi.
    int k = s.length();
    if (k <= 1) { ugodne.emplace(s); continue; }
    // Pri daljših besedah poglejmo, če lahko z brisanjem ene črke dobimo ugodno besedo.
    for (int j = 0; j < k; j++)
    {
      string t = s.substr(0, j) + s.substr(j + 1);
      // t je beseda, ki nastane, če v s pobrišemo črko s[j].
      if (ugodne.find(t) == ugodne.end()) continue;
      // t je ugodna, torej je s tudi.
      ugodne.emplace(s); zadnjaUgodna = i; break;
    }
  }
  // Vrnimo zadnjo ugodno besedo, ki smo jo našli (ta je tudi najdaljša).
  return zadnjaUgodna < 0 ? string() : besede[zadnjaUgodna];
}
```

Oglejmo si še rešitev v pythonu:

```
def Splatters(besede):
  ugodne = set(); zadnjaUgodna = None
  # Pregledujmo besede po naraščajoči dolžini.
  for k, s in sorted((len(s), s) for s in besede):
    # Enočrkovne besede so ugodne že same po sebi.
    if k <= 1: ugodne.add(s); continue
```

```
    # Pri daljših besedah poglejmo, če lahko z brisanjem ene črke dobimo ugodno besedo.
    for j in range(k):
        # Poglejmo, ali je beseda, ki nastane iz s z brisanjem črke s[j], ugodna.
        if s[:j] + s[j + 1:] not in ugodne: continue
        # Če da, potem je tudi s ugodna.
        ugodne.add(s); zadnjaUgodna = s; break
# Vrnimo zadnjo ugodno besedo, ki smo jo našli (ta je tudi najdaljša).
return zadnjaUgodna
```

16. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

22. januarja 2021

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.

Če oblika vhodnih podatkov ni natančno določena, si lahko podrobnosti tekmovalec izbere sam. Na primer, če naloga pravi, da dobimo seznam parov, je to lahko v praksi tabela (*array*), vektor, *linked list* ali še kaj drugega, pari pa so lahko bodisi strukture, ki jih je deklarirala tekmovalčeva rešitev, ali pa kaj iz standardne knjižnice (kot je `pair` v C++ ali `tuple` v pythonu).

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

1. Križci in krožci

- Neučinkovite rešitve, ki bi za niz dolžine n porabile več kot $O(n)$ časa, naj dobijo največ 13 točk, če so drugače pravilne.

- V naši rešitvi smo za preverjanje tega, ali je število križcev v nizu enako številu krožcev, uporabili eno spremenljivko, ki računa razliko med obema številoma. Za enako dobro naj šteje tudi rešitev, ki šteje križce posebej in krožce posebej, vsake v svoji spremenljivki, in ju na koncu primerja med seboj.

- V naši rešitvi smo si pri pregledovanju niza zapomnili prejšnja dva znaka v spremenljivkah, za enako dobro pa naj šteje tudi rešitev, ki vsakič znova pogleda v prejšnja dva elementa niza (ali pa enega prejšnjega in enega naslednjega ipd.). Če pride na začetku ali koncu niza do kakšnih napak pri indeksih (npr. v C++ branje znaka na indeksu -1 ali kaj podobnega), naj se rešitvi zaradi tega odšteje dve točki.

- Trije pogoji, navedeni v besedilu naloge, so mišljeni kot približno enakovredni in preverjanje vsakega od njih je vredno eno tretjino točk.

- Naloga ne predpisuje posebej, kaj naj funkcija naredi z rezultatom; lahko ga vrne (npr. kot logično vrednost tipa **bool** ali kaj podobnega), lahko ga izpiše na zaslon ipd. — vse te možnosti naj štejejo za enako dobre, glavno je, da je iz rešitve razvidno, da je do (pravilnega) rezultata v nekem trenutku res prišla.

2. Kovanci

- Besedilo naloge pravi, da je kupčkov lahko veliko. Rešitve, ki imajo eksponentno časovno zahtevnost, npr. ker delajo z rekurzijo in si ne shranjujejo že izračunanih rezultatov, naj dobijo največ 10 točk, če so sicer pravilne.

- Naša rešitev pri računu funkcije f poleg trenutne vrednosti hrani le dve prejšnji, tako da porabi le $O(1)$ pomnilnika. Za enako dobre naj se štejejo tudi rešitve, ki porabijo $O(n)$ pomnilnika, ker shranjujejo vse že izračunane rezultate, četudi večine izmed njih ne bodo več potrebovale.

3. Taksi

- Pri tej nalogi pričakujemo večinoma rešitve s časovno zahtevnostjo $O(n \cdot m)$; take naj dobijo največ 12 točk, če so sicer pravilne. Rešitve z manjšo časovno zahtevnostjo, kot sta na primer rešitvi v času $O(n \log n + m \log m)$ ali $O((n + m) \log n)$, naj dobijo vse točke, če so sicer pravilne.

- Naloga pravi, da je mreža lahko velika. Če bi kakšna rešitev slučajno imela časovno zahtevnost, ki je odvisna od velikosti mreže (za kar sicer ni kakšnega posebej dobrega razloga) namesto le od n in m, naj dobi največ 6 točk.

- Besedilo naloge ne daje nobenih zagotovil glede tega, da so vsi možni položaji central v vhodnih podatkih različni, toda če tekmovalčeva rešitev predpostavi, da so različni, naj se ji zaradi tega ne odšteva točk (če je drugače pravilna).

- Lahko se zgodi, da ležita kakšna stranka in kakšen od možnih položajev centrale na isti točki (x, y); ali pa, da leži na isti točki več strank. Če bi kakšna rešitev predpostavila, da se to ne more zgoditi, in bi zato v takih primerih vračala napačne rezultate, naj se ji zaradi tega odšteje dve točki.

4. Preusmerjanje

- Rešitve, ki porabijo $O(n^2)$ časa, npr. ker na vsakem koraku znova pregledajo celoten vhodni seznam, da ugotovijo, ali s trenutne strani vodi kakšna preusmeritev (in kam), naj dobijo največ 10 točk, če so drugače pravilne.
- V naših rešitvah smo objavili dve različici, eno s tabelo (oz. vektorjem) in eno z razpršeno tabelo (oz. slovarjem). Pri točkovanju naj se oboje šteje za enako dobro, ravno tako tudi rešitve, ki bi namesto razpršenih tabel uporabile kakšne uravnotežene drevesaste strukture (npr. `map` in `set` iz C++ove standardne knjižnice).
- Rešitvam, ki ne shranjujejo podatkov o že obiskanih straneh, pač pa v primeru cikla vedno naredijo n ali več korakov, preden ugotovijo, da obstaja cikel, naj se zaradi te neučinkovitosti odšteje dve točki.
- Če rešitev sploh ne ugotovi obstoja cikla, pač pa se na njem tudi sama zacikla v neskončni zanki, naj se ji zaradi tega odšteje polovico točk, kolikor bi jih sicer dobila glede na časovno zahtevnost postopka.
- Za morebitne drobne napake pri indeksiranju (npr. povezane s tem, ali so številke strani od 1 do n ali od 0 do $n-1$) naj se odšteje največ dve točki.
- Naloga ne predpisuje podrobnosti tega, v kakšni obliki naš podprogram dobi vhodne podatke, zato lahko tekmovalčeva rešitev podrobnosti tega določi sama. V naših primerih smo na primer enkrat imeli vektor struktur, enkrat smo uporabili pythonov tip `tuple`, dalo bi se imeti tudi dva seznama (recimo `s` in `na`, ki bi nam povedala, da obstaja preusmeritev s strani `s[i]` na stran `na[i]`) in podobno.

5. Odstranjevanje črk

- Pri tej nalogi je poudarek predvsem na opažanju, da je koristno besede pregledovati od krajših proti daljšim. Rešitve, ki tega ne počno in poskušajo mogoče z rekurzijo na vse možne načine brisati znake enega za drugim ter imajo zaradi tega eksponentno časovno zahtevnost, naj dobijo največ 5 točk, če so drugače pravilne.
- Naša rešitev shranjuje ugodne besede v razpršeno tabelo; za enako dobro naj velja tudi, če tekmovalčeva rešitev uporabi kakšno uravnoteženo drevesasto strukturo (npr. razred `set` iz C++ove standardne knjižnice). Če pa bi kakšna rešitev hranila ugodne besede v navadnem seznamu ali tabeli ali čem podobnem, kjer bi torej vsako preverjanje, ali je neka beseda ugodna, trajalo $O(n)$ časa, naj se ji zaradi tega odšteje 4 točke.
- Naša rešitev porabi pri nizu s dolžine k načeloma $O(k^2)$ časa, da sestavi vse možne nize, ki nastanejo iz s z brisanjem ene črke. To bi se dalo z nekaj pazljivosti zmanjšati na $O(k)$, enako tudi pri poizvedbah v razpršeno tabelo, če bi npr. uporabili Rabin-Karpove hash kode; vendar ni mišljeno, da bi se rešitve tekmovalcev ukvarjale s čim takim.
- Lahko se zgodi, da v vhodnem seznamu ni nobene ugodne besede. Naloga ne predpisuje natančno, kaj naj rešitev naredi v takem primeru, zato naj se za pravilno šteje karkšno koli obnašanje, iz katerega je razvidno, da rešitev ni našla ugodne besede. Lahko na primer vrne prazen niz, vrednost `None` (v pythonu) ali **`null`** (v javi ipd.), indeks -1 (če vrača indeks niza v vhodnem seznamu), lahko izpiše, da niza ni, lahko sproži izjemo in podobno.
- V naši rešitvi v C++ smo vektor `besede` prenašali po vrednosti namesto po referenci, tako da dobimo svojo kopijo vektorja in lahko besede v njem uredimo (naraščajoče po dolžini), ne da bi se to pri klicatelju kaj poznalo. Rešitvam, ki namesto tega prenašajo vektor po referenci in ga morebiti spremenijo tako, da to spremembo vidi tudi klicatelj, naj se tega ne šteje v slabo in naj se jim zaradi tega ne odšteva točk.

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
1. Križci in krožci	lažja naloga v prvi skupini
2. Kovanci	srednje težka naloga v prvi ali lažja v drugi skupini
3. Taksi	težka naloga v prvi ali srednje težka v drugi skupini[1]
4. Preusmerjanje	srednje težka naloga v prvi ali lažja v drugi skupini
5. Odstranjevanje črk	težja naloga v prvi ali lažja v drugi skupini

Če torej na primer nek tekmovalec reši le eno ali dve lažji nalogi, 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.

Zadnja leta na državnem tekmovanju opažamo, da je v prvi skupini izrazito veliko tekmovalcev v primerjavi z drugo in tretjo, med njimi pa je tudi veliko takih z zelo dobrimi rezultati, ki bi prav lahko tekmovali tudi v kakšni težji skupini. Mentorjem zato priporočamo, naj tekmovalce, če se jim zdi to primerno, spodbudijo k udeležbi v zahtevnejših skupinah.

[1] Opomba: težavnost te naloge je zelo odvisna od tega, kako točkujemo rešitve odvisno od njihove učinkovitosti. Rešitev s časovno zahtevnostjo $O(nm)$ je zelo preprosta, do učinkovitejših rešitev pa je znatno težje priti. Ker smo predvideli 12 točk (od dvajsetih) že za rešitev v času $O(nm)$, lahko štejemo to nalogo za srednje težko.