12. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

20. januarja 2017

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.");
   }
}
```

12. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

20. januarja 2017

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. Umor

V Nori vasi se je zgodil umor. Detektivi so pri preiskavi ugotovili, da bi jim prišlo prav, če bi vedeli, kdo je umorjenca nazadnje videl živega. Zaslišali so vse vaščane in ugotovili, kdo se je zadnje čase s kom pogovarjal; ti podatki so že urejeni po času pogovora (od najzgodnejših do najkasnejših). Zaradi varovanja zasebnosti je vsak od vaščanov predstavljen le z začetnico svojega imena (predpostavimo, da se pri nobenih dveh vaščanih ne začneta njuni imeni na isto črko). **Napiši program**, ki prebere podatke o pogovorih med vaščani in izpiše, kdo (če sploh kdo) je umorjenca zadnji videl živega. Podatke lahko tvoj program bere s standardnega vhoda ali pa iz datoteke `umor.txt` (karkoli ti je lažje). V prvi vrstici je ime umorjenca, v drugi vrstici število pogovorov med vaščani, nato pa je v vsaki vrstici opisan po en pogovor (začetnici dveh vaščanov, ločeni s presledkom).

Primer: recimo, da imamo naslednje vhodne podatke.

```
B
5
A B
B D
D E
E A
A S
```

Potem lahko zaključimo, da je umorjenca (vaščana `B`) zadnji živega videl `D`.

2. Naraščajoče besede

Napiši program, ki prebere vhodno besedilo in izpiše dolžino najdaljše take besede (v prebranem besedilu), v kateri so črke urejene naraščajoče po abecedi. Besedilo lahko bereš s standardnega vhoda ali pa iz datoteke `besedilo.txt` (karkoli ti je lažje). Predpostaviš lahko, da nobena vrstica vhodnega besedila ni daljša od 100 znakov in da se v besedilu pojavljajo le male črke angleške abecede (od „`a`“ do „`z`“), presledki in ločila.

Primer: recimo, da imamo vhodno besedilo

```
gospod benjamin je bil z njive zagledal tujega gospoda iti proti
gradu. sel je tedaj k potu in gostu naproti. ko sta se sesla,
spoznal ga je brz. vesel ga je pozdravil in mu pomolil roko.
```

Dolžina najdaljše naraščajoče besede v njem je 5 črk (to je beseda `gostu`; druge naraščajoče besede v tem besedilu so še `bil`, `in`, `mu`, `brz` in nekaj enočrkovnih).

3. Popularni dnevi

Za nek strežnik za n zaporednih dni poznamo število obiskovalcev na posamezen dan. Rekli bomo, da je dan d *popularen*, če ima večji obisk kot povprečje njemu najbližjih k dni iz opazovanega n-dnevnega obdobja. (Pri tem d-ja samega ne štejemo med njemu najbližjih k dni.) **Opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki ugotovi, koliko je bilo popularnih dni. Predpostaviš lahko, da imaš vrednosti n in k ter tabelo obiskovalcev po dnevih že podane v nekih spremenljivkah (tako da se ti ni treba ukvarjati z branjem podatkov iz datoteke ali česa podobnega), da je k sodo število in da je k manjši od n. Zaželeno je, da je tvoj postopek čim bolj učinkovit, tako da bo deloval hitro tudi v primerih, ko je k velik.

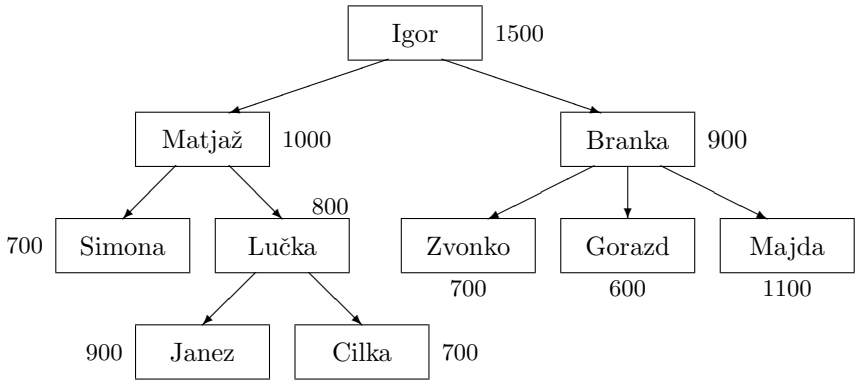
Primer: recimo, da imamo zaporedje $n = 7$ dni z naslednjim številom obiska:

$$10,\ 14,\ 17,\ 20,\ 9,\ 15,\ 14.$$

Pri $k = 2$ so popularni dnevi štirje od teh sedmih (namreč drugi, četrti, šesti in sedmi).

4. Sindikat

V nekem uspešnem slovenskem podjetju so zaposleni urejeni hierarhično. Vsak razen direktorja ima natanko enega nadrejenega. Vsak uslužbenec ima lahko pod seboj več podrejenih. Primer takšne hierarhije (številke ob imenih so njihove plače):



V tem podjetju imajo zelo močan sindikat šefov. Sindikalisti so ugotovili, da višine plač niso pravične. Nedopustno je, da imajo nekateri podrejeni višje plače od svojih nadrejenih! Zato sindikat zahteva, da mora imeti vsak zaposleni vsaj za 100 € višjo plačo od kateregakoli svojega podrejenega.

Lastnik bi rad analiziral podatke, preden se spusti v pogajanja s sindikalisti. Lastnika zanima, koliko dodatnega denarja bi potreboval vsak mesec, če bi ugodil zahtevam sindikata (in to seveda tako, da se plača nikomur ne bi znižala). Pomagaj mu in **opiši postopek** (ali napiši program ali podprogram, če ti je lažje), ki izračuna najmanjšo možno skupno vsoto denarja, ki bi ga potreboval za odpravo krivic. Poleg tega naj tudi poišče (oz. izpiše) imena zaposlenih, ki bi pri tem prejeli povišico.

Predpostaviš lahko, da so zaposleni oštevilčeni z zaporednimi številkami od 1 do n in da so podatki o njih že podani v nekaj tabelah: za zaposlenega s številko k imamo v ime[k] njegovo ime, v placa[k] njegovo plačo in v sef[k] zaporedno številko njegovega nadrejenega (če nadrejenega nima, je tu vrednost 0).

5. 3-d labirint

Dana je pravokotna karirasta mreža $w \times h$ celic. V vsaki celici stoji steber; stebri so različnih višin in te višine so podane v tabeli: $v[x][y]$ je višina stebra v celici (x, y). Po vrhovih teh stebrov se želimo sprehoditi. S stebra lahko prestopimo na sosednji steber le, če se stebra po višini razlikujeta največ za eno enoto in če se celici, v katerih tadva stebra stojita, dotikata z eno stranico. Če se celici stikata le z vogalom, prehod med njunima stebroma ni mogoč (vsaj ne neposredno). **Opiši postopek**, ki ugotovi, ali je možno priti z najnižjega stebra do najvišjega.

12. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

20. januarja 2017

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Umor

Najprej preberimo ime umorjenca in si ga zapomnimo v neki spremenljivki (v spodnji rešitvi je to na primer zrtev). Nato po vrsti berimo vrstice s podatki o pogovorih (v spremenljivki n hranimo število vrstic, ki jih še moramo prebrati — ko ta števec pade na 0, se ustavimo). Pri vsakem pogovoru pogledamo, če je eden od obeh sogovornikov kasnejši umorjenec; če je, je drugi sogovornik zadnji človek doslej, ki je umorjenca še videl živega, zato si tega sogovornika zapomnimo (v spremenljivki zadnji. Na koncu spremenljivko zadnji izpišemo. Paziti moramo še na možnost, da z umorjencem sploh ni nihče govoril; to lahko odkrijemo tako, da spremenljivko zadnji za začetku inicializiramo na nek znak, ki ni črka, in preverimo, če ima na koncu še vedno to vrednost.

```
#include <stdio.h>

int main()
{
  char zrtev, zadnji = '.', a, b;
  int n;

  scanf("%c\n", &zrtev);
  scanf("%d\n", &n);

  while (n-- > 0)
  {
    scanf("%c %c\n", &a, &b);
    if (a == zrtev) zadnji = b;
    else if (b == zrtev) zadnji = a;
  }

  if (zadnji == '.') printf("Z umorjencem ni govoril nihče.\n");
  else printf("Z umorjencem je zadnji govoril %c.\n", zadnji);
  return 0;
}
```

2. Naraščajoče besede

Vhodno besedilo lahko beremo znak po znak; poleg trenutnega znaka (v spodnji rešitvi je to spremenljivka c) si zapomnimo še prejšnjega (spremenljivka cp), tako da bomo lahko po vsakem znaku preverili, ali je trenutna beseda še urejena naraščajoče. O trenutni besedi si moramo poleg tega, ali je naraščajoča, zapomniti še njeno dolžino. Če je trenutni znak črka, povečamo dolžino in preverimo, če beseda narašča; če pa trenutni znak ni črka, pomeni, da smo na koncu besede in moramo pogledati, če je to najdaljša naraščajoča beseda doslej; če je, si njeno dolžino zapomnimo v spremenljivki naj. Zapišimo to rešitev še v C-ju:

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

int main()
{
  int naj = 0, dolzina = 0, cp = -1;
  bool narasca = true;
  do
  {
```

```
      int c = fgetc(stdin);
      if ('a' <= c && c <= 'z')
      {
        /* Smo znotraj neke besede. Povečajmo dolžino in preverimo,
           če je še vedno urejena naraščajoče. */
        dolzina++;
        if (c < cp) narasca = false;
      }
      else
      {
        /* Smo med dvema besedama. Če je bila prejšnja beseda naraščajoča
           in najdaljša taka doslej, si jo zapomnimo. */
        if (narasca && dolzina > naj) naj = dolzina;
        /* Pripravimo se na branje naslednje besede. */
        dolzina = 0; narasca = true;
      }
      cp = c; /* Zapomnimo si trenutno črko kot prejšnjo v spremenljivki cp. */
    }
    while (cp != EOF);
    printf("%d\n", naj); return 0;
}
```

3. Popularni dnevi

Recimo, da imamo naše dneve oštevilčene od 0 do $n-1$. Ko bi za nek dan d radi preverili, ali je popularen ali ne, moramo najprej ugotoviti, katerih je k njemu najbližjih dni (brez d-ja samega). Lahko si predstavljamo, da izmenično jemljemo dneve z leve in desne, dokler se nam jih ne nabere k: tako dobimo $d+1$, $d-1$, $d+2$, $d-2$, ... Načeloma bomo tako pobrali z vsake strani d-ja po $k/2$ dni, mogoče pa je tudi, da nam z ene strani zmanjka dni (na primer: če je $d < k/2$, potem pred njim v zaporedju ni $k/2$ dni), takrat pa jih bomo morali pač ustrezno več pobrati z druge strani. Spotoma lahko obisk v teh dneh tudi seštevamo in na koncu primerjamo povprečje po teh k dneh z obiskom na dan d. Če je obisk na dan d nad povprečjem, povečamo števec popularnih dni. Zapišimo ta postopek v C-ju:

```
int Popularni(int n, int k, const int obisk[])
{
  int stPopularnih = 0;
  for (int d = 0; d < n; d++)
  {
    /* Seštejmo obisk v najbližjih k dnevih. Spremenljivka s pove, koliko
       smo jih že sešteli, i pa pove, kako daleč od d-ja smo že. */
    int vsota = 0;
    for (int s = 0, i = 0; s < k; i++)
    {
      if (d - i >= 0) vsota += obisk[d - i], s++;
      if (d + i < n) vsota += obisk[d + i], s++;
    }
    if (vsota < obisk[d] * k) stPopularnih++;
  }
  return stPopularnih;
}
```

Časovna zahtevnost te rešitve je $O(n \cdot k)$, saj moramo pri vsakem od n dni iti z zanko po najbližjih k dnevih in jih seštevati. Razmislimo o tem, kako lahko rešitev še izboljšamo. Ko računamo najbližjih k dni dnevu d, načeloma nastane interval od $d-k/2$ do $d+k/2$, razen če je d blizu začetka zaporedja (pri $d < k/2$) — takrat nastane interval od 0 do k — ali pa blizu konca zaporedja (pri $d > n-1-k/2$ — takrat nastane interval od $n-k-1$ do $n-1$. (Pravzaprav ti intervali, ki smo jih pravkar zapisali, obsegajo tudi dan d sam, na kar bomo morali kasneje paziti pri izračunu povprečja.) Če se z d-jem premaknemo za en dan naprej, se tudi vsako od krajišč intervala premakne za en dan naprej ali pa ostane nespremenjeno. Iz vsote obiska po vseh dneh v intervalu zato na

levi strani lahko izpade en člen (če se levo krajišče premakne naprej), na desni strani pa lahko vsota en člen pridobi (če se premakne naprej desno krajišče). Zato po vsakem premiku d-ja ni težko popraviti dosedanje vsote, namesto da bi jo računali vsakič znova od začetka. Tako dobimo naslednjo rešitev:

```
int Popularni(int n, int k, const int obisk[])
{
  int vsota = 0, vsotaOd = 0, vsotaDo = 0, stPopularnih = 0;
  for (int d = 0; d < n; d++)
  {
    /* Naj bo novaOd..novaDo − 1 interval, ki pokriva dan d in še
       njemu najbližjih k drugih dni. */
    int novaOd = d − k / 2, novaDo = d + k / 2 + 1;
    if (novaOd < 0) novaOd = 0, novaDo = novaOd + k + 1;
    else if (novaDo > n) novaDo = n, novaOd = novaDo − k − 1;
    /* Trenutno je v spremenljivki „vsota“ vsota obiskov za dneve
       vsotaOd..vsotaDo − 1. Popravimo jo na vsoto dni novaOd..novaDo − 1. */
    while (vsotaOd < novaOd) vsota −= obisk[vsotaOd++];
    while (vsotaDo < novaDo) vsota += obisk[vsotaDo++];
    /* Preverimo, če ima trenutni dan nadpovprečni obisk. */
    if (vsota − obisk[d] < obisk[d] * k) stPopularnih++;
  }
  return stPopularnih;
}
```

Časovna zahtevnost te rešitve je le še $O(n)$, saj vsak člen zaporedja enkrat vstopi v vsoto in enkrat pade iz nje, tako da imamo s popravljanjem vsote vsega skupaj le $O(n)$ dela.

4. Sindikat

Plače je koristno popravljati po drevesu od spodaj navzgor. Na primer, če nek uslužbenec nima podrejenih, njegove plače ni treba spreminjati: znižati mu je ne smemo (tako pravi besedilo naloge), zvišati pa mu je ni treba, saj nam to ne bi nič pomagalo pri doseganju pogoja, da mora imeti vsak nadrejeni višjo plačo od svojih podrejenih.

Višje gor v drevesu pa lahko razmišljamo takole. Recimo, da opazujemo nekega uslužbenca u in smo že v vseh njegovih poddrevesih popravili plače tako, da ustrezajo zahtevam naloge. Zdaj lahko pogledamo plače u-jevih neposredno podrejenih uslužbencev in po potrebi ustrezno zvišamo u-jevo plačo. Na veljavnost pogojev v u-jevih poddrevesih to nič ne vpliva, tako da zdaj pogoj velja za u in vse njegove posredno ali neposredno podrejene. Ta postopek ponavljamo navzgor po drevesu, dokler ne obdelamo vseh uslužbencev.

Zapišimo postopek s psevdokodo:

podprogram ObdelajPoddrevo(u):
  $p := plača[u]$; $d := 0$;
  za vsakega u-jevega neposredno podrejenega uslužbenca v:
    $d := d +$ ObdelajPoddrevo(v);
    $p := \max\{p, plača[v] + 100\}$;
  if $p > plača[u]$:
    izpiši $ime[u]$;
    $d := d + p - plača[u]$; $plača[u] := p$;
  return d;

Postopek poženemo tako, da poiščemo direktorja (to je tisti u, ki nima nadrejenih) in zanj pokličemo ObdelajPoddrevo(u). Ko naš podprogram pride v neko vozlišče u, najprej z rekurzivnimi klici obdela njegova poddrevesa, nato izračuna novo plačo u-ja in vrne skupni znesek, za kolikor so se povečale plače u-ja in vseh njemu posredno ali neposredno podrejenih uslužbencev. Imena uslužbencev, ki se jim plača zviša, sproti tudi izpisujemo.

Koristno je razmisliti še o tem, kako našteti vse u-jeve neposredno podrejene uslužbence v. Preprosta rešitev je, da gremo z v po vseh uslužbencih od 1 do n in pri vsakem pogledamo, če velja $u = šef[v]$. Tako bomo imeli pri vsakem uslužbencu $O(n)$ dela in časovna zahtevnost celotnega postopka ob $O(n^2)$. Boljša rešitev je, da si na začetku, preden prvič pokličemo ObdelajPoddrevo, pripravimo za vsakega uslužbenca seznam podrejenih:

```
for u := 1 to n:
  podrejeni[u] := prazen seznam;
for u := 1 to n:
  dodaj u v seznam podrejeni[šef[u]];
```

Ti dve zanki porabita le $O(n)$ časa, s pomočjo seznamov *podrejeni* pa bomo lahko kasneje pri vsakem uslužbencu pregledali njegove podrejene v času, ki je sorazmeren s številom podrejenih. Časovna zahtevnost celotnega postopka bo le še $O(n)$, majhna slabost te rešitve pa je, da porabi $O(n)$ dodatnega pomnilnika (za sezname podrejenih).

5. 3-d labirint

V nalogi se skriva problem preiskovanja grafa. Recimo, da začnemo preiskovati mrežo v celici z najvišjim stebrom. Vpeljimo tabelo $w \times h$ logičnih vrednosti, v kateri element $d[x][y]$ pove, ali smo steber na celici (x, y) že uspeli doseči ali ne. Poleg tega bomo imeli še množico Q, v kateri hranimo koordinate stebrov, ki smo jih že dosegli, nismo pa še pregledali, kam je mogoče iz njih pot nadaljevati. Postopek teče v zanki: vsakič vzamemo nek steber iz Q in pogledamo, v katere njegove sosede je mogoče priti iz njega; če kakšen od teh sosedov doslej še ni bil označen kot dosegljiv, ga zdaj tako označimo in ga dodamo v Q, tako da bomo sčasoma pregledali tudi *njegove* sosede in tako naprej. Ko se Q izprazni, pa vemo, da smo pregledali vse stebre, ki jih je mogoče doseči z začetnega (to je z najvišjega); takrat moramo le še pregledati, če je tudi najnižji steber zdaj označen kot dosegljiv. Zapišimo našo rešitev še s psevdokodo:

```
(* Poiščimo najvišji in najnižji steber. Vse stebre označimo kot nedosegljive. *)
x_n := 1; y_n := 1; x_v := 1; y_v := 1;
for x := 1 to w do for y := 1 to h do
  d[x][y] := false;
  if v[x][y] > v[x_v][y_v] then x_v := x; y_v := y;
  if v[x][y] < v[x_n][y_n] then x_n := x; y_n := y;
(* Začnimo pri najvišjem stebru. *)
Q := {(x_v, y_v)}; d[x_v][y_v] := true;
(* Preglejmo, kaj vse je dosegljivo iz njega. *)
while Q ni prazna:
  naj bo (x, y) poljuben element Q; pobriši ta element iz Q;
  (* Steber (x, y) je dosegljiv. Preverimo njegove sosede. *)
  za vsako sosedo (x', y') celice (x, y):
    if |v[x][y] − v[x'][y']| > 1 or d[x'][y'] then continue;
    (* Steber (x', y') je mogoče doseči iz (x, y). *)
    d[x'][y'] := true; dodaj (x', y') v Q;
return d[x_n][y_n];
```

V zanki, ki mora pregledati sosede celice (x, y), so to načeloma štiri celice: $(x \pm 1, y)$ in $(x, y \pm 1)$, vendar pa moramo pri vsaki od njih še preveriti, če sploh leži na mreži (torej če je $1 \le x' \le w$ in $1 \le y' \le h$).

Načeloma je vseeno, v kakšnem vrstnem redu jemljemo stebre iz Q, saj bomo v vsakem primeru prej ali slej obiskali vse stebre, ki so dosegljivi z najvišjega. Pogosta rešitev je, da Q implementiramo z vrsto (*queue*), torej vedno vzamemo iz Q tisti steber, ki je že najdlje v Q; takšnemu vrstnemu redu pregledovanja prostora pravimo *iskanje v širino*.

12. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

20. januarja 2017

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. Umor

- Če rešitev ne deluje pravilno v primeru, da se umorjeni ni pogovarjal z nikomer, naj se ji zaradi tega odšteje pet točk.
- Umorjenec lahko v pogovoru nastopa bodisi kot prvi bodisi kot drugi sogovornik. Rešitve, ki pravilno obravnavajo le eno od teh dveh možnosti, ne pa obeh, lahko dobijo največ 12 točk.

- Rešitvam, ki po nepotrebnem preberejo celoten vhodni seznam pogovorov v pomnilnik (namesto da bi jih brale in obdelovale sproti), naj se zaradi tega odšteje tri točke.
- Za morebitne manjše napake pri branju vhodnih podatkov naj se rešitvi odšteje največ dve točki.

2. Naraščajoče besede

- Rešitvam, ki po nepotrebnem preberejo v pomnilnik celotno vhodno besedilo (namesto da bi ga brale po znakih ali pa po vrsticah), naj se zaradi tega odšteje največ tri točke.
- Nekateri programski jeziki imajo v knjižnici že pripravljene funkcije za razbijanje nizov na besede in podobne operacije. Nič ni narobe, če rešitev takšne funkcije uporablja (če seveda na koncu deluje v skladu z zahtevami naloge).
- Naloga ne določa eksplicitno, ali to, da so črke v besedi urejene naraščajoče, zahteva strogo urejenost ali pa sme biti po več zaporednih črk tudi enakih. Obe tidve interpretaciji sta sprejemljivi in rešitev lahko dobi vse točke ne glede na to, za katero od njiju se odloči.
- Če bi rešitev naredila kakšne neupravičene predpostavke o obliki vhodnih podatkov, ki ji delo občutno olajšajo (na primer to, da je vsaka beseda v svoji vrstici), naj se ji zaradi tega odšteje šest točk.
- Če rešitev ne deluje pravilno v primeru, ko je najdaljša naraščajoča beseda čisto na koncu vhodnega besedila (npr. zaradi kakšnih nerodno zastavljenih ustavitvenih pogojev v zankah), naj se ji zaradi tega odšteje tri točke.

3. Popularni dnevi

- Naloga pravi, naj bo postopek čim bolj učinkovit. Rešitve, ki imajo časovno zahtevnost $O(n \cdot k)$ namesto le $O(n)$, naj se zaradi tega odšteje štiri točke.
- Naivni rešitvi, ki poskuša vedno gledati (največ) $k/2$ dni pred in za opazovanim dnevom d, torej se ne zaveda tega, da mora npr. takrat, ko je $d < k/2$, vzeti več dni za d-jem kot pred njim), naj se zaradi tega odšteje osem točk. Če se rešitev tega problema zaveda, vendar takšnih robnih primerov ($d < k/2$ in $d > n - 1 - k/2$) ne obravnava pravilno, naj se ji zaradi tega odšteje štiri točke.
- Naloga pravi, da je dan popularen, če je obisk v njem večji od povprečja sosednjih dni. To pomeni, da če je obisk povprečju le enak, dan še ni popularen. To je razvidno tudi iz primera na koncu besedila (tretji dan ni popularen). Rešitvi, ki bi neupravičeno razglasila za popularne tudi take dni, pri katerih je obisk le enak povprečju, ne pa večji od njega, naj se zaradi tega odšteje dve točki.
- Naš primer rešitve preverja, ali je obisk nekega dne večji od povprečja, s pogojem oblike „`if (vsota < obisk[d] * k)`“. Enako dobro je tudi, če rešitev izračuna povprečje eksplicitno in preverja „`if (vsota / k < obisk[d])`“ ipd.

4. Sindikat

- Od rešitve pričakujemo predvsem, da se zaveda, da je treba plače popravljati od spodaj navzgor, in da zna izračunati nove plače v skladu z zahtevami naloge.
- Če rešitev sicer pravilno računa popravljene plače, ne izračuna pa pravilno vsote vseh zvišanj plač (ta vsota je eden od rezultatov, ki jih zahteva besedilo naloge), naj se ji zaradi tega odšteje tri točke.
- Podobno tudi, če rešitev sicer pravilno računa popravljene plače, vendar ne izpiše imen tistih zaposlenih, ki se jim je plača zvišala (oz. jih nekako drugače označi ali identificira), naj se ji zaradi tega odšteje tri točke.
- Učinkovitost postopka pri tej nalogi ni tako zelo pomembna in rešitev, ki ima časovno zahtevnost $O(n^2)$ namesto $O(n)$, lahko kljub temu dobi do 18 točk (če je drugače pravilna). Ravno tako tudi ni mišljeno, da bi se rešitev veliko ukvarjala s podrobnostmi tega, kako predstaviti drevo v pomnilniku (niti ni kakšne posebne nuje po tem, da bi bilo drevo predstavljeno še s čim drugim kot z nekaj tabelami, kot nakazuje že besedilo naloge).
- Rešitev, ki dvigne plače tako, da ima po novem vsak zaposleni vsaj 100 € višjo plačo od svojih podrejenih, vendar ti dvigi niso najmanjši možni, naj dobi največ 10 točk.
- Rešitev, ki poskuša plačo komu tudi znižati, ali pa ki spremeni plače tako, da ne zagotavlja, da bo imel vsak zaposleni po novem vsaj 100 € višjo plačo od svojih podrejenih, naj dobi največ 5 točk.

5. 3-d labirint

- Pri tej nalogi ni veliko poudarka na učinkovitosti, vendarle pa, če bi kakšna rešitev zaradi kakšne hudo nespametno implementirane rekurzije ali česa podobnega imela eksponentno časovno zahtevnost, naj dobi največ 7 točk (če je drugače pravilna).
- Ni pomembno, v kakšnem vrstnem redu rešitev preiskuje mrežo (npr. v širino, v globino ali še kako drugače), niti ne pričakujemo od nje, da bo skušala od najvišjega do najnižjega stebra (ali obratno) priti po najkrajši poti oz. ne da bi pri tem preiskala še vse druge stebre, ki so dosegljivi z njiju.
- Rešitvi, ki pomotoma dovoli neposreden premik z enega stebra na drugega tudi v primerih, ko imata njuni celici skupno le eno oglišče, ne pa stranice, naj se zaradi tega odšteje tri točke.
- Rešitvi, ki dela napake na robovih mreže, npr. ker pozabi, da tam celica določenih sosed sploh nima, naj se zaradi tega odšteje dve točki.
- Rešitvi, ki ne preverja pogoja, da je korak z enega stebra na sosednjega mogoč le, če se njuni višini razlikujeta največ za 1, ali pa ga preverja narobe, 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. Umor	lažja do srednja naloga v prvi skupini
2. Naraščajoče besede	srednje težka naloga v prvi skupini
3. Popularni dnevi	težka v prvi ali lažja do srednja naloga v drugi skupini
4. Sindikat	srednja do težja naloga v drugi skupini
5. 3-d labirint	srednja do težja v drugi ali lahka v tretji 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.