18. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

27. januarja 2023

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.");
   }
}
```

18. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

27. januarja 2023

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. Cikcakasti nizi

Recimo, da razdelimo niz znakov na krajše kose tako, da ga prerežemo povsod tam, kjer sta si dva sosednja znaka različna. Na primer:

`aabbacccaab`	$\rightarrow$	`aa` | `bb` | `a` | `ccc` | `aa` | `b`
`ggghddddoouuug`	$\rightarrow$	`ggg` | `h` | `dddd` | `oo` | `uuu` | `g`

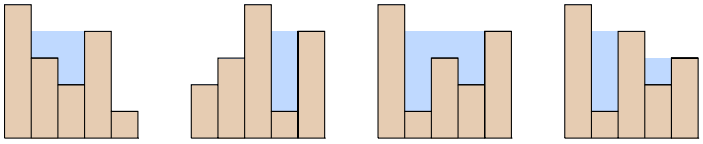
Označimo dolžine teh kosov po vrsti z d_1, d_2, d_3 in tako naprej. Rekli bomo, da je niz *cikcakast*, če te dolžine izmenično naraščajo in padajo; z drugimi besedami, če velja $d_1 < d_2 > d_3 < \ldots$ ali pa $d_1 > d_2 < d_3 > \ldots$.

Napiši program ali podprogram (funkcijo), ki kot vhod dobi niz znakov in ugotovi, ali je niz cikcakast. Niz lahko dobi kot parameter ali pa ga prebere s standardnega vhoda ali iz datoteke (karkoli ti je lažje). Predpostavi, da v nizu nastopajo le male črke angleške abecede (od `a` do `z`), lahko pa je zelo dolg, zato naj bo tvoja rešitev čim bolj učinkovita.

Nekaj primerov: nizi `aabaaa`, `cbbbabb` in `abbcdda` so cikcakasti; nizi `aabbccc`, `abbaaa` in `cdddccdcc` pa niso cikcakasti.

2. Histogram

Histogram je zaporedje n stolpcev, ki so različno visoki, vendar enako široki (po 1 enoto) in se držijo skupaj. Če od zgoraj na histogram pada dež, se zato nabira voda tam, kjer stoji več nižjih stolpcev med dvema višjima. Naslednja slika kaže štiri primere histogramov za $n = 5$:



Vidimo lahko, da je količina vode, ki se nabere v histogramu, lahko različna v odvisnosti od oblike histograma. Pri prvih dveh histogramih na gornji sliki se je nabralo po 3 enote vode, pri tretjem histogramu 6 enot vode, pri četrtem histogramu pa 4 enote vode.

Napiši program ali podprogram (funkcijo), ki kot vhod dobi zaporedje višin stolpcev in izračuna skupno količino vode, ki se nabere v takem histogramu. Višine lahko prebereš s standardnega vhoda ali iz datoteke ali pa predpostaviš, da jih dobiš kot parameter v nekem seznamu, tabeli, vektorju ali čem podobnem (karkoli ti je lažje). Višine stolpcev so naravna števila in nobena dva stolpca nista enako visoka.

Pozor: čeprav so zgoraj primeri histogramov s 5 stolpci, mora tvoja rešitev delovati za poljubno število stolpcev (poljuben n)!

3. Naredimo hitro testiranje zares hitro!

Veliko ljudi se zanima za hitro testiranje na COVID-19, zato so te organizatorji prosili, da jim pomagaš izračunati, koliko točk za testiranje potrebujejo. Testirati začnemo ob 7:00, testirati se želi n ljudi, testiranje posameznega človeka pa traja po t sekund. Vsak človek je podal tudi omejitev, do kdaj hoče zaključiti s testiranjem: testiranje i-tega človeka (za $i = 1, 2, \ldots, n$) mora biti končano najkasneje ob času $h_i : m_i$ (v urah in minutah). **Opiši postopek** (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki izračuna, najmanj koliko točk za testiranje potrebujemo, da zadostimo vsem omejitvam. Kot vhodne podatke tvoj postopek dobi n (število ljudi), t (čas testiranja v sekundah) in omejitve $h_1 : m_1, \ldots, h_n : m_n$. Podrobnosti tega, v kakšni obliki dobi te podatke, si izberi sam in jih v svoji rešitvi tudi opiši. Predpostavi, da so vhodni podatki taki, da rešitev gotovo obstaja. Dobro tudi **utemelji**, zakaj naj bi bil rezultat, ki ga tvoj postopek izračuna, pravilen.

Primer: recimo, da imamo $n = 3$ ljudi, da testiranje enega človeka traja $t = 220$ sekund in da želi biti prvi človek gotov s testiranjem do 7:08, drugi do 7:04 in tretji do 7:05. Potem potrebujemo vsaj dve točki za testiranje (ki obe ob 7:00 začneta s testiranjem, ena z drugim človekom in ena s tretjim, zatem pa ena od njiju testira še prvega človeka); z eno samo točko za testiranje ne bi šlo (ne glede na to, v kakšnem vrstnem redu bi testirali te tri ljudi, gotovo vsaj kakšen od njih ne bi bil testiran tako kmalu, kot je hotel).

4. Palindromi

Palindrom je niz, ki se ne spremeni, če njegove znake preberemo z desne proti levi namesto z leve proti desni, na primer *radar* ali *neradodaren*. **Napiši program ali podprogram** (funkcijo), ki za dani niz prešteje, koliko njegovih podnizov, dolgih vsaj 2 znaka, je palindromov. Niz lahko dobiš kot parameter ali pa ga prebereš s standardnega vhoda ali iz datoteke (karkoli ti je lažje). Predpostavi, da je niz sestavljen le iz malih črk angleške abecede. Zaželeno je, da je tvoj postopek učinkovit, tako da bo deloval hitro tudi za dolge nize (npr. nekaj deset tisoč znakov).

Primer: pri nizu `abccbbba` je pravilni odgovor 5. Palindromni podnizi dolžine vsaj 2 znaka so v tem primeru naslednji: `abccbbba`, `abccbbba`, `abccbbba`, `abccbbba`, `abccbbba`.

5. Čarobne jame

Henrik raziskuje sistem jam v računalniški igrici. Sistem je sestavljen iz n jam, ki so oštevilčene s celimi števili 1 do n. Henrik začne svojo pot v neki konkretni začetni jami s, priti pa želi do končne jame t. Jame so povezane z m prehodi, pri čemer vsak prehod neposredno povezuje dve jami; Henrik se lahko po prehodih premika v obe smeri, kolikorkrat želi. V nekaterih jamah je tudi skrinja z magičnim zvitkom, ki se ob odprtju skrinje takoj uniči, hkrati pa tudi odpre prehod med dvema drugima jamama (ti dve jami nista nujno povezani z jamo, v kateri je zvitek našel).

Opiši postopek (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki za dane podatke o jamah, prehodih in zvitkih ugotovi, ali je sploh mogoče priti od jame s do jame t. Poleg tega tudi oceni časovno zahtevnost svojega postopka oz. približno število operacij, ki jih potrebuje, da pride do rezultata (v odvisnosti od števila jam n in števila prehodov m).

Poleg števil n (število jam), m (število prehodov v začetnem stanju sistema, pred uporabo magičnih zvitkov), s (številka začetne jame) in t (številka končne jame) dobiš kot vhodne podatke še štiri tabele oz. sezname a, b, c in d, ki povedo naslednje: i-ti prehod (za $i = 1, \ldots, m$) neposredno povezuje jami $a[i]$ in $b[i]$; v i-ti jami (za $i = 1, \ldots, m$) je skrinja z zvitkom, ki ustvari prehod med jamama $c[i]$ in $d[i]$ (če i-ta jama sploh ne vsebuje skrinje z zvitkom, bo $c[i] = d[i] = -1$).

18. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

27. januarja 2023

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Cikcakasti nizi

Vhodni niz lahko pregledujemo v zanki, znak za znakom. Zapomnimo si prvi znak in poglejmo, koliko enakih znakov še pride za njim; tako dobimo dolžino prvega kosa. Nato naredimo podobno od naslednjega znaka naprej in dobimo dolžino drugega kosa; in tako dalje. Dolžine zadnjih treh kosov si zapomnimo v spremenljivkah, recimo d_1, d_2 in d_3. Zanje mora veljati bodisi $d_1 < d_2 > d_3$ bodisi $d_1 > d_2 < d_3$, sicer niz ni cikcakast (in lahko s pregledovanjem niza takoj končamo). Če pridemo do konca niza, ne da bi preverjanje tega pogoja kdaj spodletelo, lahko zaključimo, da niz je cikcakast. Paziti moramo še na robni primer: če ima niz natanko dva enako dolga kosa, bo glavna zanka prišla do konca, niz pa v resnici ni cikcakast. Zato na koncu preverimo, če sta d_1 in d_2 različna ali pa če je $d_1 = 0$ — to slednje poskrbi, da bomo kot cikcakaste pravilno prepoznali tudi nize z le enim kosom ali celo z nobenim (torej prazen niz).[1] Oglejmo si implementacijo te rešitve v jeziku C++:

```
bool JeCikcakast(const char *s)
{
  int d1 = 0, d2 = 0; // Dolžina prejšnjih dveh kosov.
  while (*s) // Preglejmo niz vse do konca.
  {
    char c = *s++; // Iz tega znaka je sestavljen trenutni kos.
    // Poglejmo, kolikokrat se znak c zdaj še ponovi.
    int d3 = 1; while (*s == c) ++s, ++d3;
    if (d1 > 0) // Ali smo že videli vsaj tri kose?
      // Preverimo, ali dolžine zadnjih treh kosov ustrezajo pogojem.
      if (! (d1 < d2 && d2 > d3) && ! (d1 > d2 && d2 < d3)) return false;
    d1 = d2; d2 = d3; // Zapomnimo si dolžini zadnjih dveh kosov.
  }
  return d1 == 0 || d1 != d2; // Če je do konca niza vse v redu, je niz cikcakast.
}
```

Lahko bi tudi brali niz sproti s standardnega vhoda, šteli zaporedne enake znake in preverjali, ali je niz cikcakast; tako sploh ni nujno imeti celotnega niza hkrati v glavnem pomnilniku:

```
bool JeCikcakast2()
{
  int d1 = 0, d2 = 0; // Dolžina prejšnjih dveh kosov.
  int c = getchar(); // Preberimo prvi znak.
  while (c >= 'a' && c <= 'z') // Preglejmo niz vse do konca.
  {
    // Naslednji znak je c; kolikokrat se še ponovi?
    int d3 = 1, cc; while ((cc = getchar()) == c) ++d3;
    if (d1 != 0) // Ali smo že videli vsaj tri kose?
      // Preverimo, ali dolžine zadnjih treh kosov ustrezajo pogojem.
      if (! (d1 < d2 && d2 > d3) && ! (d1 > d2 && d2 < d3)) return false;
    d1 = d2; d2 = d3; c = cc; // Pripravimo se na naslednji kos.
  }
  return d1 == 0 || d1 != d2; // Če je do konca niza vse v redu, je niz cikcakast.
}
```

[1] Ta robni primer je v prvotni različici naše rešitve te naloge manjkal, zato je pomotoma razglasila za cikcakaste tudi nize, sestavljene iz dveh enakih kosov. Za napako se opravičujemo.

Zapišimo prvo od gornjih dveh rešitev še v pythonu:

```
def JeCikcakast(s):
    d1 = 0; d2 = 0 # Dolžina prejšnjih dveh kosov.
    i = 0; n = len(s)
    while i < n: # Preglejmo niz vse do konca.
        c = s[i]; i += 1; d3 = 1 # Trenutni kos je iz znaka c.
        # Poglejmo, kolikokrat se znak c zdaj še ponovi.
        while i < n and s[i] == c: i += 1; d3 += 1
        if d1 > 0: # Ali smo že videli vsaj tri kose?
            # Preverimo, ali dolžine zadnjih treh kosov ustrezajo pogojem.
            if not (d1 < d2 > d3 or d1 > d2 < d3): return False
        d1 = d2; d2 = d3 # Zapomnimo si dolžini zadnjih dveh kosov.
    return d1 == 0 or d1 != d2 # Če je do konca niza vse v redu, je niz cikcakast.
```

Malo za šalo, malo zares pa je tu še rešitev za ljubitelje regularnih izrazov:

```
import re
def JeCikcakast(s):
    d1 = 0; d2 = 0 # Dolžina prejšnjih dveh kosov.
    # Preglejmo vse kose niza.
    for kos in re.finditer(r"(.)\1*", s):
        d3 = kos.end() - kos.start() # Dolžina trenutnega kosa.
        # Če smo že videli vsaj tri kose, preverimo, ali dolžine zadnjih
        # treh kosov ustrezajo pogojem.
        if d1 > 0 and not (d1 < d2 > d3 or d1 > d2 < d3): return False
        d1 = d2; d2 = d3 # Zapomnimo si dolžini zadnjih dveh kosov.
    return d1 == 0 or d1 != d2 # Če je do konca niza vse v redu, je niz cikcakast.
```

V regularnem izrazu se pika lahko ujame s poljubnim znakom, ker pa smo jo dali v oklepaje, se lahko kasneje na ta znak sklicujemo z `\1`; na koncu smo dali še zvezdico, tako da naš izraz pobere vse nadaljnje pojavitve istega znaka — to pa je ravno en kos, na kakršne hočemo pri tej nalogi razbiti vhodni niz. Ta regularni izraz je eleganten in rešitev deluje pravilno, vendar je imela pri naših poskusih to slabost, da je porabila približno 160- do 180-krat toliko dodatnega pomnilnika, kolikor je dolg posamezni kos v nizu. Ko je bil na primer s sestavljen iz 5 milijonov samih `a`-jev, je program porabil kar 930 MB pomnilnika. Ta potrata je verjetno povezana s tem, kako je v knjižnici za regularne izraze implementirano sklicevanje nazaj (pri `\1`).

Naš naslednji poskus je bil regularni izraz, ki uporablja sklicevanje nazaj le zato, da določi konec kosa:

```
    for kos in re.finditer(r"(.).*?(?!\1)", s):
```

Tu torej za prvim znakom, ki si ga zapomnimo (zato je pika v oklepajih), poberemo še poljubne nadaljnje znake, vendar čim manj (zato `*?` namesto le `*`), ustavimo pa se na mestu, kjer se nadaljevanje niza *ne ujema* z izrazom znotraj `(?!...)`. V našem primeru imamo tam `\1`, torej se ustavimo ravno takrat, ko naslednji znak ne sodi več v trenutni kos. Pri naših poskusih s tem izrazom je bila poraba dodatnega pomnilnika neodvisna od dolžine niza s, kar je dobro; je pa bil ta izraz še vedno razmeroma počasen: na nizu, sestavljenem iz 10^9 znakov `a`, se je program izvajal okrog 15 sekund.

Da se izognemo sklicevanju nazaj, zaradi katerega je iskanje pojavitev regularnega izraza počasnejše, se lahko opremo na dejstvo, da vnaprej poznamo vse možne znake, ki se utegnejo pojaviti v naših nizih, namreč male črke od `a` do `z`. Zato lahko sestavimo še preprostejši, a še manj eleganten izraz: `a+|b+|c+|...|z+`, kjer so eksplicitno naštete vse možne oblike kosov.

```
    import re, string
    ⋮
    for kos in re.finditer("+|".join(string.ascii_lowercase) + "+", s):
```

Pri tem izrazu je bila ne le poraba dodatnega pomnilnika neodvisna od dolžine niza s, pač pa je bilo tudi naštevanje kosov veliko hitrejše; niz, sestavljen iz 10^9 znakov `a`, je program s tem izrazom obdelal v manj kot pol sekunde. Nauk te zgodbe je torej, da je potrebne pri sestavljanju regularnih izrazov nekaj pazljivosti.

2. Histogram

Označimo višine stolpcev od leve proti desni s $h_1, h_2, \ldots, h_n$. Do katere višine potem pride voda nad stolpcem i? Naj bo $L_i := \max\{h_1, h_2, \ldots, h_i\}$ višina najvišjega stolpca levo od i-tega (vključno z njim samim) in podobno $D_i := \max\{h_i, h_{i+1}, \ldots, h_n\}$ višina najvišjega stolpca desno od i-tega. Potem gladina vode na stolpcu i pride ravno do višine $g_i := \min\{L_i, D_i\}$; dokler je namreč voda nižja od te višine, jo z leve strani zadržuje neki stolpec višine L_i, z desne pa neki stolpec višine D_i in bo zato rasla še naprej; ko pa doseže to višino, je vsaj na eni strani ne zadržuje nič več in se lahko tam razlije čez rob najvišjega stolpca v tisti smeri in sčasoma odteče s histograma. Tako torej vidimo, da se (za vsak i) na stolpcu višine h_i nabere voda do višine g_i, kar pomeni $g_i - h_i$ enot vode; to moramo sešteti po vseh i (od 1 do n), pa dobimo odgovor, po katerem sprašuje naloga.

Oglejmo si implementacijo te rešitve v C++. Vrednosti L_i lahko računamo v zanki od leve proti desni, saj velja zveza $L_i = \max\{L_{i-1}, h_i\}$; podobno lahko računamo D_i v zanki od desne proti levi. Ko imamo oboje, pa gremo lahko še enkrat po stolpcih, računamo količino vode in jo seštevamo.

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

int Histogram(const vector<int> &h)
{
  // Izračunajmo za vsak stolpec višino najvišjega stolpca levo in desno od njega.
  int n = h.size(); vector<int> L(n), D(n);
  for (int i = 0; i < n; ++i) L[i] = max(h[i], (i == 0) ? 0 : L[i - 1]);
  for (int i = n - 1; i >= 0; --i) D[i] = max(h[i], (i == n - 1) ? 0 : D[i + 1]);
  // Zdaj lahko za vsak stolpec določimo višino vode.
  int voda = 0;
  for (int i = 0; i < n; ++i) voda += min(L[i], D[i]) - h[i];
  return voda;
}
```

Ali v pythonu:

```
def Histogram(h):
  # Izračunajmo za vsak stolpec višino najvišjega stolpca levo in desno od njega.
  n = len(h); L = [0] * n; D = [0] * n
  for i in range(n): L[i] = max(h[i], 0 if i == 0 else L[i - 1])
  for i in range(n - 1, -1, -1): D[i] = max(h[i], 0 if i == n - 1 else D[i + 1])
  # Zdaj lahko za vsak stolpec določimo višino vode.
  return sum(min(L[i], D[i]) - h[i] for i in range(n))
```

Še ena možnost pa je, da najprej poiščemo najvišji stolpec; recimo, da je to stolpec m (z višino h_m).[2] Potem za stolpce levo od njega, torej $i < m$, vemo, da najvišji stolpec levo od i ni višji od najvišjega stolpca desno od i (kajti tam je tudi najvišji stolpec sploh), torej gladino vode pri i določa L_i in ne D_i. Desno od najvišjega stolpca, pri $i > m$, pa je ravno obratno. Dovolj je torej, če vrednosti L_i izračunamo le za $i < m$ in jih tudi kar takoj uporabimo kot gladino vode pri teh i, podobno pa potem še vrednosti D_i za $i > m$. Zato nam ni treba hraniti vseh L_i in D_i v tabelah oz. vektorjih, kot je to počela prejšnja rešitev; porabimo le $O(1)$ dodatnega pomnilnika namesto $O(n)$.

[2] Če obstaja več enako visokih najvišjih stolpcev, je vseeno, katerega vzamemo za m. Naša naloga tako ali tako zagotavlja, da bodo stolpci različno visoki, torej bo najvišji stolpec en sam, vendar bi tu opisana rešitev delovala tudi v primerih, ko je lahko več stolpcev enako visokih.

```
int Histogram2(const vector<int> &h)
{
  // Naj bo m indeks najvišjega stolpca.
  int n = h.size(), m = 0, voda = 0;
  for (int i = 1; i < n; ++i) if (h[i] > h[m]) m = i;

  // Pojdimo od levega roba do najvišjega stolpca. Gladino vode določa
  // višina najvišjega doslej obiskanega stolpca.
  for (int i = 0, najvisji = 0; i < m; ++i) {
    najvisji = max(najvisji, h[i]); // zdaj je najvisji == L[i] <= D[i] == h[m]
    voda += najvisji - h[i]; }

  // Podobno velja tudi, če gremo od desnega roba do najvišjega stolpca.
  for (int i = n - 1, najvisji = 0; i > m; --i) {
    najvisji = max(najvisji, h[i]); // zdaj je najvisji == D[i] <= L[i] == h[m]
    voda += najvisji - h[i]; }

  return voda;
}
```

Še v pythonu:

```
def Histogram2(h):
  # Naj bo m indeks najvišjega stolpca.
  n = len(h); m = 0
  for i in range(n):
    if h[i] > h[m]: m = i

  # Pojdimo od levega roba do najvišjega stolpca. Gladino vode določa
  # višina najvišjega doslej obiskanega stolpca.
  najvisji = 0; voda = 0
  for i in range(m):
    najvisji = max(najvisji, h[i]) # zdaj je najvisji == L[i] <= D[i] == h[m]
    voda += najvisji - h[i]

  # Podobno velja tudi, če gremo od desnega roba do najvišjega stolpca.
  najvisji = 0
  for i in range(n - 1, m, -1):
    najvisji = max(najvisji, h[i]) # zdaj je najvisji == D[i] <= L[i] == h[m]
    voda += najvisji - h[i]

  return voda
```

3. Naredimo hitro testiranje zares hitro!

Nobene koristi ni od tega, da bi na kakšni točki za testiranje med dvema testiranjema zevala časovna vrzel, ko ne bi nekaj časa nikogar testirali; karkšen koli veljaven razpored (tak, ki upošteva vse zahteve ljudi), v katerem so take vrzeli, ostane veljaven tudi, če ljudi za vrzeljo zamaknemo nazaj po času tako daleč, da vrzel izgine.

Razpored za posamezno točko zdaj ni nič drugega kot vrstni red, v katerem bomo na njej testirali ljudi; prvega od njih se bo testiralo od 7:00 do 7:00 + t sekund, drugega od 7:00 + t sekund do 7:00 + $2t$ sekund in tako naprej. Če imamo m točk, pa je razpored za vse skupaj spet le vrstni red, v katerem bomo prvih m ljudi testirali od 7:00 do 7:00 + t sekund, naslednjih m od 7:00 + t sekund do 7:00 + $2t$ sekund in tako naprej.

Poleg tega, ker vsako testiranje traja točno t sekund, če čas $h_i : m_i$ nekega človeka pade na sredo takega t-sekundnega območja, to pomeni, da ga v tem območju ne bomo smeli testirati (ker bi se to testiranje zanj končalo že prepozno), kar je torej za nas enako, kot če bi padel njegov čas $h_i : m_i$ na začetek tega območja. V nadaljevanju torej lahko v mislih vsak čas $h_i : m_i$ zamenjamo s celim številom, ki pove, koliko celih t-sekundnih intervalov mine od sedmih zjutraj do časa $h_i{:}m_i$; recimo temu $r_i = \lfloor (60(h_i - 7) + m_i)/t \rfloor$.

Opazimo lahko tudi, da če je neki razpored ljudi med točke za testiranje veljaven (torej če ustreza vsem zahtevam) in v njem nekoč testiramo človeka i in nekoč kasneje človeka j (ne nujno na isti točki) in je $r_i > r_j$, potem bi razpored ostal veljaven tudi, če človeka i in j v razporedu zamenjamo.[3] Omejimo se torej lahko na razporede, v katerih

[3]Recimo namreč, da je bil človek i pred zamenjavo testiran do časa t_i, človek j pa do časa $t_j > t_i$;

ljudi testiramo v naraščajočem vrstnem redu glede na r_i — najprej testiramo tistega z najmanjšim r_i in tako naprej. Recimo torej, da smo ljudi uredili in oštevilčili v tem vrstnem redu, tako da je $r_1 \leq r_2 \leq \ldots \leq r_n$.

Če imamo eno samo točko, bo človek k (z omejitvijo r_k) prišel na vrsto šele kot k-ti, torej bo končal ob času k (če se testiranje začne ob času 0), kar je v redu, če je $r_k \geq k$, sicer pa ne.

Podobno v splošnem, če imamo m točk, bo človek k tudi prišel na vrsto kot k-ti, vendar to pomeni, da bo končal ob času $\lceil k/m \rceil$, ker pač v vsaki časovni enoti obdelamo m ljudi. S tem m-jem torej lahko ustrežemo vsem zahtevam, če je $r_k \geq \lceil k/m \rceil$ (za vse k), sicer pa ne.

Tako dobimo naslednji postopek: začnimo z $m = 1$ in pregledujmo ljudi (načeloma je vseeno, v kakšnem vrstnem redu jih pregledujemo; glavno je, da so oštevilčeni naraščajoče po r_k); pri vsakem človeku k poglejmo, če je $r_k \geq \lceil k/m \rceil$, in če ni, povečujmo m za 1 tako dolgo, dokler ne bo ta pogoj izpolnjen. Ker bo na koncu gotovo $m \leq n$ (z n točkami lahko testiramo vse ljudi takoj ob sedmih, kar bo gotovo rešilo problem, razen če je kakšen od r_k enak 0, takrat pa je problem tako ali tako nerešljiv), je časovna zahtevnost $O(n)$ časa za povečevanje m-ja in pregled ljudi; pred tem pa načeloma $O(n \log n)$ za urejanje, razen če uporabimo urejanje s štetjem, za kar pa gre $O(n)$ časa.[4]

Še ena možnost: opazimo lahko, da če je naloga rešljiva (ob upoštevanju vseh omejitev) z m točkami za testiranje, je gotovo rešljiva tudi z več kot m točkami, saj bo tam prišel vsak človek na vrsto še prej (ali najkasneje ob istem času) kot pri natanko m točkah. Po eni strani vemo, da je naloga gotovo rešljiva z n točkami in da gotovo ni rešljiva z 0 točkami; najmanjši m, pri katerem je naloga še rešljiva, lahko poiščemo z bisekcijo med tema dvema skrajnostma:

$m_1 := 0$; $m_2 := n$;
while $m_2 - m_1 > 1$:
  (* m_1 *točk je gotovo premalo,* m_2 *točk je gotovo dovolj.* *)
  $m := \lfloor (m_1 + m_2)/2 \rfloor$;
  if je problem rešljiv z m točkami **then** $m_2 := m$ **else** $m_1 := m$;
(* *Ko se zanka konča, vemo, da je najmanjše primerno število točk* m_2. *)

Da preverimo, ali je problem rešljiv z m točkami, pa potrebujemo še vgnezdeno zanko, ki gre po vseh ljudeh (urejenih naraščajoče po r_k) in pri vsakem preveri, ali je $r_k \geq \lceil k/m \rceil$ (ali, z drugimi besedami: ljudi razporejamo med naših m točk, vsakega na tisto točko, ki ji je trenutno dodeljenih najmanj ljudi, potem pa preverimo, če je vsakdo testiran do zahtevanega časa). Tako imamo spet rešitev s časovno zahtevnostjo $O(n \log n)$.

Do minimalnega potrebnega m lahko pridemo tudi s požrešnim postopkom, pri katerem pregledujemo ljudi po naraščajočih k in vsakega dodelimo tisti testirni točki, ki ima zaenkrat najmanj ljudi; če pa bi bil zaradi tega ta človek testiran kasneje kot ob času r_k, odpremo novo testirno točko in ga dodamo tja. V spodnji psevdokodi nam t_i predstavlja število ljudi, ki smo jih že dodelili točki i.

$m := 0$;
za vsakega človeka k (po naraščajočih vrednostih r_k):
  $i :=$ tista točka (od 1 do m), ki ima najmanjšo vrednost t_i;
  if $m = 0$ **or** $t_i + 1 > r_k$:
    $m := m + 1$; $t_m := 1$; (* *Odprimo zanj novo točko.* *)
  else: $t_i := t_i + 1$; (* *Dodelimo tega človeka točki* i. *)

Razpored, ki ga dobimo s tem postopkom, je gotovo veljaven: človeka k dodelimo obstoječi točki i le, če je pred dodajanjem tega človeka veljalo $t_i \leq r_k - 1$, tako da bo on na tej točki testiran najkasneje ob času r_k; drugače pa odpremo zanj novo točko, kjer

ker je bil razpored takrat veljaven, mora biti $t_i \leq r_i$ in $t_j \leq r_j$. Po zamenjavi je človek j testiran bolj zgodaj kot pred zamenjavo, torej je zanj razpored še vedno veljaven; človek i pa bo po zamenjavi testiran do časa t_j, ker je $\leq r_j$ in zato $< r_i$, saj je $r_i > r_j$; tako bo torej tudi človek i še vedno testiran dovolj zgoraj in razpored je res še vedno veljaven.

[4] Da bo šlo urejanje s štetjem v $O(n)$ časa, moramo prej še pogledati, če je kakšen od r_k večji od n, in takšne r_k postaviti na n. To smemo narediti, kajti pacient z $r_k > n$ nas tako ali tako nič ne omejuje: v največ n časovnih intervalih lahko testiramo vse ljudi celo na eni sami točki, tako da omejitvi $r_k = n$ ni nič težje ustreči kot omejitvi $r_k > n$.

bo torej testiran že ob času 1, kar je gotovo $\le r_k$ (saj drugače problem sploh ne bi bil rešljiv).

Prepričajmo se zdaj, da nam ta postopek vrne razpored z najmanjšim možnim m (številom testirnih točk). Pa recimo, da kdaj ne bi bilo tako; vzemimo najmanjši tak testni primer (torej takega z najmanjšim številom ljudi n), pri katerem naš požrešni postopek porabi preveč točk; on jih torej porabi m, v resnici pa je mogoče te ljudi testirati že z $m-1$ ali manj točkami. Ali je mogoče, da je naš postopek odprl m-to točko že kaj prej kot šele pri zadnjem, n-tem človeku? To bi pomenilo, da je naš postopek že na prvih $n-1$ ljudi odprl m točk, toda ker je na vseh problemih z manj kot n ljudmi naš postopek optimalen, to pomeni, da se prvih $n-1$ ljudi ne da testirati z manj kot m točkami, zato pa se potem tudi vseh n ljudi ne da testirati z manj kot m točkami; toda to je v protislovju s predpostavko, da naš požrešni razpored teh n ljudi na m točk ni optimalen. — Zdaj torej vemo, da je naš požrešni algoritem odprl m-to točko šele za n-tega človeka. Takrat je moralo biti vsaki od dotedanjih $m-1$ točk že dodeljenih vsaj r_n ljudi, kajti drugače bi lahko človeka n dodelili eni od tistih točk in bi bil še vedno testiran najkasneje do časa r_n (in potem zanj ne bi bilo treba odpirati nove točke). Vsi ti ljudje imajo $r_k \le r_n$, saj smo rekli, da gleda naš postopek ljudi po naraščajočih časih r_k. Skupaj z n-tim človekom imamo torej vsaj $(m-1)\cdot r_n + 1$ ljudi, ki morajo vsi biti testirani najkasneje ob času r_n (nekateri morda celo še prej). Za te ljudi trdi naša predpostavka, da (ker naša požrešna rešitev ni optimalna) jih je mogoče testirati na manj kot m točkah. Toda na $m-1$ točkah je mogoče v r_n korakih testirati največ $(m-1)\cdot r_n$ ljudi, torej je nemogoče, da bi taka rešitev, boljša od naše, res obstajala. □

Pri opisanem požrešnem postopku je načeloma koristne še nekaj pazljivosti, če ga hočemo učinkovito implementirati. Vzdrževali bomo $\tau := \min_i t_i$, torej minimum t_i po vseh doslej odprtih točkah, poleg tega pa bomo za vsak t vzdrževali še množico $S[t]$ vseh točk, ki jim je bilo doslej dodeljenih natanko t ljudi (torej ki imajo trenutno $t_i = t$). To nam bo pri vsakem človeku omogočilo v $O(1)$ časa izbrati točko z najmanjšo t_i. Opišimo zdaj naš postopek podrobneje:

$m := 0$; $\tau := 0$; **for** $t := 1$ **to** n **to** $S[t] := \{\}$;
za vsakega človeka k (po naraščajočih vrednostih r_k):
  if $m = 0$ **or** $\tau \ge r_k$: (* *Treba bo odpreti novo točko.* *)
    $m := m + 1$; $\tau := 1$; dodaj m v $S[1]$;
  $i :=$ poljuben element množice $S[\tau]$;
  (* *Dodelimo tega človeka točki i.* *)
  pobriši i iz $S[\tau]$ in ga dodaj v $S[\tau + 1]$;
  if je $S[\tau]$ prazna **then** $\tau := \tau + 1$;

4. Palindromi

Preprosta, a neučinkovita rešitev je, da gremo z dvema gnezdenima zankama po vseh možnih podnizih vhodnega niza, za vsak podniz pa preverimo, ali je palindrom (in če je, povečamo števec palindromov, ki ga na koncu vrnemo kot rezultat). Da preverimo, ali je podniz palindrom, moramo preveriti, če sta njegov prvi in zadnji znak enaka, če sta drugi in predzadnji znak enaka in tako naprej; premikajmo se torej po njem hkrati z dvema indeksoma, z enim od leve proti desni in z drugim od desne proti levi, ter primerjajmo znake na teh indeksih; če se indeksa srečata, ne da bi opazili kakšno neujemanje, lahko zaključimo, da je bil ta podniz palindrom. Ker moramo pregledati $O(n^2)$ podnizov in imamo pri vsakem $O(n)$ dela, ima ta rešitev časovno zahtevnost kar $O(n^3)$.

```
int Palindromi1(const string &s)
{
  int n = s.length(), stPalindromov = 0;
  for (int i = 0; i + 1 < n; i++) for (int j = i + 1; j < n; ++j)
  {
    // Preverimo, ali je s[i...j] palindrom.
    bool ok = true;
    for (int ii = i, jj = j; ii < jj; )
      if (s[ii++] != s[jj--]) { ok = false; break; }
```

```
    if (ok) ++stPalindromov;
  }
  return stPalindromov;
}
```

Ali v pythonu:

```
def Palindromi1(s):
  n = len(s); stPalindromov = 0
  if n <= 1: return 0
  for i in range(n - 1):
    for j in range(i + 1, n):
      # Preverimo, ali je s[i...j] palindrom.
      ii = i; jj = j
      while ii < jj and s[ii] == s[jj]: ii += 1; jj -= 1
      if ii >= jj: stPalindromov += 1
  return stPalindromov
```

Boljšo rešitev dobimo, če upoštevamo, da se v daljšem palindromu skrivajo krajši. Podniz $s[i]s[i+1]\ldots s[j-1]s[j]$ je lahko palindrom le, če je palindrom tudi malo krajši podniz $s[i+1]s[i+2]\ldots s[j-1]$ (in če sta poleg tega znaka $s[i]$ in $s[j]$ enaka). Takšne podnize je torej koristno pregledovati od krajših proti daljšim; čim opazimo, da nimamo več palindroma, nam daljših podnizov ni več treba gledati.

Postavimo se na primer v mislih v znak $s[i]$ in glejmo podnize s središčem v tem znaku: to so $s[i-d]\ldots s[i+d]$ za $d = 1, 2, \ldots$; če sta $s[i-1]$ in $s[i+1]$ enaka, imamo tu palindrom dolžine 3; če sta poleg tega tudi $s[i-2]$ in $s[i+2]$ enaka, imamo tu palindrom dolžine 5; in tako naprej. Čim opazimo pri kakšnem d neujemanje med $s[i-d]$ in $s[i+d]$, lahko s tem i končamo, saj potem naši podnizi tudi pri večjih d ne bodo več palindromi.

Podobno lahko obravnavamo tudi podnize sode dolžine. Če gledamo recimo podnize s središčem na meji med znakoma $s[i-1]$ in $s[i]$, so to podnizi oblike $s[i-d]\ldots s[i+d-1]$ za $d = 1, 2, \ldots$; če sta $s[i-1]$ in $s[i]$ enaka, imamo palindrom dolžine 2; če sta poleg tega tudi $s[i-2]$ in $s[i+1]$ enaka, imamo palindrom dolžine 4; in tako naprej.

Zdaj potrebujemo torej le dve gnezdeni zanki, eno po i (središče podniza) in eno po d (dolžina podniza). Pri vsakem povečanju d-ja imamo le $O(1)$ dela, da preverimo, ali se znaka, ki smo ju na levem in desnem koncu dodali v podniz, ujemata; tako imamo rešitev s časovno zahtevnostjo $O(n^2)$.[5]

```
int Palindromi2(const string &s)
{
  int n = s.length(), stPalindromov = 0;
  for (int i = 1; i < n; ++i)
  {
    // Preštejmo palindrome lihe dolžine (vsaj 3) s središčem v s[i].
    L = i - 1; D = i + 1;
    while (L >= 0 && D < n && s[L] == s[D])
      ++stPalindromov, --L, ++D;
    // Preštejmo palindrome sode dolžine (vsaj 2) s središčem med s[i - 1] in s[i].
    int L = i - 1, D = i;
    while (L >= 0 && D < n && s[L] == s[D])
      ++stPalindromov, --L, ++D;
  }
  return stPalindromov;
}
```

Ali v pythonu:

```
def Palindromi2(s):
  n = len(s); stPalindromov = 0
  if n <= 1: return 0
  for i in range(1, n):
```

[5]Za namene naše naloge je ta rešitev dovolj učinkovita, kot zanimivost pa omenimo, da obstaja tudi rešitev z linearno časovno zahtevnostjo, $O(n)$; gl. *Bilten RTK 2013*, str. 128–130.

```
    # Preštejmo palindrome lihe dolžine (vsaj 3) s središčem v s[i].
    L = i - 1; D = i + 1
    while L >= 0 and D < n and s[L] == s[D]:
      stPalindromov += 1; L -= 1; D += 1
    # Preštejmo palindrome sode dolžine (vsaj 2) s središčem med s[i - 1] in s[i].
    L = i - 1; D = i;
    while L >= 0 and D < n and s[L] == s[D]:
      stPalindromov += 1; L -= 1; D += 1
  return stPalindromov
```

5. Čarobne jame

Za začetek je koristno predelati naše vhodne podatke tako, da bomo imeli za vsako jamo seznam njenih *sosed* (torej tistih jam, ki so neposredno povezane z njo s prehodi); za jamo u recimo temu seznamu $S[u]$.

Sistem jam lahko pregledujemo sistematično, da ugotovimo, kam vse je mogoče priti iz s: iz s je mogoče priti v njene sosede, iz teh v njihove sosede in tako naprej. Koristno je torej vzdrževati tabelo, v kateri bomo označevali, za katere jame že vemo, da so dosegljive iz s; poleg tega bomo hranili tudi seznam Q z jamami, za katere smo že ugotovili, da so dosegljive iz s, nismo pa še pogledali, kam je mogoče potem priti naprej iz njih — to so torej jame, ki jih bomo morali še pregledati. Tako dobimo približno takšen postopek:

označi jamo s za dosegljivo in jo dodaj v seznam Q;
dokler Q ni prazen:
  naj bo u poljubna jama iz Q; pobriši jo iz Q;
  za vsako u-jevo sosedo v:
    če v še nimamo označene kot dosegljive,
      jo označi zdaj in jo dodaj v seznam Q;

Ta postopek bo sčasoma dosegel vse jame, ki jih je sploh mogoče doseči iz s; na koncu moramo torej le še preveriti, ali je med njimi tudi jama t. (Lahko ga tudi prekinemo predčasno, čim se izkaže jama t za dosegljivo.) To, v kakšnem vrstnem redu jemljemo jame iz Q, za naš namen načeloma ni pomembno; če vzamemo vedno tisto jamo, ki je že najdlje v Q, dobimo znani postopek iskanja v širino (*breadth-first search*, BFS).

Doslej še nismo upoštevali čarobnih zvitkov, ki jih omenja besedilo naloge. Recimo, da se v jami u nahaja zvitek, ki ustvari nov prehod med jamama $c[u]$ in $d[u]$. Če jama u ni dosegljiva iz s, Henrik takega zvitka tako ali tako ne bi mogel uporabiti in se zaradi njega pri dosegljivosti nič ne spremeni. Če pa u je dosegljiva, bomo sčasoma prišli do nje v glavni zanki našega postopka in lahko takrat razmislimo o tem, kaj za nas pomeni novi prehod od $c[u]$ do $d[u]$. Tu ločimo dve možnosti: (1) Če niti za $c[u]$ niti za $d[u]$ takrat še ne vemo, ali sta dosegljivi ali ne, je dovolj, če dodamo $c[u]$ na seznam sosedov jame $d[u]$ in obratno; tako bomo novi prehod primerno upoštevali v bodoče, če bomo kdaj prišli do jame $c[u]$ ali $d[u]$. (2) Če pa za vsaj eno od $c[u]$ in $d[u]$ že vemo, da je dosegljiva, potem zdaj zaradi novega prehoda vemo, da sta dosegljivi obe in ju lahko kot taki tudi označimo (če še nista) ter ju dodamo v seznam Q.

Časovna zahtevnost naše rešitve je $O(n + m)$, saj vsako jamo — teh pa je $O(n)$ — le enkrat dodamo v Q (ko jo označimo za dosegljivo), zato jo tudi le enkrat vzamemo iz Q in le enkrat pregledamo njene sosede; s pregledovanjem sosed pa imamo le toliko dela, kolikor je prehodov, torej $O(m)$.

Oglejmo si še primer implementacije tega postopka v C++, pri katerem pa bomo predpostavili, da gredo številke jam (in prehodov) od 0 naprej namesto od 1 naprej:

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

bool JeDosegljiva(int n, int m, int s, int t,
                  const vector<int> &a, const vector<int> &b,
                  const vector<int> &c, const vector<int> &d)
{
```

```
  // Pripravimo sezname sosed.
  vector<vector<int>> S(n);
  for (int i = 0; i < m; ++i) S[a[i]].emplace_back(b[i]), S[b[i]].emplace_back(a[i]);

  // Pomožni podprogram, ki označi jamo kot dosegljivo in jo doda v Q.
  vector<bool> dosegljiva(n, false); vector<int> Q;
  auto Oznaci = [&] (int u) { if (! dosegljiva[u]) dosegljiva[u] = true, Q.emplace_back(u); };

  // Na začetku vemo, da je dosegljiva jama s.
  Oznaci(s);

  // Preglejmo preostanek sistema jam.
  while (! Q.empty())
  {
    int u = Q.back(); Q.pop_back();

    // Ker je u dosegljiva, so dosegljive tudi njene sosede.
    for (int v : S[u]) Oznaci(v);

    // Upoštevajmo čarobni zvitek v jami u, ki odpre prehod med jamama cu in du.
    if (int cu = c[u], du = d[u]; cu >= 0 && du >= 0)

      // Če cu in du še nista dosegljivi, si novi prehod le zapomnimo.
      if (! dosegljiva[cu] && ! dosegljiva[du])
        S[cu].emplace_back(du), S[du].emplace_back(cu);

      // Sicer vemo, da sta dosegljivi obe in ne le ena od njiju.
      else Oznaci(cu), Oznaci(du);
  }
  return dosegljiva[t];
}
```

In še v pythonu:

```
def JeDosegljiva(n: int, m: int, s: int, t: int,
                 a: list[int], b: list[int], c: list[int], d: list[int]) -> bool:
  # Pripravimo sezname sosed.
  S = [[]] * n
  for ai, bi in zip(a, b): S[ai].append(bi); S[bi].append(ai)

  # Pomožni podprogram, ki označi jamo kot dosegljivo in jo doda v Q.
  dosegljiva = [False] * n; Q = []
  def Oznaci(u):
    if not dosegljiva[u]: dosegljiva[u] = True; Q.append(u)

  # Na začetku vemo, da je dosegljiva jama s.
  Oznaci(s)

  # Preglejmo preostanek sistema jam.
  while Q:
    u = Q.pop()

    # Ker je u dosegljiva, so dosegljive tudi njene sosede.
    for v in S[u]: Oznaci(v)

    # Upoštevajmo čarobni zvitek v jami u, ki odpre prehod med jamama cu in du.
    cu = c[u]; du = d[u]
    if cu >= 0 and du >= 0:

      # Če cu in du še nista dosegljivi, si novi prehod le zapomnimo.
      if not (dosegljiva[cu] or dosegljiva[du]):
        S[cu].append(du); S[du].append(cu)

      # Sicer vemo, da sta dosegljivi obe in ne le ena od njiju.
      else: Oznaci(cu); Oznaci(du)

  return dosegljiva[t]
```

18. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

27. januarja 2023

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. Cikcakasti nizi

- Naloga poudarja, da so nizi lahko dolgi in da naj bo rešitev učinkovita. Rešitve s časovno zahtevnostjo, slabšo od linearne, naj dobijo največ 10 točk, če so sicer pravilne.

- V eni od naših rešitev smo brali niz s standardnega vhoda znak po znak in ga sproti obdelovali. Za enako dobre naj štejejo tudi rešitve, ki preberejo celoten niz v glavni pomnilnik, nato pa ga obdelajo.

- Rešitvam, ki porabijo linearno mnogo dodatnega pomnilnika (za povrhu vhodnega niza) — npr. zato, ker si eksplicitno pripravijo celoten seznam kosov, na katere se pri tej nalogi razdeli niz, ali pa njihovih dolžin — naj se zaradi tega odšteje dve točki.

- Za manjše, nebistvene napake pri preverjanju cikcakavosti (npr. če program namesto pogoja $d_1 < d_2 > d_3$ preverja $d_1 \leq d_2 \geq d_3$) naj se odšteje največ dve točki.

2. Histogram

- Za vse točke pričakujemo rešitev s časovno zahtevnostjo $O(n)$. Rešitve s časovno zahtevnostjo $O(n^2)$ naj dobijo največ 15 točk, če so sicer pravilne. Rešitve s časovno zahtevnostjo, slabšo od $O(n^2)$, naj dobijo največ 10 točk, če so sicer pravilne. Med slednje štejejo tudi morebitne rešitve, kjer bi bila časovna zahtevnost kaj odvisna od višine stolpcev (kajti glede teh višin ne daje naloga nobenih zagotovil, razen tega, da so višine pač različna naravna števila).

- V našem opisu rešitve imamo dve različici z linearno časovno zahtevnostjo: eno, ki porabi $O(n)$ dodatnega pomnilnika (poleg vhodnega histograma), in eno ki porabi le $O(1)$ dodatnega pomnilnika. Oboje naj se šteje za enako dobro in lahko dobi vse točke.

3. Naredimo hitro testiranje zares hitro!

- Ključno pri tej nalogi je opažanje, da je smiselno paciente urediti po času, do katerega hočejo biti testirani (v naši rešitvi je ta čas označen z r_i), in jih testirati v tem vrstnem redu. Zaželeno je, da poskuša tekmovalčeva rešitev podati vsaj nekakšno utemeljitev, zakaj je to res, vendar prav veliko v tej smeri ne smemo pričakovati.

- Rešitve, ki porabijo $O(n \log n)$ časa namesto $O(n)$ časa, naj veljajo za enako dobre in lahko tudi dobijo vse točke (če so pravilne). Sem sodijo npr. rešitve, ki ljudi ne urejajo s štetjem, ali pa rešitve, ki določajo najmanjši m z bisekcijo.

- Vse točke lahko dobijo tudi rešitve, ki bi morda predpostavile, da ležijo vsi časi znotraj enega dne in je zato nabor možnih vrednosti zelo omejen, kar lahko olajša urejanje. Od rešitev se tudi ne pričakuje, da se ukvarjajo s podrobnostmi postopka za urejanje.

- Rešitve, ki bi porabile $O(n^2)$ časa (npr. ker gredo morda za vsako možno število točk m po vseh ljudeh, da preverijo, ali se dá sestaviti veljaven razpored za m testirnih točk), naj dobijo največ 15 točk, če so drugače pravilne.

- Pri tej nalogi je načeloma mogoče, da problem sploh ni rešljiv (če hoče biti kdo gotov s testiranjem že v manj kot t sekundah po začetku testiranja ob 7:00), vendar se rešitvam tekmovalcev s tem robnim primerom ni treba ukvarjati (oz. se ga sploh zavedati), saj besedilo naloge zagotavlja, da bodo vhodni podatki taki, da bo problem zagotovo rešljiv.

4. Palindromi

- Pri tej nalogi pričakujemo za vse točke rešitev s časovno zahtevnostjo $O(n^2)$. Rešitve z zahtevnostjo $O(n^3)$ naj dobijo največ 12 točk, če so sicer pravilne.

- Rešitvi, ki bi pomotoma štela tudi palindromne podnize dolžine 1 (in tako povečala število palindromov za n), naj se zaradi tega odšteje dve točki.
- Če bi kakšna rešitev pomotoma štela le palindrome lihe dolžine ali le palindrome sode dolžine, naj se ji zaradi tega odšteje pet točk.

5. Čarobne jame

- Rešitev, ki bi (npr. zaradi kakšne nespametne rekurzije) lahko porabila eksponentno mnogo časa (ali pa celo padla v neskončno zanko zaradi kakšnih ciklov v grafu ipd.), naj dobi največ 10 točk.
- Dodajanje novih povezav zaradi čarobnih zvitkov bi se dalo upoštevati tudi na kak nerodnejši način od tistega v naši rešitvi; lahko bi na primer po vsakem dodajanju povezave začeli preiskovati graf spet od začetka; ali pa bi ga preiskovali v zanki, dokler ne bi enkrat pregledali celega grafa, ne da bi pri tem dodali kakšno novo povezavo. Zaradi takih neučinkovitosti, ki povečajo časovno zahtevnost za faktor $O(n)$, naj se rešitvi odšteje 5 točk. Enako velja tudi za podobne neučinkovitosti pri predstavitvi grafa (npr. če si rešitev ne pripravi seznamov sosedov, pač pa vsakič, ko vzame naslednjo točko iz vrste, pregleda *vse* povezave v grafu; ali pa če uporablja matriko sosednosti namesto seznamov sosedov).
- Možna napaka pri tej nalogi je, da ob dodajanju nove povezave ne obravnavamo posebej primera, ko smo eno od krajišč te povezave že pregledali, drugega pa še ne; če takrat le dodamo novo povezavo, nam ne bo pomagala priti iz že pregledanega krajišča v drugo krajišče. Rešitvam, ki spregledajo ta primer, naj se zaradi tega odšteje dve točki.
- Če se kakšna rešitev za zvitke sploh ne zmeni in pregleda le prvotni sistem jam, naj dobi največ 12 točk (če vsaj to naredi pravilno).
- V naši rešitvi je tudi primer implementacije v C++, vendar še enkrat poudarimo, da naloga zahteva le opis postopka in ne nujno implementacije v kakšnem konkretnem programskem jeziku.
- Naloga pravi, naj tekmovalec tudi oceni časovno zahtevnost svoje rešitve. Tistim rešitvam, ki tega ne naredijo ali pa so v svoji oceni povsem zgrešene, naj se zaradi tega odšteje dve točki.

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. Cikcakasti nizi	lažja do srednja naloga v prvi skupini
2. Histogram	srednje težka naloga v prvi ali lažja v drugi skupini
3. Hitro testiranje	srednje težka naloga v drugi skupini
4. Palindromi	težka naloga v prvi ali srednja v drugi skupini[6]
5. Čarobne jame	težja naloga v drugi ali lahka 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.

[6] Težavnost te naloge je odvisna od tega, kako točkujemo različno učinkovite rešitve. Če bi se dalo dobiti vse točke že za rešitev v času $O(n^3)$, bi postala to morda srednje težka naloga za prvo skupino; če pa bi bilo treba za vse točke najti rešitev v času $O(n)$, bi bila to težka naloga za tretjo skupino.