19. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

26. januarja 2024

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.");
   }
}
```

19. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

26. januarja 2024

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 `https://rtk.ijs.si/`.

1. Podatkovni center

V podatkovnem centru se nahajajo omare s strežniki, ki so oštevilčene od 1 do n. Na vratih v center, ki so označena z 0, in vratih za vsako omaro je nameščen senzor, ki sporoča varnostnemu sistemu, kdaj so bila vrata odprta ali zaprta. Varnostni sistem je treba dopolniti tako, da javi napako, če:

- se bodo odprta vrata odprla ali zaprta vrata zaprla,
- je katera od omar odprta, ko se vhodna vrata zaprejo.

Ko se vhodna vrata v center zaprejo, je na sistem poslana datoteka s številom vrat v centru (vključuje vhodna vrata) in vsakim dogodkom, ki so ga senzorji zaznali. Vsak dogodek je v svoji vrstici in je opisan z dvema številoma (ločenima s presledkom): številko vrat (od 0 do n, pri čemer 0 pomeni vhodna vrata, ostale pa so omare) in številom 0 ali 1, ki pove, ali so se vrata odprla (1) ali zaprla (0).

Napiši program, ki bo preveril podatke v datoteki in izpisal sporočilo ob nepravilnem dogodku. Če se vhodna vrata zaprejo brez napake, mora program izpisati številke vseh omar, ki so bile pred tem vsaj enkrat odprte. Tvoj program lahko bere podatke s standardnega vhoda ali pa iz datoteke `vhod.txt` (karkoli ti je lažje). Predpostaviš lahko, da je $n \leq 100$, da so bila pred prvim dogodkom v datoteki vsa vrata zaprta in da se zapiranje vhodnih vrat pojavi v zadnji vrstici datoteke, prej pa ne.

Primer datoteke (komentarji na desni niso del datoteke):

Datoteka	Komentar
`4`	število vrat ($n+1$)
`0 1`	vhodna vrata se odprejo
`1 1`	omara 1 se odpre
`3 1`	omara 3 se odpre
`1 0`	omara 1 se zapre
`3 0`	omara 3 se zapre
`2 1`	omara 2 se odpre
`2 0`	omara 2 se zapre
`0 0`	vhodna vrata se zaprejo

Pri tej vhodni datoteki bi moral program ugotoviti, da ni bilo napak, vsaj enkrat odprte pa so bile omare 1, 2 in 3.

2. Plesalci

Na odru je n plesalcev, ki plešejo ples v vrsti. Mesta, kjer stojijo pred nastopom, so oštevilčena s števili od 1 do n. Ples je razdeljen na m dejanj, vsako dejanje pa lahko opišemo s seznamom, kjer i-to število v njem pove, na katero mesto se v tem dejanju premakne oseba, ki je ob začetku dejanja stala na mestu i. Ker so plesalci izkušeni, lahko celotno dejanje izvedejo vsi naenkrat.

Po plesu plesalci obdržijo vrstni red in se pripravijo na ponovno izvedbo plesa pred novo množico ljudi. **Napiši program** (ali podprogram oz. funkcijo), ki izračuna, kolikokrat morajo izvesti ples, da se bo (ob koncu zadnje izvedbe celotnega plesa) vrstni red povrnil na prvotnega. Predpostaviš lahko, da sta m in n manjša od 1000 in da tudi potrebno število izvajanj plesa ne bo večje od 1000. Zaželeno je, da je tvoj postopek kolikor toliko učinkovit. Podrobnosti tega, v kakšni obliki tvoj (pod)program dobi ali prebere vhodne podatke, si izberi sam in jih v svoji rešitvi tudi opiši.

Primer: recimo, da imamo $n = 5$ plesalcev in $m = 2$ dejanji, opisani s seznamoma $[3, 4, 1, 5, 2]$ in $[4, 3, 2, 1, 5]$. Potem plesalec, ki je bil ob začetku plesa na mestu 1, pride po prvem dejanju na mesto 3, od tam pa v drugem dejanju na mesto 2. Ob drugi izvedbi plesa pride ta plesalec v prvem dejanju z mesta 2 na mesto 4, v drugem dejanju pa od tam na mesto 1. Podobno bi se dalo razmišljati še naprej in tudi za druge plesalce. Pri tem konkretnem primeru se izkaže, da je treba ples izvesti šestkrat, preden pridejo vsi plesalci nazaj na svoj začetni položaj.

3. Nizi

Napiši podprogram (oz. funkcijo) `NastejNize(s, k)`, ki kot parametra dobi niz s in celo število k. Predpostaviš lahko, da bo niz s sestavljen le iz malih črk angleške abecede, pri čemer se nobena ne pojavi v s več kot enkrat. Če dolžino s-ja označimo z n, bo veljalo $1 \leq n \leq k \leq 30$. Tvoj podprogram naj izpiše vse take nize t, ki so dolgi natanko k znakov, vsi ti znaki so iz niza s in vsak znak niza s se pojavi vsaj enkrat v nizu t. Nize lahko izpišeš v poljubnem vrstnem redu (vsakega natanko enkrat), zaželeno pa je, da je tvoja rešitev učinkovita in ne generira nizov po nepotrebnem.

Primer: če dobimo $s =$ `ga` in $k = 3$, moramo izpisati šest nizov: `aag`, `aga`, `gaa`, `agg`, `gag`, `gga` (ne nujno v tem vrstnem redu).

4. Zavarovanje

Pošta želi pridobiti zaupanje javnosti in se zato odloči, da bo pošiljke zavarovala in v primeru, ko jih ne dostavi pravočasno, pošiljateljem izplačala odškodnino.

Želijo pa zavarovati tudi manj vredne pošiljke, zato se odločijo nastaviti neko minimalno vrednost odškodnine o, ki bo v primeru prepozne dostave izplačana ne glede na vrednost pošiljke. Pošta torej v primeru prepozne dostave pošiljke izplača maksimum minimalne vrednosti odškodnine in vrednosti pošiljke.

Pošto zanima, kako izbira minimalne odškodnine o vpliva na znesek, ki ga morajo v najslabšem primeru (če izgubijo vso pošto v sistemu) izplačati pošiljateljem. **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki izračuna ta znesek za različne vrednosti o. Kot vhodne podatke tvoj postopek dobi: število pošiljk v sistemu (n); vrednosti teh pošiljk ($v_1, \ldots, v_n$); število minimalnih odškodnin, o katerih pošta razmišlja (m); in višine teh minimalnih odškodnin ($o_1, \ldots, o_m$). Tvoj postopek naj za vsako o_i (za $i = 1, \ldots, m$) izračuna skupno vrednost plačane odškodnine, če se izgubi vseh n pošiljk. Zaželeno je, da je tvoj postopek čim učinkovitejši.

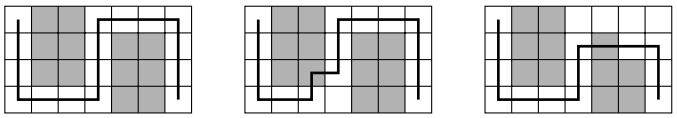
Primer: če je $o = 10$ in imamo tri pošiljke z vrednostmi 1, 20 in 25, za odškodnino velja:

- odškodnina 1. pošiljke je 10, saj je vrednost pošiljke 1, kar je manjše od minimalne odškodnine;
- odškodnina 2. pošiljke je 20, saj je vrednost pošiljke 20, kar je večje od minimalne odškodnine;
- odškodnina 3. pošiljke je 25, saj je vrednost pošiljke 25, kar je večje od minimalne odškodnine.

V najslabšem primeru (če izgubi vse pošiljke) mora torej pošta plačati $10 + 20 + 25 = 55$ enot denarja.

5. Najkrajše poti

Dana je pravokotna karirasta mreža, ki ima h vrstic in w stolpcev. Nekatera polja na mreži so prehodna, na drugih pa so ovire (in so zato neprehodna). Po čim krajši poti želimo priti od zgornjega levega do spodnjega desnega kota mreže s premiki gor, dol, levo in desno. Polji v zgornjem levem in spodnjem desnem kotu sta zagotovo prehodni. Na poti lahko eno oviro odstranimo, da si morebiti skrajšamo razdaljo (ni pa to obvezno). **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki kot vhodni podatek dobi opis mreže in izračuna dolžino najkrajše take poti (ali pa ugotovi, da taka pot sploh ne obstaja). Podrobnosti tega, kako je mreža opisana, si izberi sam in jih v svoji rešitvi tudi opiši.



Primer: gornje slike kažejo tri primere mrež, pri čemer beli kvadratki predstavljajo prehodna polja, sivi pa neprehodna. Debela črta kaže eno od možnih najkrajših poti; kjer prečka sivo polje, to pomeni, da oviro na njem odstranimo. Na levi in srednji sliki je obakrat ista mreža, pri kateri obstaja več enako dobrih najkrajših poti (dolgih po 15 korakov); tu sta prikazani dve izmed njih. Na desni sliki je drugačna mreža, pri kateri je najkrajša pot dolga 13 korakov (tu prikazana pot je tudi edina te dolžine).

19. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

26. januarja 2024

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Podatkovni center

Za vsaka vrata moramo vzdrževati podatek o tem, ali so trenutno odprta ali zaprta (to potrebujemo, da bomo lahko preverjali, ali je kak dogodek v vhodni datoteki nepravilen), in o tem, ali so doslej kdaj že bila odprta (to potrebujemo za izpis na koncu, če v vhodnih podatkih ni napak). To lahko predstavimo z dvema tabeloma oz. seznamoma logičnih vrednosti (ki jih na začetku vse inicializiramo na **false**). Dogodke bomo brali vrstico po vrstico in sproti popravljali vsebino tabel in preverjali, ali je prišlo do odpiranja že odprtih ali zapiranja že zaprtih vrat. Na koncu (ko pridemo do zapiranja vhodnih vrat) pa moramo še preveriti, če so takrat vse omare zaprte, in če so, izpisati tiste, ki so bile dotlej vsaj enkrat odprte.

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

int main()
{
  // Preberimo število omar.
  int n; cin >> n;

  // Pripravimo vektorja s podatki o omarah.
  vector<bool> odprta(n + 1, false), kdajOdprta(n + 1, false);

  // Prebirajmo dogodke in jih preverjajmo.
  while (true)
  {
    // Preberimo naslednji dogodek.
    int vrata, odpiranje; cin >> vrata >> odpiranje;

    // Preverimo, ali je prišlo do napake.
    if (odpiranje && odprta[vrata]) {
      cerr << "Odpiranje že odprtih vrat." << endl; return 1; }
    else if (! odpiranje && ! odprta[vrata]) {
      cerr << "Zapiranje že zaprtih vrat." << endl; return 2; }

    // Popravimo podatke o stanju teh vrat.
    odprta[vrata] = (odpiranje == 1);
    if (odpiranje) kdajOdprta[vrata] = true;

    // Ko se vhodna vrata zaprejo, končajmo.
    if (vrata == 0 && ! odpiranje) break;
  }

  // Preverimo, če so vse omare zaprte.
  for (int omara = 1; omara <= n; ++omara) if (odprta[omara]) {
    cerr << "Omara je na koncu odprta." << endl; return 3; }

  // Če smo prišli do sem, ni napak in moramo izpisati omare, ki so bila kdaj odprte.
  for (int omara = 1; omara <= n; ++omara)
    if (kdajOdprta[omara]) cout << omara << endl;
  return 0;
}
```

Zapišimo takšno rešitev še v pythonu:

```
# Preberimo število omar.
import sys
n = int(sys.stdin.readline())

# Pripravimo seznama s podatki o omarah.
odprta = [False] * (n + 1); kdajOdprta = [False] * (n + 1)

# Prebirajmo dogodke in jih preverjajmo.
for vrstica in sys.stdin.readlines():

  # Preberimo naslednji dogodek.
  vrata, odpiranje = map(int, vrstica.split())

  # Preverimo, ali je prišlo do napake.
  if odpiranje and odprta[vrata]:
    sys.stderr.write("Odpiranje že odprtih vrat.\n"); sys.exit(1)
  if not odpiranje and not odprta[vrata]:
    sys.stderr.write("Zapiranje že odprtih vrat.\n"); sys.exit(2)

  # Popravimo podatke o stanju teh vrat.
  odprta[vrata] = (odpiranje == 1)
  if odpiranje: kdajOdprta[vrata] = True

# Na koncu preverimo, če so vse omare zaprte.
for omara in range(1, n + 1):
  if odprta[omara]: sys.stderr.write("Omara je na koncu odprta.\n"); sys.exit(3)

# Če smo prišli do sem, ni napak in moramo izpisati omare, ki so bila kdaj odprte.
print([omara for omara in range(1, n + 1) if kdajOdprta[omara]])
```

2. Plesalci

Nalogo lahko rešimo s simulacijo. Oštevilčimo plesalce od 1 do n glede na njihov začetni položaj; razpored plesalcev v določenem trenutku lahko potem predstavimo s tabelo, recimo T, v kateri element $T[p]$ pove, na katerem mestu se takrat nahaja plesalec številka p. Tudi opis posameznega dejanja je podobna tabela, recimo D, v kateri element $D[i]$ pove, kam se v tem dejanju premakne plesalec, ki je bil na začetku dejanja na mestu i. Če tu za i vzamemo $T[p]$, torej mesto, kjer se trenutno nahaja plesalec p, vidimo, da se ta plesalec v tem dejanju premakne na mesto $D[T[p]]$. Tako lahko (v zanki po plesalcih) za vsakega plesalca izračunamo novi položaj in s tem dobimo novi razpored ob koncu dejanja.

Gornji postopek lahko izvedemo v zanki po vseh dejanjih in tako dobimo razpored plesalcev po prvem izvajanju celotnega plesa. Če je to že enako začetnemu razporedu (kjer za vsak p velja, da plesalec p stoji na mestu p), lahko končamo, sicer pa moramo ples izvesti še enkrat (in to ponavljati, dokler ne pridemo nazaj na začetni razpored). Pri tem pa bi bilo neučinkovito, če bi vsakič simulirali vsako dejanje plesa posebej; dovolj je, če to naredimo prvič, nato pa si zapomnimo, kako so se prerazporedili plesalci od začetka do konca enega izvajanja plesa. Tako bomo za simulacijo prvega izvajanja plesa porabili $O(nm)$ časa, za vsako nadaljnje izvajanje pa le še $O(n)$ časa.

Oglejmo si implementacijo te rešitve v C++. Za vhodne podatke predpostavimo, da jih dobimo kot vektor vektorjev (za vsako dejanje po en vektor z n elementi). Naš podprogram bo interno štel plesalce in mesta od 0 naprej namesto od 1 naprej, ker bomo tako lahko njihove številke lažje uporabljali kot indekse v tabele oz. vektorje; zato vhodnim podatkom (iz spremenljivke dejanja) odštejemo 1, preden jih uporabimo.

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

int KolikoIzvajanj(const vector<vector<int>> &dejanja)
{
  int n = dejanja[0].size(); // število plesalcev

  // Izračunajmo končni razpored plesalcev po eni izvedbi plesa.
  vector<int> zacetno(n); for (int i = 0; i < n; ++i) zacetno[i] = i; // začetno stanje
  vector<int> ples = zacetno, novo(n);
  for (auto &dejanje : dejanja) {

    // Plesalec, ki je bil na začetku plesa na mestu „i“, se je do začetka tega dejanja
```

```
    // premaknil na mesto ples[i], v tem dejanju pa od tam pride na mesto dejanje[ples[i]] - 1.
    for (int i = 0; i < n; ++i) novo[i] = dejanje[ples[i]] - 1;
    swap(novo, ples); }
  // Ponavljajmo ples, dokler ne pridemo nazaj v začetno stanje.
  vector<int> stanje = ples; int stIzvajanj = 1;
  while (stanje != zacetno) {
    // Ponovimo ples še enkrat.
    for (int i = 0; i < n; ++i) novo[i] = ples[stanje[i]];
    swap(novo, stanje); ++stIzvajanj; }
  return stIzvajanj;
}
```

Zapišimo takšno rešitev še v pythonu:

```
from typing import Sequence

def KolikoIzvajanj(dejanja: Sequence[Sequence[int]]) -> int:
  n = len(dejanja[0]) # število plesalcev
  # Izračunajmo končni razpored plesalcev po eni izvedbi plesa.
  zacetno = list(range(n)) # začetno stanje
  ples = zacetno[:]
  for dejanje in dejanja:
    # Plesalec, ki je bil na začetku plesa na mestu „i“, se je do začetka tega dejanja
    # premaknil na mesto ples[i], v tem dejanju pa od tam pride na mesto dejanje[ples[i]] - 1.
    ples = [dejanje[ples[p]] - 1 for p in range(n)]
  # Ponavljajmo ples, dokler ne pridemo nazaj v začetno stanje.
  stIzvajanj = 1; stanje = ples
  while stanje != zacetno:
    # Ponovimo ples še enkrat.
    stanje = [ples[stanje[p]] for p in range(n)]
    stIzvajanj += 1
  return stIzvajanj
```

Dosedanja rešitev je za naše namene dovolj dobra, vseeno pa razmislimo še o učinkovitejši rešitvi. Matematično si lahko vsako dejanje predstavljamo kot permutacijo množice $\{1, \ldots, n\}$; za d-to dejanje je to recimo permutacija π_d. Tudi celoten ples lahko opišemo s takšno permutacijo, ki ni nič drugega kot kompozitum permutacij, ki predstavljajo posamezna dejanja: plesalec p pride na mesto $\pi(p) = \pi_m(\pi_{m-1}(\cdots \pi_2(\pi_1(p)) \cdots))$, torej ples opisuje permutacija $\pi = \pi_m \circ \pi_{m-1} \circ \ldots \circ \pi_1$. Če ples izvedemo še enkrat, pride plesalec p na mesto $\pi(\pi(p))$, torej $\pi^2(p)$; in v splošnem: po k izvedbah plesa je plesalec p na mestu $\pi^k(p)$. Naloga potem pravzaprav sprašuje po najmanjšem k, pri katerem je π^k ravno identična permutacija (torej taka, pri kateri za vsakega plesalca p velja $\pi^k(p) = p$).

Permutacijo π lahko predstavimo tudi tako, da jo razbijemo na cikle. Pri $\pi = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{smallmatrix}\right)$ tako na primer vidimo, da gre plesalec 1 na mesto 3, plesalec 3 pa na mesto 1 in tako tvorita cikel $(1\,3)$; plesalec 2 gre na mesto 4, plesalec 4 na mesto 5 in plesalec 5 na mesto 2, kar nam dá še en cikel $(2\,4\,5)$; tako je $\pi = (1\,3)(2\,4\,5)$. V vsaki izvedbi plesa se vsak plesalec premakne za eno mesto naprej po svojem ciklu. Tako se plesalci na ciklih dolžine 2 znajdejo na svojem začetnem položaju po vsaki drugi izvedbi plesa; plesalci na ciklih dolžine 3 po vsaki tretji izvedbi; in tako naprej. Rezultat, po katerem sprašuje naloga, je torej najmanjši skupni večkratnik dolžin ciklov v permutaciji, ki opisuje naš ples. Ker imamo n plesalcev, so tudi dolžine ciklov največ n; v $O(n \log n)$ časa jih lahko razcepimo na prafaktorje (s pomočjo Eratostenovega rešeta) in si za vsak prafaktor zapomnimo najvišjo stopnjo, s katero se pojavlja v kakšnem od teh razcepov; tisto je potem tudi njegova stopnja v najmanjšem skupnem večkratniku, tako da jih moramo le še potencirati na ustrezno stopnjo in zmnožiti med seboj, pa bomo dobili iskani rezultat.

3. Nizi

Naloga je zelo primerna za reševanje z rekurzijo. Naj bo t izhodni niz, ki ga gradimo znak po znak in ga na koncu izpišemo. Na vsakem koraku moramo na vse možne (primerne)

načine izbrati, katerega izmed znakov s-ja bomo uporabili kot naslednji znak t-ja, nato pa izvesti rekurzivni klic, ki bo po enakem postopku zapolnil še preostanek t-ja; ko zapolnimo še zadnjo črko t-ja, pa ta niz izpišemo.

Kateri znaki s-ja pa pridejo v poštev za naslednji znak t-ja? Recimo, da moramo zapolniti še k' (od k) znakov niza t in da nam je ostalo neuporabljenih še n' (od n) znakov niza s. Če je $n' = k'$, moramo odslej na vsakem koraku vzeti v t enega od še neuporabljenih znakov s-ja, drugače naš t na koncu ne bo ustrezal zahtevi, da mora vsebovati vsak znak s-ja vsaj enkrat. Če pa je $n' < k'$, lahko za trenutni znak t-ja vzamemo poljuben znak s-ja, saj bomo imeli kasneje še dovolj časa, da uporabimo vse še neuporabljene znake s-ja.

Podatke o tem, kateri znaki s-ja so še neuporabljeni, bi lahko prenašali pri rekurziji kot n-bitno celo število, v katerem bi i-ti bit povedal, ali smo i-ti znak s-ja že uporabili ali ne. Elegantna rešitev pa je tudi ta, da že uporabljene znake premaknemo na konec s-ja, tako da so neuporabljeni znaki vedno na začetku; potem je dovolj kot parameter pri rekurziji prenašati že zgolj število neuporabljenih znakov s-ja. Oglejmo si primer takšne rešitve v C++:

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

// Ta podprogram na vse možne načine zapolni t[0..k − 1] z znaki iz niza „s“
// (pri čemer poskrbi, da bo vsak znak iz s[0..n − 1] uporabljen vsaj enkrat)
// in vse tako dobljene nize izpiše.
void Rekurzija(string &s, int n, string &t, int k)
{
  // Na vse možne načine izberimo, katerega izmed znakov s[0..n − 1]
  // bomo uporabili kot t[k − 1]. Če je v t-ju samo še toliko prostih mest,
  // kolikor je v s-ju neuporabljenih znakov, moramo nujno uporabiti
  // enega od neuporabljenih znakov, sicer pa lahko uporabimo poljuben znak s-ja.
  for (int i = 0; i < (n == k ? n : s.length()); ++i)
  {
    t[k − 1] = s[i];
    // Če je k = 1, smo s tem zapolnili vse znake t-ja in ga moramo le še izpisati.
    if (k == 1) { cout << t << endl; continue; }
    // Sicer nadaljujmo z rekurzivnim klicem.
    // Če je bil s[i] doslej neuporabljen, ga premaknimo med uporabljene.
    if (i < n) swap(s[i], s[n − 1]);
    // Na vse možne načine zapolnimo preostanek t-ja.
    Rekurzija(s, (i < n) ? n − 1 : n, t, k − 1);
    // Obnovimo prejšnje stanje s-ja.
    if (i < n) swap(s[i], s[n − 1]);
  }
}

void NastejNize(string s, int k) { string t(k, '.'); Rekurzija(s, s.length(), t, k); }
```

Zapišimo še rešitev v pythonu. Tukaj niza s ne moremo spreminjati, kot smo to počeli v gornji rešitvi; lahko bi sicer ustvarili nov niz oblike s[:i] + s[i + 1:], torej s brez i-te črke, vendar bi nam to vsakič vzelo $O(n)$ časa; lahko bi namesto tega predelali s v seznam (list), ki bi ga potem lahko spreminjali; lahko pa za spremembo pustimo niz s pri miru in kot parameter prenašamo število, v katerem vsak bit pove, ali smo tisti znak s-ja že uporabili ali ne. Spodnja rešitev temelji na zadnji od teh možnosti:

```
def NastejNize(s: str, k: int) -> None:
  # „t“ je seznam, v katerega zlagamo znake izhodnega niza.
  t = [""] * k; n = len(s)
  # Spodnji podprogram na vse možne načine zapolni t[ti:] z znaki
  # niza „s“ (pri čemer poskrbi, da bo vsak znak s-ja uporabljen vsaj enkrat)
```

```
# in vse tako dobljene nize izpiše.
def Rekurzija(uporabljeni: int, stNeuporabljenih: int, ti: int) -> None:
    # Na vse možne načine izberimo, katerega izmed znakov s-ja
    # bi uporabili za t[ti].
    for si in range(n):
        # Morda smemo uporabljati le še neuporabljene znake.
        zeUporabljen = (uporabljeni & (1 << si) != 0)
        if k - ti == stNeuporabljenih and zeUporabljen: continue
        # Postavimo ga na trenutno mesto v t.
        t[ti] = s[si]
        # Če smo prišli do konca t-ja, ga izpišimo.
        if ti == k - 1: print("".join(t))
        # Sicer nadaljujmo rekurzivno, pri čemer označimo s[si] kot uporabljenega.
        else: Rekurzija(uporabljeni | (1 << si),
                        stNeuporabljenih - (0 if zeUporabljen else 1), ti + 1)
# Glavni klic rekurzije: vsi znaki s-ja so še neuporabljeni,
Rekurzija(0, n, 0) # niz t pa gledamo od začetka naprej.
```

4. Zavarovanje

Pri dani minimalni odškodnini o velja, da moramo plačati odškodnino o pri tistih pošiljkah i, za katere je $v_i < o$, pri ostalih pa moramo plačati odškodnino v_i. Odškodnina o torej velja pri najmanj vrednih nekaj pošiljkah. Zato je koristno, če vrednosti pošiljk najprej uredimo naraščajoče; potem odškodnina o velja pri prvih nekaj pošiljkah. To, pri koliko pošiljkah, lahko ugotovimo na primer z bisekcijo oz. dvojiškim iskanjem. Recimo, da se pri tem izkaže, da ima prvih k pošiljk vrednost $< o$, ostalih $n-k$ pošiljk pa vrednost $\geq o$; skupna vrednost odškodnine je torej $o \cdot k + v_{k+1} + v_{k+2} + \ldots + v_n$. Da bomo lahko takšne zneske hitreje računali, si je koristno vnaprej pripraviti tabelo delnih vsot, v kateri bomo za vsak k imeli vsoto zadnjih $n-k$ pošiljk. Zdaj imamo vse, kar potrebujemo, da lahko opišemo naš postopek s psevdokodo:

```
uredi pošiljke naraščajoče po vrednosti, tako da velja v_1 ≤ v_2 ≤ ... ≤ v_n;
(* Pripravimo tabelo delnih vsot s[0..n]. *)
s[n] := 0; for k := n downto 1 do s[k - 1] := s[k] + v_k;
for i := 1 to m: (* Obdelajmo vse minimalne odškodnine. *)
  (* Naj bo k število pošiljk z vrednostjo pod o_i. *)
  if o_i > v_n then k := n
  else:
    L := 0; D := n;
    while D - L > 1:
      (* Na tem mestu velja v_L < o_i ≤ v_D; pri L = 0 si mislimo v_0 = -∞. *)
      M := ⌈(L + D)/2⌉;
      if o_i ≤ v_M then D := M else L := M;
    k := L; (* Tu velja v_L < o_i ≤ v_{L+1}. *)
  izpiši, da je pri minimalni odškodnini o_i skupni znesek odškodnin
    enak k · o_i + s[k];
```

Časovna zahtevnost tega postopka je $O(n \log n)$ za urejanje pošiljk po vrednosti, $O(n)$ za izračun delnih vsot in nato $O(\log n)$ za bisekcijo pri vsaki od m odškodnin; skupaj torej $O((n+m)\log n)$.

Naloge bi se lahko lotili tudi še drugače; na primer, če se minimalna odškodnina o poveča, potem se poveča tudi k (število pošiljk, katerih vrednost je manjša od o). Če bi torej vrednosti $o_1, \ldots, o_m$ iz vhodnih podatkov najprej uredili naraščajoče, bi se k ves čas le povečeval in ga ne bi bilo treba računati z bisekcijo, pač pa bi pri vsakem naslednjem o_i le pogledali, če je treba dosedanji k še kaj povečati. Lahko si predstavljamo, da zlivamo urejeni zaporedji vrednosti in odškodnin. Tudi tabele delnih vsot potem ne potrebujemo, saj lahko skupno vrednost zadnjih $n-k$ pošiljk popravljamo sproti, ko povečujemo k. Zapišimo s psevdokodo še ta postopek:

uredi pošiljke naraščajoče po vrednosti, tako da velja $v_1 \le v_2 \le \ldots \le v_n$;
$k := 0$; $s := 0$; **for** $i := 1$ **to** n **do** $s := s + v_i$;
uredi odškodnine naraščajoče po vrednosti, tako da velja $o_1 \le o_2 \le \ldots \le o_m$;

for $i := 1$ **to** m: (* *Obdelajmo vse minimalne odškodnine.* *)
  while $k < n$ **and** $v_{k+1} < o_i$:
    $k := k + 1$; $s := s - v_k$;
  izpiši, da je pri minimalni odškodnini o_i skupni znesek odškodnin enak $k \cdot o_i + s$;

Če bi želeli rezultate izpisati v prvotnem vrstnem redu o_i (takem, v kakršnem smo jih dobili v vhodnih podatkih), pa bi si bilo dobro pri urejanju ob vsakem o_i zapomniti tudi njegov položaj v vhodnih podatkih, rezultate zapisovati v neko tabelo in jih na koncu izpisati v pravem vrstnem redu. V vsakem primeru ima ta rešitev časovno zahtevnost $O(n \log n + m \log m)$ zaradi urejanja obeh zaporedij; zlivanje sámo pa vzame le $O(n+m)$ časa.

Še ena različica te druge rešitve je, da zložimo vse vrednosti $v_1, \ldots, v_n$ in $o_1, \ldots, o_m$ v en seznam dolžine $n+m$ (pri vsakem elementu seznama vzdržujmo tudi podatek o tem, ali gre za vrednost pošiljke ali za odškodnino), ga uredimo in se v zanki sprehodimo po njem. Če je trenutni element seznama vrednost neke pošiljke, povečajmo k in popravimo s; če pa je trenutni element ena od odškodnin, izračunajmo skupni znesek odškodnine $k \cdot o_i + s$. Ta rešitev ima časovno zahtevnost $O((n+m)\log(n+m))$ zaradi urejanja seznama dolžine $n+m$.

5. Najkrajše poti

Nalogo lahko rešujemo z iskanjem v širino po prostoru stanj, pri čemer v opis stanja poleg položaja na mreži vključimo tudi podatek o tem, ali smo na poti do tja že odstranili kakšno oviro ali ne (kajti če smo jo, to pomeni, da smemo v bodoče hoditi samo še po prehodnih poljih). Stanje si lahko torej predstavljamo kot trojico (x, y, o), kjer je $o \in \{0, 1\}$ število doslej odstranjenih ovir. V vrsti Q bomo hranili stanja, do katerih že znamo priti, nismo pa še pogledali, kam se da priti iz njih; poleg tega bomo imeli še tabelo d, v kateri bomo za vsako stanje hranili dolžino najkrajše poti do njega (če do njega še ne poznamo nobene poti, bo tam -1). Na začetku dodamo v vrsto začetno stanje $(0, 0, 0)$, pri katerem se nahajamo v zgornjem levem kotu mreže — polju $(0, 0)$ — in nismo odstranili še nobene ovire; nato pa na vsakem koraku vzamemo stanje z začetka vrste in pogledamo, kam se lahko iz njega premaknemo. Pri tem pazimo na možnosti, da je premik neveljaven (če bi padli ven iz mreže ali pa bi morali odstraniti oviro, pri čemer smo nekoč prej eno že odstranili). Na novo dosežena stanja dodajamo v vrsto, ustavimo pa se lahko takoj, ko dosežemo kakšno od stanj v spodnjem desnem kotu, torej v polju $(w-1, h-1)$. Mrežo si mislimo predstavljeno z dvodimenzionalno tabelo $w \times h$ logičnih vrednosti, v kateri element $Prehodno[x, y]$ pove, ali je polje (x, y) prehodno ali ne. Zapišimo zdaj postopek iskanja v širino s psevdokodo:

podprogram NajkrajšaPot(w, h, $Prehodno$):
  if $w = h = 1$ **then return** 0; (* *robni primer* *)
  naj bo d tabela velikosti $w \times h \times 2$; (* *dolžine najkrajših poti* *)
  for $y := 0$ **to** $h - 1$ **do for** $x := 0$ **to** $w - 1$ **do for** $o := 0$ **to** 1 **do** $d[x, y, o] := -1$;
  $DX := [-1, 1, 0, 0]$; $DY := [0, 0, -1, 1]$; (* *možne smeri premika* *)

  $Q :=$ prazna vrsta; dodaj v Q stanje $(0, 0, 0)$; $d[0, 0, 0] := 0$;
  while Q ni prazna:
    naj bo (x, y, o) stanje na začetku vrste Q; pobriši ga iz Q;
    for $smer := 1$ **to** 4:

      $x' := x + DX[smer]$; $y' := x + DY[smer]$;

      (* *Ali pademo pri tem premiku iz mreže?* *)
      if $x' < 0$ **or** $x' \ge w$ **or** $y' < 0$ **or** $y' \ge h$ **then continue**;

      (* *Ali moramo pri tem odstraniti oviro? In če da, ali je to prvič?* *)
      if $Prehodno[x, y]$ **then** $o' := o$
      else if $o = 1$ **then continue else** $o' := 1$;

      (* *Če smo do tega stanja prišli prvič, ga dodajmo v vrsto.* *)

if $d[x', y', o'] \geq 0$ **then continue**;
$d[x', y', o'] := d[x, y, o] + 1$; dodaj (x', y', o') na konec vrste Q;
(* *Čim dosežemo spodnji desni kot, lahko končamo.* *)
if $x' = w - 1$ **and** $y' = h - 1$ **then return** $d[x', y', o]$;
return -1; (* *Iskana pot sploh ne obstaja.* *)

Časovna zahtevnost te rešitve je $O(w \cdot h)$; za vsako polje mreže imamo po dve stanji in za vsako stanje moramo pregledati največ štiri možne premike, torej imamo z vsakim poljem konstantno mnogo dela.

Še en način, kako rešiti to nalogo, pa je naslednji: uporabimo spet iskanje v širino, vendar tokrat ne po prostoru stanj, pač pa kar po prvotni mreži; opis stanja je torej zdaj le par koordinat (x, y). Pri tem pa dodajmo omejitev, da na neprehodna polja sicer lahko vstopimo, ne bomo pa gledali možnih premikov iz takih polj. Iskanje v širino izvedimo dvakrat, enkrat z začetkom v zgornjem levem kotu in enkrat z začetkom v spodnjem desnem. Tako sčasoma za vsako polje (x, y) dobimo njegovo oddaljenost (po najkrajši poti po prehodnih poljih) od obeh kotov; recimo tema oddaljenostma $d_1(x, y)$ in $d_2(x, y)$. Če kakšno polje iz kakšnega kota ni dosegljivo, si mislimo, da je na oddaljenosti ∞. Potem nam vsota $d_1(x, y) + d_2(x, y)$ pove dolžino najkrajše take poti, ki se začne v zgornjem levem kotu, gre do polja (x, y) — pri tem odstrani oviro, če je bilo polje (x, y) prej neprehodno — in se od tam nadaljuje do spodnjega desnega kota. Dovolj bo torej, če na koncu vrnemo najmanjšo izmed vsot $d_1(x, y) + d_2(x, y)$ (po vseh poljih mreže). Tudi ta rešitev ima časovno zahtevnost $O(w \cdot h)$.

19. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

26. januarja 2024

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). 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. Podatkovni center

- Naloga ne predpisuje podrobnosti glede oblike izpisa, tako da je vseeno, kaj točno program izpiše v primeru napake ali kako formatira seznam vrat, ki so bila vsaj enkrat odprta (če ni prišlo do napake). Tudi ni treba, da vrača pri različnih napakah različne kode ob zaključku izvajanja, kot to počne naša rešitev.

- Ker je število omar vnaprej omejeno na 100, ni nič narobe, če si program vnaprej pripravi tabelo te maksimalne velikosti.
- Rešitev sme predpostaviti, da v vhodnih podatkih ni drugih napak kot tistih, za katere naloga eksplicitno zahteva, da jih moramo zaznavati. Tako se na primer sme predpostaviti, da je v zadnji vrstici vhodne datoteke res zapiranje vrat št. 0.
- V našem primeru rešitve smo za vsaka vrata hranili po dve logični vrednosti, šlo pa bi seveda tudi drugače. Lahko bi npr. za vsaka vrata hranili eno od treh možnih stanj: trenutno odprta; trenutno zaprta, a so bila v preteklosti nekoč že odprta; trenutno zaprta in še nikoli niso bila odprta. Lahko bi tudi hranili množico odprtih vrat (npr. `set` v pythonu). Vse te rešitve naj štejejo za enako dobre, če delujejo pravilno.
- Okvirni nasvet za točkovanje: 5 točk za primerno branje vhodnih podatkov; 5 točk za preverjanje, ali se kdaj odprejo že odprta vrata ali zaprejo že zaprta vrata; 5 točk za preverjanje, ali so na koncu vse omare zaprte; in 5 točk za izpis omar, ki so bile vsaj enkrat odprte.
- Če program po prvi odkriti napaki nadaljuje z branjem vhodnih podatkov in morda javi še kakšno napako, naj se mu tega ne šteje v slabo.
- Če program po tistem, ko je prišlo do napake, morda vendarle izpiše seznam omar, ki so bile vsaj enkrat odprte, naj se mu zaradi tega odšteje dve točki.
- Če v seznamu omar, ki so bile vsaj enkrat odprte, izpiše tudi vhodna vrata (št. 0), naj se programu zaradi tega odšteje eno točko.

2. Plesalci

- Za vse točke pričakujemo rešitev, ki porabi $O(nm)$ časa za simulacijo le pri prvem izvajanju plesa, pri nadaljnjih izvedbah pa odsimulira ples kot celoto v $O(n)$ časa in torej ne gre vsakič znova po dejanjih. Rešitve, ki porabijo $O(nm)$ časa tudi za vsako nadaljnje izvajanje plesa, naj dobijo največ 13 točk, če so sicer pravilne.
- Ne pričakujemo pa res učinkovitih rešitev, ki bi razbile permutacijo na cikle in računale najmanjši skupni večkratnik njihovih dolžin (čeprav seveda ni narobe, če kdo reši nalogo na ta način).
- Za drobne napake, povezane npr. s tem, ali se šteje plesalce in mesta od 0 naprej namesto od 1 naprej, naj se rešitvi odšteje največ 2 točki.

3. Nizi

- Naloga pravi, naj bo rešitev učinkovita. Pri tem si želimo predvsem, da program ne bi zgeneriral vseh n^k nizov dolžine k iz črk niza s ter pri vsakem šele takrat, ko je zgeneriran do konca, preverjal, ali vsebuje vsako črko s-ja vsaj enkrat ali ne. Rešitve, ki delujejo na ta način, naj dobijo največ 12 točk.
- Rešitev lahko, ko vidi, da mora uporabljati le take črke s-ja, ki še niso bile uporabljene, gre pazljivo samo po neuporabljenih črkah s-ja (kot naš primer rešitve v C++) ali pa gre po vseh črkah s-ja in pri vsaki preveri, ali je bila že uporabljena ali ne (kot naš primer rešitve v pythonu); oboje naj šteje za enako dobro.
- Če program izpiše kak niz po večkrat ali pa za izogibanje temu porabi eksponentno veliko pomnilnika, da shranjuje vse že izpisane nize, naj se mu zato odšteje 5 točk.

4. Zavarovanje

- Pri tej nalogi pričakujemo za vse točke rešitev s časovno zahtevnostjo, manjšo od kvadratne. V rešitvah smo opisali tri različice, eno z zahtevnostjo $O((n+m)\log n)$, eno z $O(n\log n+m\log m)$ in eno z $O((n+m)\log(n+m))$, kar naj vse šteje za enako dobro (verjetno je možna še kakšna različica). Rešitve z zahtevnostjo $O(n \cdot m)$ pa naj dobijo največ 13 točk, če so drugače pravilne.
- V našem opisu rešitve smo podali tudi psevdokodo bisekcije, vendar od tekmovalcev tega ne pričakujemo; dovolj je že, če rešitev pove, kaj točno bi z bisekcijo naredili. (Spomnimo se, da je v mnogih programskih jezikih bisekcija tako ali tako že del standardne knjižnice; npr. `lower_bound` v C++, modul `bisect` v pythonu.) Ravno tako tudi ne pričakujemo, da bi reševalec opisoval podrobnosti postopka urejanja (kot jih tudi mi nismo opisovali v naši rešitvi).

5. Najkrajše poti

- Za vse točke pričakujemo rešitev s časovno zahtevnostjo $O(w \cdot h)$. Rešitve z večjo, vendar še vedno polinomsko časovno zahtevnostjo naj dobijo največ 15 točk; rešitve z eksponentno zahtevnostjo pa največ 10 točk.
- Podrobnosti predstavitve mreže pri tej nalogi niso zelo pomembne; v našem opisu rešitve smo predlagali tabelo logičnih vrednosti, enako dober pa bi bil na primer tudi seznam h nizov s po w znaki, kjer vsak znak predstavlja eno polje mreže.
- V našem opisu rešitve smo za koordinate polj v mreži uporabljali števila od 0 do $w-1$ oz. $h-1$; enako dobro bi bilo seveda tudi od 1 do w oz. h.
- Rešitev, ki sploh ne poskuša odstranjevati ovir, naj dobi največ polovico točk, kolikor bi jih sicer dobila glede na svojo časovno zahtevnost. Po drugi strani pa, če rešitev poskuša odstraniti po največ eno oviro, vendar zaradi kakšnih manjših napak v implementaciji včasih ne odstrani nobene ovire ali pa več kot eno, naj se ji zaradi takšnih manjših napak odšteje največ 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
1. Podatkovni center	srednje težka naloga v prvi ali lahka v drugi skupini
2. Plesalci	težja naloga v prvi ali lažja v drugi skupini
3. Nizi	srednje težka naloga v drugi skupini
4. Zavarovanje	srednje težka naloga v drugi ali lažja v tretji skupini
5. Najkrajše poti	težja naloga v drugi ali lažja v tretji skupini

Če torej na primer neki 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.