15. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

24. januarja 2020

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.");
  }
}
```

15. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

24. januarja 2020

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. PINi

Primož si ne uspe zapomniti varnostne kode (PIN) za svojo bančno kartico, zato si jo bo napisal kar na zadnjo stran kartice. Ker pa ima nekaj malega pojma o varovanju podatkov, je sklenil, da bo številko prikril s postopkom, ki ga je izumil sam.

Varnostna koda je štirimestno število z vrednostjo od 0000 do 9999. Zapišemo jo lahko tudi kot zaporedje štirih števk *abcd*, kjer lahko vsaka števka zavzame vrednosti od 0 do 9.

Namesto kode *abcd* bi rad Primož na bančno kartico zapisal štirimestno število *wxyz*, iz katerega bo izračunal originalno kodo *abcd* po naslednjem postopku:

$a = z$;
$b = w \oplus x$;
$c = x \oplus y$;
$d = y \oplus z$;

Računsko operacijo $\oplus$ je definiral takole: operacija sešteje dve števili, nato pa izračuna ostanek pri deljenju z 11. Če je rezultat 10, ga spremeni v 0.

Če bi imel Primož varnostno kodo 6910, bi si na kartico napisal 1846. Iz te številke lahko z nekaj enostavnimi operacijami dobi originalno varnostno kodo:

$wxyz = 1846$, torej $w = 1$, $x = 8$, $y = 4$, $z = 6$;

$a = z = 6$;
$b = w \oplus x = 1 \oplus 8 = 9$;
$c = x \oplus y = 8 \oplus 4 = 1$;
$d = y \oplus z = 4 \oplus 6 = 0$;

Koda je torej $abcd = 6910$.

Kmalu pa je ugotovil, da za svojo varnostno kodo ne more najti ustrezne začetne kombinacije. Malo je še poskušal in ugotovil, da obstaja kar nekaj takih štirimestnih kod, ki nikakor ne morejo biti rezultat njegovega postopka.

Namesto da bi si izmislil boljši postopek prikrivanja številke, je sklenil, da bo stopil do bankomata in spremenil varnostno kodo svoje kartice. Še pred tem pa bi rad ugotovil, katerih varnostnih kod ne sme uporabiti.

Napiši program, ki pregleda vse možne varnostne kode od 0000 do 9999 in izpiše vse, ki se jih ne da izračunati po zgoraj opisanem postopku. Tvoja rešitev naj bo čim učinkovitejša.

2. INTERCAL

V ezoteričnem programskem jeziku INTERCAL se vsak ukaz začne z „DO“, „PLEASE“ ali „PLEASE DO“ (v nadaljevanju ukaza se te fraze ne pojavljajo). Tistim, ki se začnejo s „PLEASE“ ali „PLEASE DO“, rečemo *vljudni*, ostalim (tistim, ki se začnejo z „DO“) pa *nevljudni*. To, na katero od teh treh fraz se ukaz začne, nič ne vpliva na njegov pomen ali delovanje, vendar pa prevajalnik kode ne bo prevedel, če se vljudni ukazi v njem pojavljajo preredko ali prepogosto (ker uporabnik ni vljuden ali pa je pretirano vljuden). Natančneje povedano, delež vljudnih ukazov (glede na vse ukaze) mora biti vsaj a in kvečjemu b. Pri tem sta a in b podana kot deleža, ne v odstotkih, torej velja $0 \leq a \leq b \leq 1$.

Napiši program ali podprogram (funkcijo), ki kot vhodne podatke dobi ali prebere obe meji (a in b) in zaporedje ukazov (ukazov je največ 10 000, vsak je v svoji vrstici, ta se začne na eno od zgornjih treh fraz in presledek in je dolga največ 100 znakov). Izpiše naj primerno popravljeno zaporedje ukazov, pri čemer sme spreminjati le začetno frazo („DO“, „PLEASE“ ali „PLEASE DO“) posameznega ukaza. V popravljenem zaporedju naj bo delež vljudnih ukazov vsaj a in kvečjemu b. Če se zaporedja ne da popraviti v skladu s temi omejitvami, naj tvoj program izpiše, da je problem nerešljiv.

Ukaze lahko bereš s standardnega vhoda ali iz datoteke `vhod.txt` ali pa predpostaviš, da jih dobiš podane v neki tabeli, seznamu, vektorju ali čem podobnem.

Primer. Recimo, da imamo $a = 0{,}5$ in $b = 0{,}8$ ter da dobimo naslednje zaporedje ukazov:

```
DO ,1 <- #13
PLEASE DO ,1 SUB #1 <- #238
DO ,1 SUB #2 <- #108
DO ,1 SUB #3 <- #112
DO ,1 SUB #4 <- #0
```

Tu je vljuden le en ukaz od petih; delež vljudnih ukazov je torej 1/5, kar je manj od a. Primerno popravljeno zaporedje ukazov je na primer takšno (to ni edina možna rešitev):

```
DO ,1 <- #13
PLEASE DO ,1 SUB #1 <- #238
DO ,1 SUB #2 <- #108
PLEASE ,1 SUB #3 <- #112
PLEASE ,1 SUB #4 <- #0
```

Zdaj so vljudni trije ukazi od petih, torej je delež vljudnih ukazov 3/5, kar je res večje ali enako a ter manjše ali enako b.

3. Najdaljša pot

Na pravokotni plošči s karirasto mrežo imamo v vsakem polju ploščico z vrisano puščico, ki lahko kaže v eno od štirih smeri ($\leftarrow, \rightarrow, \uparrow, \downarrow$). Ploščice so naključno postavljene na vsa polja. Mreža ima največ 1000 stolpcev in 1000 vrstic.

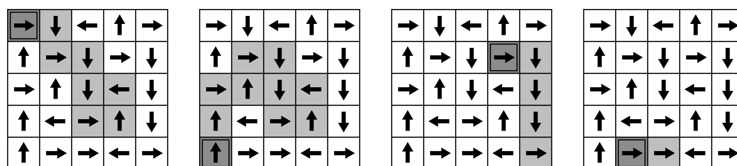
Postavimo se na izbrano začetno polje in se začnemo premikati. Iz polja, na katerem smo trenutno, se lahko premaknemo samo na sosednje polje, na katero kaže puščica našega polja. Na sosednje polje se ne moremo premakniti:

- če se nahajamo na polju, ki je na robu igralne plošče, in puščica našega polja kaže izven igralne plošče,
- če je v sosednjem polju puščica, ki kaže nazaj na naše polje,
- če smo v tem sosednjem polju že bili (križanja in zanke niso dovoljeni).

Opiši postopek (ali napiši program ali podprogram oz. funkcijo — karkoli ti je lažje), ki kot vhodni podatek dobi opis mreže in izračuna dolžino najdaljše poti, ki jo lahko prehodimo, če si smemo začetno polje izbrati poljubno.

Če ti je ta naloga pretežka, lahko za 14 točk od 20 rešiš lažjo različico, pri kateri si začetnega polja ne smemo izbrati poljubno, pač pa vedno začnemo v zgornjem levem polju.

Primer: spodnja slika kaže štiri poti na eni in isti mreži, vsakič z drugim začetnim poljem. Začetno polje je pobarvano temno sivo in ima dvojni rob, ostala polja na poti pa so svetlo siva.



Dolžina poti: 8 10 5 2

Izkaže se, da je pri tej mreži največja možna dolžina poti 10.

4. Domače naloge

Imamo seznam n domačih nalog, ki jih bomo po vrsti reševali d dni. Prvi dan bomo torej rešili prvih nekaj nalog, drugi dan naslednjih nekaj in tako naprej. Vsak dan lahko rešimo poljubno število nalog. Za vsako nalogo poznamo njeno težavnost t_i, ki je realno število med 0 in 10. Učinkovitost vaje za posamezen dan je enaka največji težavnosti naloge, ki smo jo rešili ta dan. **Opiši postopek**, ki ugotovi kako naj seznam nalog razdelimo po dnevih, da bo vsota učinkovitosti po vseh dnevih največja možna. Kot vhodne podatke dobi tvoj postopek števila n, d, $t_1, \ldots, t_n$.

Primer: če imamo $n = 8$ nalog s težavnostmi $[6, 9, 5, 8, 3, 4, 5, 8]$ in bi jih radi razdelili na $d = 3$ dni, je največja možna vsota učinkovitosti enaka 25; dosežemo jo na primer z razdelitvijo $[6, 9], [5, 8, 3], [4, 5, 8]$ (ni pa to edina primerna razdelitev).

5. Razbijanje permutacijske šifre

Začasno smo dobili dostop do super tajne naprave za šifriranje, ki jo za komunikacijo uporabljata KGB in ameriški predsednik. Naprava je pravzaprav permutacijska šifra: kot vhod dobi niz n znakov in izpiše te znake v premešanem vrstnem redu, pri čemer jih vedno premeša na enak način. Ugotovili bi radi, za katero permutacijo gre (torej: kateri vhodni znak pride na katero mesto na izhodu). Pri tem lahko v napravo spuščamo različne nize, dolge po n znakov (ki si jih sami izberemo), in opazujemo, kaj pride ven. Abeceda, iz katere so sestavljeni naši nizi, ima največ b različnih znakov — recimo, da so znaki predstavljeni kar s celimi števili od 0 do $b - 1$.

Opiši postopek, ki za podana n in b sestavi zaporedje vhodnih nizov za šifrirno napravo, nato pa vrnjene zašifrirane nize uporabi, da razbije šifro. Radi bi torej, da bi tvoj postopek na koncu vedel o delovanju naprave dovolj, da bi znal za poljuben niz n znakov napovedati, v kaj ga bo naprava zašifrirala. Za n in b predpostavi, da sta celi števili z območja od 1 do 1000.

15. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

24. januarja 2020

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. PINi

Napišimo si najprej podprogram, ki iz zapisanih števk $wxyz$ izračuna varnostno kodo $abcd$ po pravilih iz besedila naloge. Ker v običajnih programskih jezikih nimamo operatorja $\oplus$, si bomo pomagali z operatorjem za ostanek po deljenju (npr. `%` ali **mod**): operanda moramo sešteti, izračunati ostanek po deljenju z 11, nato pa obdržati le ostanek po deljenju tega ostanka z 10.

```
int IzracunajKodo(int w, int x, int y, int z)
{
   int a = z;
   int b = ((w + x) % 11) % 10;
   int c = ((x + y) % 11) % 10;
   int d = ((y + z) % 11) % 10;
   return a * 1000 + b * 100 + c * 10 + d;
}
```

Če nas zdaj za neko konkretno kodo $abcd$ zanima, ali jo je mogoče dobiti iz kakšne kombinacije $wxyz$, gremo lahko z nekaj gnezdenimi zankami po vseh $wxyz$, pri vsakem izračunamo kodo in preverimo, če se ujema z našo:

```
bool JeDosegljiva(int koda)
{
   for (int w = 0; w <= 9; w++) for (int x = 0; x <= 9; x++)
   for (int y = 0; y <= 9; y++) for (int z = 0; z <= 9; z++)
      if (IzracunajKodo(w, x, y, z) == koda) return true;
   return false;
}
```

Glavni blok programa lahko kliče to funkcijo v zanki po vseh možnih kodah in sproti izpisuje tiste, ki se izkažejo za nedosegljive:

```
int main()
{
   for (int koda = 0; koda <= 9999; ++koda)
      if (! JeDosegljiva(koda)) printf("%04d\n", koda);
   return 0;
}
```

Ta rešitev je precej neučinkovita, saj za vsako od 10000 kod pregleda v najslabšem primeru do 10000 kombinacij $wxyz$, da se prepriča, če je tista koda res nedosegljiva. Boljšo rešitev dobimo, če gremo po vseh kombinacijah $wxyz$ le enkrat, pri vsaki izračunamo njej pripadajočo kodo in v neki tabeli označujemo, katere kode smo na ta način dosegli. Na koncu se moramo le še enkrat sprehoditi po tej tabeli in izpisati kode, ki jih nismo dosegli nobenkrat:

```
int main()
{
   bool dosegljiva[10000];
   for (int koda = 0; koda <= 9999; ++koda) dosegljiva[koda] = false;
   for (int w = 0; w <= 9; w++) for (int x = 0; x <= 9; x++)
   for (int y = 0; y <= 9; y++) for (int z = 0; z <= 9; z++)
```

```
      dosegljiva[IzracunajKodo(w, x, y, z)] = true;
   for (int koda = 0; koda <= 9999; ++koda)
      if (! dosegljiva[koda]) printf("%04d\n", koda);
   return 0;
}
```

Izkaže se, da je nedosegljivih 2008 od vseh 10000 možnih PINov.

Oglejmo si še rešitev v pythonu:

```
def IzracunajKodo(w, x, y, z):
   a = z
   b = ((w + x) % 11) % 10
   c = ((x + y) % 11) % 10
   d = ((y + z) % 11) % 10
   return a * 1000 + b * 100 + c * 10 + d

def JeDosegljiva(koda):
   for w in range(10):
      for x in range(10):
         for y in range(10):
            for z in range(10):
               if IzracunajKodo(w, x, y, z) == koda: return True
   return False

for koda in range(10000):
   if not JeDosegljiva(koda): print("%04d" % koda)
```

In še učinkovitejša različica:

```
dosegljiva = [False] * 10000
for w in range(10):
   for x in range(10):
      for y in range(10):
         for z in range(10):
            dosegljiva[IzracunajKodo(w, x, y, z)] = True
for koda in range(10000):
   if not dosegljiva[koda]: print("%04d" % koda)
```

2. INTERCAL

Za začetek lahko iz števila ukazov (recimo mu n) in števil a in b izračunamo minimalno in maksimalno število vljudnih ukazov, pri katerih se program še prevede. Če označimo število vljudnih ukazov z v, je delež vljudnih glede na vse ukaze enak v/n, torej mora veljati $v/n \geq a$ in $v/n \leq b$. Iz prvega pogoja dobimo $v \geq a \cdot n$, iz drugega pa $v \leq b \cdot n$. Poleg tega mora biti v celo število, torej je najmanjša primerna vrednost v resnici $\lceil a \cdot n \rceil$ (to je vrednost $a \cdot n$, zaokrožena navzgor na celo število), največja pa $\lfloor b \cdot n \rfloor$ (to je vrednost $b \cdot n$, zaokrožena navzdol na celo število). Pri tem se lahko izkaže, da primernega v sploh ni; na primer, če bi imeli $n = 7$ ukazov in meji $a = 0{,}6$ in $b = 0{,}7$, nam spodnja meja $a = 0{,}6$ pove, da potrebujemo vsaj $\lceil a \cdot n \rceil = \lceil 0{,}6 \cdot 7 \rceil = \lceil 4{,}2 \rceil = 5$ vljudnih ukazov, zgornja meja $b = 0{,}7$ pa nam pove, da smemo imeti največ $\lfloor b \cdot n \rfloor = \lfloor 0{,}7 \cdot n \rfloor = \lfloor 4{,}9 \rfloor = 4$ vljudne ukaze. V takem primeru lahko le zaključimo, da je problem nerešljiv, saj obema pogojema hkrati ne bomo mogli ustreči.

Drugače pa si lahko izberemo poljuben v z območja od $\lceil a \cdot n \rceil$ do $\lfloor b \cdot n \rfloor$ in popravimo dano zaporedje ukazov tako, da bo vsebovalo natanko v vljudnih ukazov. Pri tem se nam ni nujno treba ozirati na to, kateri ukazi so bili že od prej vljudni in koliko jih je bilo, saj naloga ne predpisuje nobenih omejitev glede tega, koliko ukazov spremenimo in katere. Obstoječe fraze „DO“, „PLEASE“ ali „PLEASE DO“ lahko preprosto porežemo z začetka ukazov in nato pri prvih v ukazih na začetek postavimo „PLEASE“, pri ostalih pa „DO“ — tako bo nastal program z v vljudnimi ukazi.

```
#include <cmath>
#include <cstring>
#include <vector>
```

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

void Popravi(double a, double b, const vector<string>& ukazi)
{
  // Iz deležev a in b izračunajmo najmanjše in največje
  // dovoljeno število vljudnih ukazov.
  int stUkazov = ukazi.size();
  int minVljudnih = (int) ceil(a * stUkazov);
  int maxVljudnih = (int) floor(b * stUkazov);

  // Če je zgornja meja manjša od spodnje, je problem nerešljiv.
  if (maxVljudnih < minVljudnih) { printf("Problem je neresljiv.\n"); return; }

  // Sprehodimo se po zaporedju ukazov.
  static const vector<string> prefiksi = {"PLEASE DO", "PLEASE", "DO"};
  for (int i = 0; i < ukazi.size(); i++)
  {
    const char *s = ukazi[i].c_str();

    // Poglejmo, kateri prefiks se pojavlja na začetku tega ukaza.
    int dolzinaPrefiksa = 0;
    for (const string &p : prefiksi) {
      int d = p.length();
      if (strncmp(s, p.c_str(), d) == 0) {
        dolzinaPrefiksa = d; break; } }

    // Izpišimo preostanek ukaza s primernim prefiksom.
    printf("%s%s\n", (i < minVljudnih) ? "PLEASE" : "DO", s + dolzinaPrefiksa);
  }
}
```

Zapišimo to rešitev še v pythonu:

```
import math

def Popravi(a, b, ukazi):
  # Iz deležev a in b izračunajmo najmanjše in največje
  # dovoljeno število vljudnih ukazov.
  stUkazov = len(ukazi)
  minVljudnih = int(math.ceil(a * stUkazov))
  maxVljudnih = int(math.floor(b * stUkazov))

  # Če je zgornja meja manjša od spodnje, je problem nerešljiv.
  if maxVljudnih < minVljudnih: print("Problem je neresljiv."); return

  # Sprehodimo se po zaporedju ukazov.
  for i, ukaz in enumerate(ukazi):

    # Odrežimo prefiks, ki se pojavlja na začetku ukaza.
    for p in ["PLEASE DO", "PLEASE", "DO"]:
      if ukaz.startswith(p): ukaz = ukaz[len(p):]; break

    # Izpišimo ukaz s primernim prefiksom.
    print("%s%s" % ("PLEASE" if i < minVljudnih else "DO", ukaz))
```

3. Najdaljša pot

Če si izberemo nek začetni položaj, lahko v zanki sledimo poteku poti v skladu s pravili naloge. Na vsakem koraku izračunamo koordinati polja, proti kateremu kaže puščica na trenutem polju. Če je novi položaj zunaj mreže, je premik neveljaven in pot se konča; podobno tudi, če se izkaže, da puščica na novem polju kaže nazaj na trenutno polje ali pa da smo novo polje obiskali že kdaj prej. Če pa ne drži nič od tega, je premik veljaven in lahko s sledenjem poti nadaljujemo na novem polju. Da lahko poceni preverimo, ali smo neko polje prej že obiskali, si pomagamo s tabelo (v spodnjem podprogramu je to vektor zeBili), v kateri označujemo, katera polja smo že obiskali.

Ker naloga sprašuje po najdaljši poti sploh, po vseh možnih začetnih položajih, moramo razmislek iz gornjega odstavka oviti še v eno zanko, ki preizkusi vse možne

začetne položaje in si zapomni najdaljšo tako odkrito pot. Zapišimo takšno rešitev v C++:

```
#include <vector>
using namespace std;
typedef enum { Gor, Dol, Levo, Desno } Smer;
const int DX[ ] = { 0, 0, −1, 1 };
const int DY[ ] = { −1, 1, 0, 0 };

int NajdaljsaPot(const vector<vector<Smer>> &mreza)
{
  const int w = mreza[0].size(), h = mreza.size();
  vector<int> zeBili(w * h, −1);
  int najDolzina = 0;
  for (int z = 0; z < w * h; z++)
  {
    int x = z % w, y = z / w, dolzina = 1;
    // Izračunajmo dolžino poti z začetkom na polju (x, y).
    // Že obiskana polja bomo v tabeli zeBili označili z vrednostjo z.
    zeBili[z] = z;
    while (true)
    {
      // Izračunajmo novi koordinati pri tem premiku.
      Smer s = mreza[y][x]; x += DX[s]; y += DY[s];
      // Ali bi padli iz mreže?
      if (x < 0 || x >= w || y < 0 || y >= h) break;
      // Ali kaže novo polje nazaj na prejšnje?
      Smer s2 = mreza[y][x];
      if (DX[s2] == −DX[s] && DY[s2] == −DY[s]) break;
      // Ali smo v novem polju že bili?
      if (zeBili[x + y * w] == z) break;
      // Če ne, se premaknimo vanj.
      zeBili[x + y * w] = z; ++dolzina;
    }
    // Če je to najdaljša pot doslej, si jo zapomnimo.
    if (dolzina > najDolzina) najDolzina = dolzina;
  }
  return najDolzina;
}
```

Za označevanje že obiskanih polj bi lahko imeli tabelo oz. vektor logičnih vrednosti (**bool**-ov), ki bi jih pri vsakem začetnem položaju najprej vse inicializirali na **false**. To je rahlo potratno, saj je prav mogoče, da večine polj pri prejšnji poti sploh nismo obiskali in so zato tisti elementi tabele že imeli vrednost **false**. Druga možnost je, da ko določimo dolžino poti (pri nekem začetnem položaju), sledimo isti poti še enkrat in postavljamo vrednosti v zeBili na **false**. Tretja možnost, ki smo jo uporabili v gornjem podprogramu, pa je, da je zeBili vektor celoštevilskih vrednosti (**int** namesto **bool**), v katerem uporabljamo vrednost z (trenutni začetni položaj) kot **true**, katerokoli manjšo vrednost (ki je tam ostala še od prejšnjih začetnih položajev) pa si razlagamo kot **false**.

Še ena možnost pa je, da namesto tabele uporabimo množico oz. razpršeno tabelo, v katero shranjujemo le polja, ki smo jih že obiskali. Ta pristop uporablja spodnja pythonovska različica naše rešitve:

```
Gor = 0; Dol = 1; Levo = 2; Desno = 3
DX = [0, 0, −1, 1]; DY = [−1, 1, 0, 0]

def NajdaljsaPot(mreza):
  w = len(mreza[0]); h = len(mreza)
  najDolzina = 0
  for y0 in range(h):
    for x0 in range(w):
```

```
    # Preglejmo pot z začetkom v (x0, y0).
    x = x0; y = y0; zeBili = set(); zeBili.add((x, y))
    while True:
      # Izračunajmo novi koordinati pri tem premiku.
      s = mreza[y][x]; x += DX[s]; y += DY[s]
      # Ali bi padli iz mreže?
      if x < 0 or x >= w or y < 0 or y >= h: break
      # Ali kaže novo polje nazaj na prejšnje?
      s2 = mreza[y][x]
      if DX[s2] == −DX[s] and DY[s2] == −DY[s]: break
      # Ali smo v novem polju že bili?
      if (x, y) in zeBili: break
      # Če ne, se premaknimo vanj.
      zeBili.add((x, y))
    najDolzina = max(najDolzina, len(zeBili))
  return najDolzina
```

Pri mreži velikosti $w \times h$ je lahko pot dolga največ $w \cdot h$ polj (npr. če lepo sistematično cikcaka po celi mreži). V takem primeru bomo pri $O(wh)$ začetnih položajih morali slediti poti vsaj $O(wh)$ korakov daleč, preden se bomo ustavili, zato je časovna zahtevnost naše rešitve v najslabšem primeru kar $O(w^2h^2)$. Razmislimo še o učinkovitejši rešitvi.

Dolžino poti, ki se začne na polju u, bomo označili z $d[u]$. Naloga torej sprašuje po $\max_u d[u]$, pri čemer gre u po vseh poljih mreže.

Videli smo, da se pot konča bodisi zato, ker naslednja poteza s trenutnega polja sploh ni možna (ker bi padli iz mreže ali pa se premaknili v polje, ki kaže nazaj na trenutno polje) ali pa ker bi nas pripeljala na neko že prej obiskano polje — v tem slednjem primeru torej vidimo, da možni premiki tvorijo cikel. Če tak cikel sestavlja k polj, lahko za vsako od njih takoj zaključimo, da je dolžina poti z začetkom na tistem polju natanko k (ker bi taka pot sledila ciklu, dokler ne bi prišla nazaj na začetno polje).

Kaj pa, če u ne leži na ciklu? Ena možnost je, da premik z njega sploh ni mogoč; tedaj je pač $d[u] = 1$. Sicer pa nas premik s polja u pripelje v enega od njegovih sosedov, recimo v; nadaljevanje poti točke u gotovo ne bo več doseglo (saj bi to pomenilo, da u leži na ciklu, kar pa ni res), torej bo nadaljevanje poti poteklo popolnoma enako, kot če bi s potjo začeli šele pri v namesto pri u. Dolžina tega preostanka poti je torej $d[v]$, celotna pot skupaj z začetnim korakom od u do v pa je še za en korak daljša; torej imamo $d[u] = d[v] + 1$.

Recimo, da začnemo s potjo pri u_1 in izvedemo $k-1$ korakov: $u_1 \to \ldots \to u_k$; in recimo, da iz u_k naslednji korak ni mogoč zato, ker bi padli čez rob mreže ali pa prišli v polje, ki kaže nazaj na u_k. Potem lahko takoj zaključimo, da je $d[u_k] = 1$, $d[u_{k-1}] = 2$ in tako nazaj do $d[u_1] = k$. Za vsa ta polja torej zdaj poznamo dolžino poti z začetkom na njih in se nam z njimi ne bo treba več ukvarjati.

Podobno je, če se nam pot sčasoma zacikla: recimo, da pri u_k opazimo, da nas premik od tam pelje v u_i za nek $i < k$. Polja $u_i, u_{i+1}, \ldots, u_k$ torej tvorijo cikel dolžine $c := k - i + 1$. Zanje torej lahko zaključimo, da je $d[u_i] = d[u_{i+1}] = \ldots = d[u_k] = c$; za polja pred njimi na poti pa, da je $d[u_{i-1}] = c + 1$ in tako nazaj do $d[u_1] = k$.

Če pa med sledenjem poti iz u_1 pridemo (recimo po $k-1$ korakih) do nekega polja u_k, za katero že poznamo $d[u_k]$, lahko takoj zaključimo, da je $d[u_{k-1}] = d[u_k] + 1$ in tako nazaj do $d[u_1] = d[u_k] + k - 1$. Nadaljevanju poti zato v tem primeru ni treba slediti.

V vsakem primeru torej za vsak korak, ki ga pri sledenju poti naredimo, uspemo tudi določiti $d[\cdot]$ za eno novo polje. Zato pri sledenju vseh poti skupaj (po vseh možnih začetnih položajih) naredimo le toliko korakov, kolikor je vseh polj v mreži; časovna zahtevnost te rešitve je tako le še $O(wh)$. Zapišimo še to rešitev:

```
def NajdaljsaPot2(mreza):
  w = len(mreza[0]); h = len(mreza); n = w * h
  # Polja bomo predstavili s števili: (x, y) predstavlja število x + y * w.
  # Funkcija Nasl(u) vrne številko polja, v katero se premaknemo iz polja u.
  # Če premik iz u ni veljaven, vrne −1.
```

```
def Nasl(u):
   # Pretvorimo številko polja v koordinate.
   x = u % w; y = u // h; s = mreza[y][x]
   # Izračunajmo novi položaj.
   x += DX[s]; y += DY[s]
   # Ali smo padli iz mreže?
   if x < 0 or x >= w or y < 0 or y >= h: return −1
   # Ali puščica v novem polju kaže nazaj v staro?
   s2 = mreza[y][x]
   if DX[s2] == −DX[s] and DY[s2] == −DY[s]: return −1
   # Vrnimo številko novega polja.
   return x + y * w

# V tabeli d bomo hranili dolžine poti z začetkom pri posameznem polju.
d = [0] * n
for z in range(n):
   # Mogoče že poznamo dolžino poti z začetkom na polju z.
   if d[z]: continue
   # Sicer sledimo poti iz z, dokler se pot ne konča, zacikla ali pa naleti na polje,
   # pri katerem dolžino poti od tam naprej že poznamo. k šteje prehojena polja,
   # u je trenutni položaj; v d[u] vpisujemo števila −1, −2, . . . , ki povedo
   # položaj polja na poti (z njihovo pomočjo bomo lažje določili dolžino cikla).
   k = 1; u = z; d[u] = −k
   while True:
      v = Nasl(u)
      if v < 0 or d[v] != 0: break
      k += 1; u = v; d[u] = −k
   # Če je v < 0, to pomeni, da se je pot ustavila, ker premik naprej iz polja u
   # ni mogoč. Polja na poti (od z do u) dobijo dolžine poti od k do 1.
   if v < 0: d[z] = k; c = 0
   # Če je v > 0, to pomeni, da je pot dosegla polje v, za katero že
   # poznamo pravo vrednost d[v]. Polja na poti od z do u dobijo
   # dolžine poti od d[v] + k do d[v] + 1.
   elif d[v] > 0: d[z] = k + d[v]; c = 0
   # Sicer je v < 0, kar pomeni, da se naša pot zacikla: iz u se pot nadaljuje na v,
   # ki pa smo jo že obiskali nekoč prej, namreč d[v] − d[u] korakov prej. Cikel torej
   # pokriva c = d[v] − d[u] + 1 polj. Polja na poti od z do v dobijo dolžine poti
   # od k do c, vsa ostala polja na poti (od v do u) pa dobijo dolžino poti c.
   else: d[z] = k; c = d[v] − d[u] + 1
   # Pojdimo še enkrat po poti od z naprej in vpisujmo v d dolžine, ki jih zdaj poznamo.
   u = z
   for i in range(k − 1):
      v = Nasl(u); d[v] = max(d[u] − 1, c); u = v
return max(d)
```

4. Domače naloge

Učinkovitost posameznega dne je enaka težavnosti ene od nalog; in vsaka naloga lahko vpliva le na učinkovitost enega dne (namreč tisti dan, ko jo rešujemo); učinkovitost po d dneh torej ne more biti večja od vsote težavnosti najtežjih d nalog. To zgornjo mejo pa tudi res lahko dosežemo, če razporedimo naloge med dni tako, da vsak dan dobi eno od najtežjih d nalog. Za ostale naloge je potem vseeno, kako jih razporedimo med dneve. Če na primer v vhodnem zaporedju $t_1, \ldots, t_n$ nastopi d najtežjih nalog na indeksih $m_1 < m_2 < \ldots < m_d$, lahko prvi dan rešimo naloge od 1 do $m_2 - 1$, drugi dan naloge od m_2 do $m_3 - 1$ in tako naprej, predzadnji dan rešimo naloge od m_{d-1} do $m_d - 1$, zadnji dan pa naloge od m_d do n.

Poseben primer nastopi, če je $d > n$, torej če je dni več kot nalog. Takrat je pač najbolje v n dneh rešiti vsak dan po eno nalogo, $d - n$ dni pa lahko lenarimo.

Vprašanje je zdaj le še to, kako poiskati d največjih števil v seznamu $t_1, \ldots, t_n$. Če je d majhen, gremo lahko preprosto z zanko po seznamu in sproti vzdržujemo spisek največjih d doslej prebranih elementov; vsakič ko preberemo nov element iz seznama,

ga primerjamo z najmanjšim izmed d doslej največjih in slednjega zamenjamo z novim, če je le-ta večji. Zapišimo ta postopek s psevdokodo:

$M :=$ prazen seznam;
for $i := 1$ **to** n:
  if ima M manj kot d elementov:
    dodaj nalogo i v M;
  else if je naloga i težja od najlažje naloge v M:
    najlažjo nalogo v M zamenjaj z nalogo i;

Koristno je, če vzdržujemo podatek o tem, kje v seznamu M je trenutno najlažja naloga (izmed tistih, ki so v M); tako jo bomo imeli pri roki, ko jo bo treba primerjati z novo nalogo i. Ko pa najlažjo nalogo v M zamenjamo z i, se bomo morali sprehoditi po seznamu M, da bomo ugotovili, katera je zdaj najlažja naloga v njem; to vzame $O(d)$ časa in se v najslabšem primeru lahko zgodi pri vsakem i (npr. če je vhodno zaporedje $t_1, \ldots, t_n$ naraščajoče), zato ima ta rešitev časovno zahtevnost $O(n \cdot d)$.[1]

Če za vzdrževanje najtežjih d doslej prebranih nalog namesto v seznama (kot je M zgoraj) uporabimo kopico, rdeče-črno drevo ali kaj podobnega, lahko časovno zahtevnost zmanjšamo na samo $O(n \log d)$. Še boljša rešitev je postopek quickselect, ki poišče največjih d elementov v samo $O(n)$ časa (v C++ovi standardni knjižnici lahko v ta namen uporabimo funkcijo nth_element).

5. Razbijanje permutacijske šifre

Preprosta rešitev je, da pošljemo napravi niz, ki ima na vseh mestih ničle, razen na enem mestu, kjer je enica. Tudi na izhodu bo torej niz z eno enico (drugod pa bodo ničle). Če smo recimo na vhodu poslali enico kot i-ti znak niza, na izhodu pa smo jo dobili kot j-ti znak, zdaj vemo, da šifrirnik preslika i-ti vhod v j-ti izhod. Ta poskus izvedimo n-krat, vsakič z drugim i-jem, pa bomo za vsak vhod vedeli, v kateri izhod ga naprava preslika. (Pravzaprav vemo vse že po $n - 1$ poskusih, saj potem za edini še preostali vhod ne more biti drugače, kot da se preslika na edini še preostali izhod.)

To rešitev lahko poskusimo na razne načine še izboljšati; na primer, ker smemo uporabljati znake od 0 do $b-1$ namesto le 0 in 1, lahko v enem poskusu pošljemo niz, ki ima na enem mestu znak z vrednostjo 1, na naslednjem mestu znak z vrednostjo 2 in tako naprej do $b-1$, drugod pa ima ničle; tako bomo lahko za $b-1$ vhodov naenkrat ugotovili, v katere izhode se preslikajo. Tako bomo za razbijanje šifre potrebovali približno $n/(b-1)$ poskusov namesto n poskusov.

Še boljša rešitev pa je naslednja. Če pošljemo na vhod d nizov, dolgih po n znakov, si jih lahko predstavljamo kot zapisane v tabelo z n stolpci in d vrsticami. Vse, kar šifrirnik naredi, je to, da premeša stolpce tabele. Če hočemo biti zmožni ugotoviti, kateri stolpec je prišel kam, morajo biti vsi stolpci različni med seboj. Ker imajo stolpci po d znakov in za vsak znak je b možnosti, je možnih b^d različnih stolpcev. Vzeti moramo torej dovolj velik d, da bo $b^d \geq n$, torej $d \geq \log_b n$, torej $d = \lceil \log_b n \rceil$.

Za stolpce zdaj lahko vzamemo poljubne nize, samo da so vsi različni; še posebej elegantno je, če vzamemo kar zapis števil od 0 do $n - 1$ v b-iškem sestavu (pri številih, ki imajo manj kot d števk, vrinimo na začetku ničle). Pokličimo torej kodirnik d-krat, in sicer pri i-tem klicu na j-tem vhodu podajmo i-to števko števila j v b-iškem zapisu. Nato moramo za vsak izhod le stakniti skupaj števke, ki jih je šifrirnik dajal na tem izhodu pri vseh d klicih, pa dobimo številko pripadajočega vhoda (v b-iškem zapisu).

[1] Koristno bi bilo pregledovati elemente vhodnega zaporedja v naključnem vrstnem redu; potem velja, da več nalog ko smo že pregledali, težje so zdaj naloge v M in manj verjetno je, da bi bila naloga i težja od najlažje naloge v M; zato prihaja do sprememb v M vse redkeje. Pričakovano število takih sprememb je potem le $O(d \ln(n/d))$, kar je (če je število dni d majhno v primerjavi s številom nalog n) precej manj od $O(n)$, ki smo jih imeli zgoraj v najslabšem primeru (pri naraščajočem vhodnem zaporedju). S tem prijemom smo se že srečali pri 4. nalogi za prvo skupino leta 2003; za več o tem glej str. 550 v zbirki *Rešene naloge s srednješolskih računalniških tekmovanj 1988–2004*.

15. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

24. januarja 2020

NASVETI ZA MENTORJE O IZVEDBI TEKMOVANJA IN OCENJEVANJU

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

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

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

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

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

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

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

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

1. PINi

- Neučinkovite rešitve, ki za vsako šifro *abcd* pregledujejo vse možne kombinacije *wxyz* (tako kot naša prva rešitev), naj dobijo največ 13 točk, če so drugače pravilne.

- Format izpisa pri tej nalogi ni posebej predpisan, pomembno je le, da je iz njega razvidno, katerih šifer *abcd* po postopku iz naloge ni mogoče dobiti. Za morebitne drobne napake pri izpisu (npr. če manjkajo vodilne ničle, tako da se `0042` pomotoma izpiše kot `42`) naj se odšteje največ dve točki.

- Naši rešitvi imata po štiri gnezdene zanke, ki gredo z w, x, y in z od 0 do 9; enako dobro bi bilo tudi, če bi šli z eno zanko od 0 do 9999 in iz tega števca izluščili števke w, x, y in z z deljenjem.
- Za morebitne ne-elegantnosti pri izračunu operacije $\oplus$ naj se rešitvi ne odšteva točk, če je izračun sicer pravilen. Na primer, namesto `b = ((w + x) % 11) % 10` bi lahko naredili `b = (w + x) % 11; if (b == 10) b = 0` in podobno.

2. INTERCAL

- Pri tej nalogi ni poudarek na branju vhodnih podatkov ali izpisu rezultatov, zato je vseeno, kako rešitev počne te stvari. Kot pravi že besedilo naloge, lahko rešitev tudi predpostavi, da dobi ukaze podane v tabeli ali seznamu.
- Naša rešitev se nič ne ozira na obstoječe prefikse „DO“, „PLEASE“ ali „PLEASE DO“ v vhodnih podatkih, ampak jih preprosto poreže in namesto njih vrine nove, ki poskrbijo, da je delež vljudnih ukazov primeren. Nič pa ni narobe, če poskuša tekmovalčeva rešitev uporabiti obstoječe prefikse v ukazih (in npr. popraviti le minimalno število ukazov, da bo delež vljudnih ukazov prišel v predpisane meje), dokler je rezultat v skladu z zahtevami naloge.
- Naša rešitev poskuša pazljivo brisati prefikse od daljših proti krajšim in ko odkrije pravega, z brisanjem preneha. Toda ker naloga pravi, da se te fraze kasneje v ukazih ne pojavljajo več, ni nič narobe, če poskuša tekmovalčeva rešitev brisati bolj agresivno, npr. če po tistem, ko je že pobrisala prefiks „DO“, poskuša vseeno pobrisati še prefiks „PLEASE“; ali pa celo, če poskuša pobrisati vse pojavitve podnizov „DO“ in „PLEASE“ v ukazu.
- Rešitvam, ki ne preverijo možnosti, da je problem nerešljiv (ker sta a in b tako blizu skupaj, da pri nobenem številu vljudnih ukazov ne dobimo deleža vljudnih ukazov na intervalu $[a, b]$), naj se zaradi tega odšteje pet točk.
- Rešitve, ki bi delovale le za primer iz besedila naloge, ali pa le za programe neke fiksne majhne dolžine (npr. pet ukazov, ker ima primer v nalogi le pet ukazov), naj dobijo največ 5 točk.

3. Najdaljša pot

- Pri tej nalogi je poudarek bolj na pravilnosti kot na učinkovitosti. Rešitev s časovno zahtevnostjo $O(w^2h^2)$, če je w širina in h višina mreže, lahko dobi največ 18 točk (če je drugače pravilna), rešitev s časovno zahtevnostjo $O(wh)$ pa vseh 20 točk.
- Kot pravi že besedilo naloge, naj se rešitvam, ki izračunajo dolžino poti le za en začetni položaj (namesto minimuma po vseh možnih začetnih položajih), zaradi tega odšteje 6 točk.
- Iz primerov v besedilu naloge je razvidno, da si pod „dolžino“ poti tu predstavljamo število obiskanih polj, ne števila premikov (ki je za 1 manjše od števila obiskanih polj). Če tekmovalčeva rešitev kot dolžino poti vzame število premikov, naj se ji zaradi tega odšteje največ 2 točki.
- Naš primer rešitve računa premike s pomočjo tabel DX in DY, enako dobro pa je tudi, če ima rešitev za vsako možno smer po en stavek **if**.
- Naloga pravi, da je plošča pravokotna, primer na sliki pa ima kvadratno mrežo. Če tekmovalčeva rešitev predpostavi, da je mreža vedno kvadratna, naj se ji zaradi tega odšteje največ 2 točki.

4. Domače naloge

- Poudarek pri tej nalogi je na ideji, da je treba poiskati d najtežjih nalog in vsako razporediti v svoj dan, ne pa toliko na učinkoviti implementaciji. Dovolj dobro je že, če rešitev uredi vhodno zaporedje $t_1, \ldots, t_n$, da določi najtežjih d nalog. Rešitev se sme tudi sklicevati na algoritme in podatkovne strukture, kakršne najdemo v standardni knjižnici kakšnega programskega jezika.

- Če ima rešitev težave v primeru, ko je $d > n$ (več dni kot nalog), naj se ji zaradi tega odšteje največ štiri točke.

- Rešitvam, ki lahko vračajo napačne rezultate, če ima več nalog enako težavnost, naj se zaradi tega odšteje največ štiri točke. (Na primer: recimo, da imamo $d = 3$ in zaporedje težavnosti $7, 7, 8, 8$. Kakšna neprevidna rešitev bi lahko ugotovila, da ima d-ta najtežja naloga težavnost 7, in pomislila, da lahko torej razdeli zaporedje tako, da poišče prvih $d-1$ nalog s težavnostjo vsaj 7 in zaporedje prereže po vsaki od njih; tako nastanejo trije dnevi: $[7], [7], [8, 8]$. Ta rešitev je napačna, saj obstaja boljša razdelitev: $[7, 7], [8], [8]$.)

- Ni pa mišljeno, da bi se rešitev kaj ukvarjala s tem, da pride pri predstavitvi realnih števil v računalniku do raznih zaokrožitvenih napak (in da bi mogoče lahko dve težavnosti šteli za enaki že, če se razlikujeta za neko dovolj majhno vrednost).

5. Razbijanje permutacijske šifre

- Rešitve, ki pošljejo napravi $O(n)$ nizov, lahko dobijo največ 15 točk (če so drugače pravilne). Če pošlje napravi manj kot $O(n)$ nizov, lahko dobi vse točke, četudi porabi npr. $O(n^2)$ časa za obdelavo rezultatov (npr. ker za vsakega od n izhodov pregleda vse možne vhode, da ugotovi, kateri vhod se je preslikal vanj).

- Pri tej nalogi ni mišljeno, da bi morali vse vhodne nize pripraviti vnaprej, še preden jih začnemo pošiljati šifrirni napravi. Rešitev sme na primer sestaviti nek vhodni niz, ga poslati šifrirni napravi, potem pa nekako upoštevati od nje prejeti izhodni niz pri sestavljanju naslednjega vhodnega niza. (Kakšne velike koristi od tega sicer ni.)

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. PINi	lažja naloga v prvi skupini
2. INTERCAL	srednje težka naloga v prvi ali lažja v drugi skupini
3. Najd. pot	težja naloga v prvi ali lažja v drugi skupini
4. Dom. nal.	težka naloga v prvi ali srednje težka v drugi skupini
5. Razb. šifre	težja naloga 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.