13. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

19. januarja 2018

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.");
   }
}
```

13. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

19. januarja 2018

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. Avtobus

Si na avtobusu in dobiš SMS „Kje si?“ od mame, ki čaka doma s kosilom. Želiš odgovoriti z imenom postaje, kjer se nahajaš, hkrati pa hočeš, da bo odgovor točen, torej, da ko SMS pošlješ, boš res na postaji, ki je napisana v SMSu. Dana imaš imena n postaj, $p_1, \ldots, p_n$, ki jih boš še obiskal do doma, in čase $d_1, \ldots, d_n$ v sekundah, ki jih bo avtobus potreboval za potovanje do naslednje postaje (d_i je čas vožnje od postaje p_{i-1} do p_i in je gotovo večji od 0). Da napišeš eno črko postaje, potrebuješ eno sekundo, prav tako pa tudi za to, da odpošlješ SMS (odpošlješ ga torej lahko eno sekundo po tistem, ko si napisal zadnjo črko sporočila). **Napiši program** ali del programa, ki izpiše, katero ime postaje boš sporočil. Če tako ime ne obstaja, naj izpiše „`doma`“. Če obstaja več primernih postaj, izpiši prvo med njimi (torej tisto z najnižjim indeksom i). Tvoj (pod)program lahko predpostavi, da so nizi p_i in števila d_i že podani v globalnih spremenljivkah (tabele ali seznami), lahko pa jih prebere s standardnega vhoda ali iz kakšne datoteke (karkoli ti je lažje).

2. Jedilnik

Srednje šole po Sloveniji si prizadevajo, da bi njihovi dijaki jedli čim bolj zdravo malico. Ravnatelj ene od srednjih šol bi rad, da mu **napišeš program**, s katerim bo lahko analiziral jedilnike, saj si želi, da bi bili njegovi dijaki zdravi in srečni.

```
primer_jedilnika = [
  "svinjski zrezek v omaki", "sirov kanelon", "ocvrt oslič",
  "svinjski zrezek v omaki", "ocvrt oslič", "sirov burek",
  "sirov kanelon", "ocvrt oslič", "sirov kanelon", "sirov kanelon"]
```

Jedilnik opišemo s tabelo oz. seznamom nizov, kot vidite na primeru. Vsak zaporedni element seznama ustreza enemu od zaporednih dni. Jedilnik se periodično ponavlja. Če je dolžina jedilnika n, bodo dijaki $(n+1)$-vi dan spet jedli tisto, kar je na prvem mestu na jedilniku.

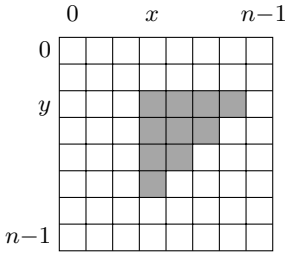
Ravnatelj je sicer ugotovil, da se jedi ponavljajo. Če pa je od dneva, ko je bila neka jed nazadnje na jedilniku, minilo dovolj časa, ni s tem nič narobe. Napišite funkcijo `KdajSpet(L)`, ki dobi nek jedilnik (seznam oz. tabelo `L`) in vrne enako dolg seznam, kjer je za vsak dan ena številka, ki pove, koliko dni po tistem dnevu bo ista jed naslednjič spet na jedilniku. (Če ne znaš vrniti seznama iz funkcije, lahko rezultate namesto tega tvoja funkcija tudi preprosto izpiše.) Upoštevajte, da je jedilnik periodičen, torej se na primer jedilnik `["burek", "jogurt", "burek"]` v naslednjih 10 dneh nadaljuje kot `["burek", "jogurt", "burek", "burek", "jogurt", "burek", "burek", "jogurt", "burek", "burek"]`.

Primer:

- `KdajSpet(["burek", "jogurt", "burek"])` mora vrniti `[2, 3, 1]`.
- `KdajSpet(primer_jedilnika)` (pri čemer je seznam `primer_jedilnika` tak, kot je definiran zgoraj v opisu naloge) mora vrniti `[3, 5, 2, 7, 3, 10, 2, 5, 1, 2]`.

3. Trikotniki

Podana je tabela velikosti $n \times n$ (elementi tabele so cela števila), v kateri bomo zelo pogosto izvajali poizvedbe o največji vrednosti v trikotnem območju. Vsaka poizvedba bo opisana s koordinatama (x, y) levega zgornjega kota trikotnika ter velikostjo trikotnika v. Odgovor na poizvedbo je največja vrednost po vseh tistih celicah (x', y'), za katere velja $x \leq x'$, $y \leq y'$, $(y' - y) + (x' - x) < v$. Primer kaže naslednja slika:



Primer tabele za $n = 8$. Pri poizvedbi (x, y) in $v = 4$ nas zanima maksimum vrednosti po vseh tistih celicah tabele, ki so na sliki pobarvani sivo. Trikotnik, ki nas pri posamezni poizvedbi zanima, ima vedno takšno obliko in orientacijo, kot jo vidimo na tej sliki, spremeni se lahko le njegova velikost in položaj v tabeli.

Opiši postopek (in oceni njegovo časovno zahtevnost), ki bo čim hitreje odgovarjal na opisane poizvedbe. (Bolj učinkoviti postopki dobijo več točk kot manj učinkoviti.) Vrednosti v tabeli se ne bodo spreminjale, zato si smeš pomagati z vnaprejšnjo obdelavo vrednosti v tabeli (torej si še pred prvo poizvedbo izračunaš kakšne pomožne podatke, ki ti bodo pomagali, da boš kasneje hitreje odgovarjal na poizvedbe), vendar naj bo časovna zahtevnost tega dela $o(n^3)$ — tvoj postopek naj porabi strogo manj od reda n^3 operacij.

4. Točke in koši

Napiši podprogram (funkcijo) Kosi(m), ki izračuna, na koliko načinov lahko ekipa na košarkarski tekmi doseže m točk. Posamezni koš je lahko vreden 1, 2 ali 3 točke. Z drugimi besedami nas torej zanima, na koliko načinov se da izraziti m kot vsoto števil 1, 2 ali 3 (pri tem je pomemben tudi vrstni red seštevancev). Zaželeno je, da je tvoja rešitev čim bolj učinkovita.

Primer: $m = 3$ točke se da doseči na 4 načine: $1 + 1 + 1$, $1 + 2$, $2 + 1$ in 3.

5. Povprečni položaj znaka

V nekem besedilu želimo izračunati povprečni položaj določenega znaka v posamezni vrstici in skupno v celotnem besedilu. (Pri tem štejemo male in velike črke kot različne znake.)

Položaj znaka je številka mesta, kjer se znak nahaja, šteto od začetka vrstice. Prvi znak v vrstici se nahaja na mestu 1.

Izračunani povprečni položaj zaokrožimo navzdol na celo število. Če se znak v neki vrstici ali besedilu sploh ne pojavlja, bomo kot povprečni položaj vzeli vrednost 0.

Primer besedila:

```
V Notranjem stoji vas, Vrh po imenu. V tej vasici je živel
v starih časih Krpan, močan in silen človek. Bil je neki
tolik, da ga ni kmalu takega. Dela mu ni bilo mar; ampak nosil
je od morja na svoji kobilici angleško sol, kar je bilo
pa že tistikrat ostro prepovedano.
```

Če nas zanima znak „`k`“, je njegov povprečni položaj:

- v prvi vrstici: 0,0 $\Longrightarrow$ 0
- v drugi vrstici: 49,0 $\Longrightarrow$ 49
- v tretji vrstici: 25,75 $\Longrightarrow$ 25
- v četrti vrstici: 34,66 ... $\Longrightarrow$ 34
- v peti vrstici: 12,0 $\Longrightarrow$ 12
- v celotnem besedilu: 31,7 $\Longrightarrow$ 31.

Napiši podprogram (ali opiši postopek), ki bo za dani znak izračunal in izpisal

- število pojavitev tega znaka v vsaki vrstici in v celotnem besedilu,
- povprečni položaj tega znaka v vsaki vrstici in v celotnem besedilu.

Besedilo lahko tvoj podprogram prebere s standardnega vhoda ali pa iz kakšne datoteke (karkoli ti je lažje).

13. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

19. januarja 2018

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Avtobus

Dolžino niza p_i označimo s $|p_i|$. Da napišemo in pošljemo ime postaje i, potrebujemo $|p_i| + 1$ sekund; torej, če vožnja do nje traja manj kot $|p_i| + 1$ sekund, njenega imena ne bomo mogli izpisati, sicer pa lahko. Postaje lahko pregledujemo po naraščajočih i, dokler ne najdemo prve take, pri kateri velja $|p_i| + 1 \leq d_1 + \ldots + d_i$. Vsote $d_1 + \ldots + d_i$ nam seveda ni treba računati pri vsakem i znova, ampak jo hranimo v neki spremenljivki in ji vsakič le prištejemo d_i trenutne postaje. Oglejmo si primer take rešitve v C++:

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

vector<string> imena;    // p_i
vector<int> casi;        // d_i

int main()
{
  int casVoznje = 0;
  for (int i = 0; i < imena.size(); i++)
  {
    casVoznje += casi[i];
    if (imena[i].length() + 1 <= casVoznje) {
      cout << imena[i] << endl; return 0; }
  }
  cout << "doma" << endl;
  return 0;
}
```

Še primer rešitve v pythonu:

```
imena = [...]    # p_i
casi = [...]     # d_i

def KajIzpisati():
  casVoznje = 0
  for i in range(len(imena)):
    casVoznje += casi[i]
    if len(imena[i]) + 1 <= casVoznje: return imena[i]
  return "doma"
print(KajIzpisati())
```

2. Jedilnik

Preprosta rešitev je, da uporabimo dve gnezdeni zanki. V zunanji zanki (po i) gremo po vseh elementih jedilnika; v notranji zanki (po d) pa gremo od trenutnega elementa naprej, dokler ne najdemo naslednje pojavitve iste jedi. Pri tem pazimo na to, da se jedilnik ciklično ponavlja na vsakih n dni (kjer je n dolžina jedilnika). Če smo začeli pri dnevu i in se premaknili za d dni naprej, nas zanima jed na dan $i + d$, v jedilniku pa jo najdemo na indeksu $(i + d) \bmod n$. Notranja zanka se ustavi najkasneje pri $d = n$, saj se po n dneh ponovi cel jedilnik.

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

vector<int> KdajSpet(const vector<string>& jedilnik)
{
  int n = jedilnik.size();
  vector<int> rezultati;
  for (int i = 0; i < n; i++)
  {
    int d = 1;
    while (jedilnik[(i + d) % n] != jedilnik[i]) d++;
    rezultati.push_back(d);
  }
  return rezultati;
}
```

Zapišimo to rešitev še v pythonu:

```
def KdajSpet(jedilnik):
  n = len(jedilnik); rezultati = []
  for i in range(n):
    d = 1
    while jedilnik[(i + d) % n] != jedilnik[i]: d += 1
    rezultati.append(d)
  return rezultati
```

Slabost te rešitve je, da v najslabšem primeru (če so v jedilniku vse jedi različne in se torej vsaka ponovi šele po n dnevih) notranja zanka naredi vsakič po $n-1$ iteracij in je zato časovna zahtevnost tega postopka kar $O(n^2)$.

Boljša rešitev je, da si pri branju jedilnika za vsako jed zapomnimo, kdaj smo jo nazadnje videli. Primerna podatkovna struktura za to je slovar (ki je v praksi ponavadi implementiran kot razpršena tabela). Ko recimo na dan i naletimo na neko jed, pogledamo v slovar; če tam vidimo, da smo jo nazadnje videli na dan p, si lahko zdaj (v izhodnem seznamu) zapišemo, da se jed z dneva p ponovi po $i-p$ dneh. Z zanko po i je koristno iti do $2n$, kar nam zagotovi, da bomo vsako jed zanesljivo videli vsaj dvakrat. Oglejmo si primer takšne rešitve v C++, kjer lahko kot slovar uporabimo razred unordered_map iz standardne knjižnice:

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

vector<int> KdajSpet2(const vector<string>& jedilnik)
{
  int n = jedilnik.size();
  unordered_map<string, int> kdajPrej;
  vector<int> rezultati; rezultati.resize(n);
  for (int i = 0; i < 2 * n; i++)
  {
    const string &jed = jedilnik[i % n];
    if (auto it = kdajPrej.find(jed); it != kdajPrej.end())
      if (int prej = it->second; prej < n)
        rezultati[prej] = i - prej;
    kdajPrej[jed] = i;
  }
  return rezultati;
}
```

Podobno lahko naredimo tudi v pythonu, kjer je slovar še bolj pri roki:

```
def KdajSpet2(jedilnik):
  n = len(jedilnik); rezultati = [-1] * n
```

```
  kdajPrej = {}
  for i in range(2 * n):
    jed = jedilnik[i % n]
    prej = kdajPrej.get(jed, n)
    if prej < n: rezultati[prej] = i − prej
    kdajPrej[jed] = i
  return rezultati
```

3. Trikotniki

(1) Najpreprostejša rešitev je, da pri poizvedbi pregledamo vse celice trikotnika, na katerega se poizvedba nanaša. V neki spremenljivki hranimo največjo vrednost doslej in jo sproti popravljamo, ko zagledamo kakšno še večjo vrednost. Za pregled trikotnika uporabimo dve gnezdeni zanki, eno po vrsticah in drugo po celicah znotraj vrstice. Spotoma pazimo še na to, da nekateri deli trikotnika mogoče štrlijo ven iz mreže in do tistih celic tabele ne smemo dostopati (ker sploh ne obstajajo):

```
int Poizvedba(int x, int y, int v)
{
  int naj = T[x][y];
  for (int dy = 0; dy < v && y + dy < n; dy++)
    for (int dx = 0; dx < v − dy && x + dx < n; dx++)
      if (T[x + dx][y + dy] > naj) naj = T[x + dx][y + dy];
  return naj;
}
```

Časovna zahtevnost te rešitve je $O(v^2)$, saj mora pregledati vse celice trikotnika, teh pa je $v + (v-1) + (v-2) + \ldots + 2 + 1 = v(v+1)/2$. Do boljših rešitev lahko pridemo, če si (kot svetuje že besedilo naloge) vnaprej izračunamo maksimume nekaterih območij tabele in jih nato pri poizvedbi na primeren način skombiniramo v maksimum ravno tistega trikotnega območja, po katerem sprašuje trenutna poizvedba.

(2) Vsako vrstico lahko na primer razdelimo na „bloke“, dolge po približno $\sqrt{n}$ celic (tako je tudi blokov v vsaki vrstici približno $\sqrt{n}$). Za vsak blok si zapomnimo maksimum vrednosti po vseh celicah v bloku. Ta predpriprava nam vzame $O(n^2)$ časa in zasede $O(n\sqrt{n})$ prostora.

Pri poizvedbi lahko razmišljamo podobno kot prej, le da ko smo pri prvi celici kakšnega bloka, preverimo, če ta blok v celoti leži v našem trikotniku; če da, vzamemo maksimum bloka (ki smo si ga izračunali vnaprej) in nato preostanek bloka preskočimo (nadaljujemo pri prvi celici naslednjega bloka). Tako moramo posamično pregledati le tiste celice, ki ležijo v trikotniku, ne leži pa cel njihov blok v trikotniku; takih je največ $\sqrt{n}$ na začetku vrstice in $\sqrt{n}$ konec vrstice. Vmes pa je v vrstici še največ $\sqrt{n}$ celih blokov. Tako imamo z vsako vrstico $O(\sqrt{n})$ dela, s celim trikotnikom pa $O(v\sqrt{n})$.

(3) Podobna, a še boljša rešitev je naslednja: naredimo podobno kot zgoraj, le da z bloki dolžine 2, 4, 8, 16 in tako naprej. To si lahko predstavljamo tako, kot da smo nad vsako vrstico zgradili binarno drevo, ki vsebuje maksimume vse daljših blokov (katerih dolžine so potence števila 2). Priprava vseh teh maksimumov nam vzame $O(n^2)$ časa in zasede $O(n^2)$ prostora.

Pri poizvedbi opazimo, da lahko zdaj vsako vrstico našega trikotnika razdelimo na bloke tako, da uporabimo največ dva bloka posamezne dolžine (enega na začetku vrstice in enega na koncu). Dolžine blokov so 2^k za $k = 0, 1, \ldots, \lfloor \log_2 n \rfloor$, torej moramo pregledati približno $O(\log n)$ blokov, da dobimo maksimum ene vrstice trikotnika. Časovna zahtevnost ene poizvedbe je torej $O(v \log n)$.

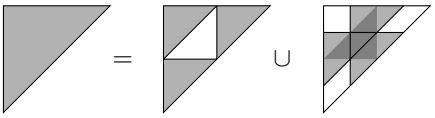
(4) Rešitev lahko še izboljšamo, če smo pripravljeni žrtvovati še malo več pomnilnika. Za vsako celico (x, y) naše tabele in za vsak k od 0 do $\lfloor \log_2 n \rfloor$ izračunajmo maksimum vrednosti v bloku dolžine 2^k, ki se začne pri celici (x, y). To je torej podobno prejšnji rešitvi, le da se bloki zdaj prekrivajo. Pri vsakem k imamo torej $O(n^2)$ blokov in predpriprava podatkov nam zdaj vzame $O(n^2 \log n)$ časa ter prav toliko prostora.

Pri poizvedbi spet razdelimo trikotnik na vrstice, nato pa v vsaki vrstici vzemimo največji tak k, za katerega je $2^k \leq$ od dolžine vrstice. Če torej skombiniramo dva bloka dolžine 2^k — enega, ki se začne na začetku vrstice, in enega, ki se konča na koncu vrstice

— bosta skupaj pokrila celo vrstico (verjetno se bosta tudi prekrivala med sabo, kar pa nas za izračun maksimuma nič ne moti). Tako moramo torej v vsaki vrstici trikotnika pregledati le dva bloka in časovna zahtevnost poizvedbe je le še $O(v)$.

(5) Še boljša rešitev pa je naslednja: vnaprej izračunajmo maksimume trikotnih območij prav take oblike, na kakršne se nanašajo poizvedbe pri tej nalogi, za vse možne položaje zgornjega levega kota trikotnika in za velikosti oblike $v = 2^k$. Pri vsakem k imamo torej $O(n^2)$ trikotnikov, zato nam ta predpriprava podatkov vzame $O(n^2 \log n)$ časa in prostora.

Pri poizvedbi s trikotnikom velikosti v nato vzemimo največji tak k, za katerega je $2^k \leq v$. Naš trikotnik velikosti v lahko popolnoma pokrijemo s šestimi trikotniki velikosti 2^k, kot vidimo na spodnjem primeru:



Slika kaže, kako lahko trikotnik velikosti v pokrijemo s šestimi trikotniki velikosti $v/2$. Prvi trije manjši trikotniki pokrijejo tri četrtine večjega, zadnjo četrtino (ki je na desni sliki osenčena temno sivo) pa pokrijemo z drugimi tremi manjšimi trikotniki. Pri naših poizvedbah manjši trikotniki nimajo nujno stranice $v/2$, pač pa stranico 2^k, ki je $\geq v/2$ (saj smo za 2^k vzeli največjo tako potenco števila 2, ki je $\leq v$). S takimi bo večji trikotnik še toliko lažje pokriti (naši trikotniki velikosti 2^k se bodo še malo bolj prekrivali med seboj kot na gornji sliki, kar pa nas pri računanju maksimuma nič ne ovira).

Ker smo za vsakega od šestih manjših trikotnikov že vnaprej izračunali maksimum vrednosti v njem, moramo zdaj le še poiskati največjega izmed teh šestih maksimumov, pa imamo rezultat, po katerem sprašuje poizvedba. Tako nam torej posamezna poizvedba vzame le še $O(1)$ časa.

4. Točke in koši

Naj bo $f(k)$ število načinov, na katere se dá izraziti k kot vsoto števil 1, 2 in 3. Če je prvi seštevanec enak a, morajo ostali seštevanci dati vsoto $k - a$, to pa je mogoče narediti na $f(k - a)$ načinov. Za prvi seštevanec so, tako kot za vsakega drugega, le tri možnosti: a je lahko 1, 2 ali 3. Vsoto k lahko torej dobimo na $f(k-1)$ takih načinov, pri katerih ima prvi seštevanec vrednost 1, na $f(k-2)$ takih načinov, pri katerih ima prvi seštevanec vrednost 2, in še na $f(k-3)$ takih načinov, pri katerih ima prvi seštevanec vrednost 3. Vsega skupaj imamo torej $f(k-1) + f(k-2) + f(k-3)$ načinov, kako dobiti vsoto k. Tako smo dobili zvezo

$$f(k) = f(k-1) + f(k-2) + f(k-3).$$

Potrebne je še nekaj pazljivosti pri robnih primerih. Vsoto 0 je mogoče dobiti na en način, namreč tako, da sploh ne damo nobenega koša; za negativne k pa vsote sploh ni mogoče dobiti. Vzeti moramo torej $f(0) = 1$ in $f(k) = 0$ za $k < 0$. To funkcijo lahko zelo preprosto računamo z rekurzijo:

```
int Kosi(int m)
{
  if (m < 0) return 0;
  else if (m == 0) return 1;
  else return Kosi(m - 1) + Kosi(m - 2) + Kosi(m - 3);
}
```

Ali v pythonu:

```
def Kosi(m):
  if m < 0: return 0
  elif m == 0: return 1
  else: return Kosi(m - 1) + Kosi(m - 2) + Kosi(m - 3)
```

Slabost te rešitve je, da računa iste stvari po večkrat. Na primer, klic Kosi(m) pokliče Kosi(m − 1), ta pa Kosi((m − 1) − 1), torej Kosi(m − 2). Toda Kosi(m − 2) se pokliče kasneje še enkrat neposredno iz Kosi(m). Sčasoma je takega ponavljanja vse več in naša rešitev je zaradi tega pri večjih m neuporabno počasna. Bolje je, če si že izračunane vrednosti zapomnimo in jih kasneje ne računamo znova. Iz formul zgoraj vidimo, da bomo pri izračunu $f(k)$ potrebovali vrednosti $f(k-1)$, $f(k-2)$ in $f(k-3)$, torej bo najlažje, če bomo vrednosti funkcije $f(k)$ računali kar po vrsti po naraščajočih k. Pri tem si moramo vedno zapomniti le zadnje tri že izračunane vrednosti, starejše (za $k-4$, $k-5$ in tako naprej) pa lahko sproti pozabljamo, saj jih ne bomo več potrebovali. Tako dobimo naslednjo rešitev:

```
int Kosi2(int m)
{
  if (m < 0) return 0;
  int r[3] = { 1, 0, 0 };
  for (int k = 1; k <= m; k++)
    /* Na tem mestu za i = 0, 1, 2 velja: r[(k − i) % 3] = f(k − i). */
    r[k % 3] = r[0] + r[1] + r[2];
  return r[m % 3];
}
```

In podobno v pythonu:

```
def Kosi2(m):
  if m < 0: return 0
  r = [1, 0, 0]
  for k in range(1, m + 1):
    r[k % 3] = r[0] + r[1] + r[2]
  return r[m % 3]
```

Zadnje tri vrednosti funkcije f torej hranimo v tabeli r in to tako, da je $f(k)$ shranjena v r[k % 3]. Ko izračunamo $f(k)$, lahko v tabeli r povozimo vrednost $f(k-3)$ z vrednostjo $f(k)$, saj je obema namenjena ista celica tabele in vrednosti $f(k-3)$ ne bomo več potrebovali.

Opazimo lahko, da je formula za $f(n)$ zelo podobna tisti za Fibonaccijeva števila, le da tam nastopa vsota prejšnjih dveh členov, tu pa vsota prejšnjih treh. Vrednostim $f(n)$, s kakršnimi se ukvarjamo pri tej nalogi, zato včasih pravijo „Tribonaccijeva števila“. Pokazati je mogoče, da zanje velja približno $f(n) \approx 0{,}618 \cdot 1{,}839^n$.

5. Povprečni položaj znaka

Naloge se lahko lotimo na več načinov. Ena možnost je, da beremo vhodne podatke po znakih; pri tem v nekaj spremenljivkah vzdržujmo trenutni položaj v vrstici (v spodnji rešitvi je to x), število pojavitev iskanega znaka v trenutni vrstici (nVrst) in vsoto njihovih položajev (sVrst). Poleg tega vzdržujmo tudi število pojavitev iskanega znaka v vsem doslej prebranem besedilu (nBes) in vsoto njihovih položajev (sBes). Vsak prebrani znak primerjamo z iskanim; če se ujema, povečamo števili pojavitev (nVrst in nBes) za 1 in prištejemo trenutni položaj k vsotama sVrst in sBest. Ko pridemo do konca trenutne vrstice, izpišemo število pojavitev in povprečni položaj (slednjega izračunamo tako, da vsoto položajev delimo s številom pojavitev, pri tem pa pazimo, da ne bomo delili z 0), nato pa trenutni položaj x postavimo nazaj na 0, da bomo pripravljeni na začetek naslednje vrstice. Ko pridemo do konca celotnega vhodnega besedila, pa na podoben način izpišemo še število pojavitev in povprečni položaj v celotnem besedilu. Oglejmo si primer take rešitve v C-ju:

```
#include <stdio.h>

int Povprecni(char c)
{
  int nBes = 0, sBes = 0, nVrst = 0, sVrst = 0, stVrstice = 0, x = 0;
  while (true)
  {
```

```
    /* Preberimo naslednji znak. */
    int ch = fgetc(stdin); x++;
    /* Smo na koncu vrstice? */
    if (ch == '\n' || (ch == EOF && x > 0))
    {
      x = 0; stVrstice++;
      printf("V %d. vrstici: %d pojavitev, povprečni položaj %d.\n",
            stVrstice, nVrst, sVrst / (nVrst <= 0 ? 1 : nVrst));
      nVrst = 0; sVrst = 0;
    }
    if (ch == EOF) break; else if (ch == '\n') continue;
    /* Ali je trenutni znak tisti, ki ga iščemo? */
    if (ch == c) nBes++, nVrst++, sBes += x, sVrst += x;
  }
  printf("Skupaj: %d pojavitev, povprečni položaj: %d.\n",
        nBes, sBes / (nBes <= 0 ? 1 : nBes));
}
```

Lahko pa namesto po znakih beremo vhodno besedilo po vrsticah. Pri vsaki vrstici se v notranji zanki sprehodimo po znakih, iščemo pojavitve iskanega znaka ter ustrezno povečujemo spremenljivki nVrst in sVrst. Na koncu vrstice izpišemo število pojavitev in povprečni položaj, nato pa ustrezno povečamo še število pojavitev in vsoto položajev v celotnem besedilu (nBes in sBes). Na koncu izpišemo še statistike za celotno besedilo, enako kot pri prejšnji rešitvi. Oglejmo si implementacijo te rešitve v pythonu:

```
import sys

def Povprecni(c):
  stVrstice = 0; nBes = 0; sBes = 0
  for vrstica in sys.stdin:
    stVrstice += 1; nVrst = 0; sVrst = 0
    for x in range(len(vrstica)):
      if vrstica[x] == c:
        nVrst += 1; sVrst += (x + 1)
    print("V %d. vrstici: %d pojavitev, povprečni položaj %d." % (
        stVrstice, nVrst, sVrst / max(nVrst, 1)))
    nBes += nVrst; sBes += sVrst
  print("Skupaj: %d pojavitev, povprečni položaj %d." % (
      nBes, sBes / max(nBes, 1)))
```

13. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

19. januarja 2018

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. Avtobus

- Rešitvam, ki po nepotrebnem računajo vsoto $d_1 + \ldots + d_i$ vsakič od začetka, namesto da bi le prištele trenutni d_i k prejšnji vsoti, naj se zaradi tega odšteje tri točke.
- Naloga pravi, da lahko SMS pošljemo 1 sekundo po tistem, ko natipkamo zadnji znak sporočila. Rešitvam, ki pomotoma predpostavijo, da ga lahko pošljemo takoj, ko natipkamo zadnji znak (in ne šele 1 sekundo kasneje), naj se zaradi tega odšteje dve točki.

- Za morebitne manjše napake pri branju vhodnih podatkov naj se rešitvi odšteje največ dve točki.
- Če rešitev zahtevanega niza ne izpiše, pač pa ga le vrne kot rezultat neke funkcije, naj se ji zaradi tega ne odšteva točk.
- Rešitev sme predpostaviti, da je dolžina posameznega niza p_i krajša od neke razumne zgornje meje (npr. nekaj deset znakov).

2. Jedilnik

- Rešitve, ki imajo časovno zahtevnost $O(n^2)$, lahko dobijo največ 15 točk. Rešitve, ki imajo časovno zahtevnost $O(n \log n)$ (npr. ker uporabljajo slovar, implementiran z drevesom, ali pa ker urejajo seznam parov $\langle L[i], i \rangle$), lahko dobijo največ 18 točk.
- Pri tej nalogi so jedi predstavljene z nizi, vendar ni mišljeno, da bi bil velik poudarek na delu z nizi. Rešitev sme predpostaviti, da so ti nizi krajši od neke razumne zgornje meje (npr. nekaj deset znakov). Ne sme pa predpostaviti, da je nabor vseh možnih jedi znan vnaprej, npr. da lahko v vhodnih podatkih nastopajo le tisti nizi, kakršne vidimo v primerih v besedilu naloge.
- Naloga večkrat poudari, da se jedilnik ciklično ponavlja. Rešitvi, ki tega ne upošteva in npr. pravilno obdela le tiste jedi, ki se naslednjič pojavijo že v istem ciklu (ne pa šele v naslednjem), naj se zaradi tega odšteje 8 točk.

3. Trikotniki

- Naloga pravi, naj bo postopek čim bolj učinkovit. Rešitve, pri katerih je časovna zahtevnost posamezne poizvedbe $O(n^2)$, lahko dobijo največ 10 točk. Rešitve, ki imajo časovno zahtevnost poizvedbe strogo manj kot $O(n^2)$, lahko dobijo vse točke.
- Rešitev sme brez škode predpostaviti, da velja $0 \leq x < n$, $0 \leq y < n$ in $0 < v$, torej da so vhodni podatki smiselni.
- Lahko se zgodi, da kakšen del poizvedovalnega območja leži zunaj mreže. Če poskuša rešitev v takih primerih dostopati do neobstoječih celic tabele, naj se ji zaradi tega odšteje 3 točke.
- Rešitev, ki bi npr. predpostavila, da je n vedno 8 ali pa v vedno 4, ker je slučajno tako v primeru v besedilu naloge, naj dobi največ 5 točk.
- Naloga pravi, da mora rešitev za morebitno predpripravo podatkov pred prvo poizvedbo porabiti strogo manj kot $O(n^3)$ časa. Rešitve, ki porabijo $O(n^3)$ ali še več časa, lahko dobijo največ 10 točk.

4. Točke in koši

- Naloga pravi, naj bo rešitev čim bolj učinkovita. Rešitve z eksponentno zahtevnostjo (npr. preprosta rekurzivna rešitev brez pomnjenja že izračunanih rezultatov) lahko dobijo največ 12 točk. Rešitve s polinomsko časovno zahtevnostjo lahko dobijo vse točke.
- Pri rešitvah s polinomsko časovno zahtevnostjo je vseeno, ali rešitev porabi le konstantno mnogo pomnilnika (kot npr. funkcija `Kosi2` iz našega primera rešitve, ki hrani med izračunom le prejšnje tri vrednosti funkcije f) ali pa kar $O(n)$ pomnilnika (npr. ker bi si shranjevala vse doslej izračunane vrednosti funkcije f).
- Za morebitno čudno obnašanje pri $m < 0$ naj se rešitvam ne odšteva točk. Za morebitne napake pri majhnih nenegativnih m, npr. $m = 0, 1, 2$, naj se rešitvi odšteje tri točke.
- Vrednost $f(m)$, ki jo mora funkcija vrniti, narašča eksponentno hitro v odvisnosti od m, torej kmalu postane prevelika za običajne celoštevilske tipe (npr. 32-bitni **int**, kot ga ponavadi najdemo v C-ju in podobnih jezikih). Pri tej nalogi ni mišljeno, da bi se morala tekmovalčeva rešitev kaj ukvarjati s to težavo (npr. uporabiti arbitrary-precision aritmetiko) ali se je sploh kakorkoli zavedati.

5. Povprečni položaj znaka

- Objavili smo dve rešitvi: eno, ki bere vhodno besedilo po znakih, in eno, ki ga bere po vrsticah. Oboje je enako dobro. Prav tako je enako dobro tudi, če rešitev prebere celo vhodno besedilo naenkrat v pomnilnik. Rešitev sme tudi predpostaviti neko zgornjo mejo za dolžino posamezne vrstice (npr. da je vrstica dolga največ 100 znakov ali kaj podobnega).
- Pri tej nalogi ni mišljeno, da bi bil poudarek na branju vhodnih podatkov. Za drobne napake pri branju vhodnega besedila naj se rešitvi odšteje največ 2 točki. (Primer take drobne napake bi na primer bil, če bi v naši prvi rešitvi, torej tisti, ki bere po znakih, pozabili na pogoj `|| (ch == EOF && x > 0)`, ki skrbi za to, da pravilno obdelamo zadnjo vrstico tudi tedaj, če se ne konča na znak za konec vrstice.)
- Če rešitev vhodnega besedila sploh ne bere, ampak predpostavi, da ga kar dobi v nekem seznamu ali tabeli, naj se ji zaradi tega odšteje največ tri točke.
- Format izpisa pri tej nalogi ni posebej predpisan, pomembno je le, da so v njem med drugim vsi podatki, ki jih naloga zahteva, torej število pojavitev znaka in povprečni položaj (zaokrožen navzdol na celo število). Nič ni narobe, če rešitev poleg tega izpiše še kaj drugega (npr. povprečni položaj kot realno število). Zaradi morebitnih napak pri zaokrožanju povprečnega položaja na celo število naj se rešitvi odšteje največ dve točki.
- Če rešitev zaradi kakšnega deljenja z 0 (npr. če se iskani znak v neki vrstici sploh ne pojavlja) daje napačne rezultate ali pa se celo sesuje, naj se ji zaradi tega odšteje štiri točke.

Težavnost nalog

Državno tekmovanje ACM v znanju računalništva poteka v treh težavnostnih skupinah (prva je najlažja, tretja pa najtežja); na tem šolskem tekmovanju pa je skupina ena sama, vendar naloge v njej pokrivajo razmeroma širok razpon zahtevnosti. Za občutek povejmo, s katero skupino državnega tekmovanja so po svoji težavnosti primerljive posamezne naloge letošnjega šolskega tekmovanja:

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM
1. Avtobus	lažja naloga v prvi skupini
2. Jedilnik	srednje težka naloga v prvi skupini
3. Trikotniki	srednja do težja naloga v drugi skupini[1]
4. Točke in koši	težka naloga v prvi ali lažja do srednja v drugi skupini
5. Povp. položaj	težja naloga v prvi ali lažja v drugi skupini

Če torej na primer nek tekmovalec reši le eno ali dve lažji nalogi, pri ostalih pa ne naredi (skoraj) ničesar, to še ne pomeni, da ni primeren za udeležbo na državnem tekmovanju; pač pa je najbrž pametno, če na državnem tekmovanju ne gre v drugo ali tretjo skupino, pač pa v prvo.

Podobno kot prejšnja leta si tudi letos želimo, da bi čim več tekmovalcev s šolskega tekmovanja prišlo tudi na državno tekmovanje in da bi bilo šolsko tekmovanje predvsem v pomoč tekmovalcem in mentorjem pri razmišljanju o tem, v kateri težavnostni skupini državnega tekmovanja naj kdo tekmuje.

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

[1] Opomba: težavnost te naloge je zelo odvisna od tega, kako točkujemo rešitve odvisno od njihove učinkovitosti. Na primer, če bi dali vse točke že za rešitev, kjer traja vsaka poizvedba $O(v^2)$ časa, bi bila lahko to lažja naloga za prvo skupino. Če bi za vse točke zahtevali rešitev s poizvedbami v času $O(1)$, bi bila to težka naloga tudi za tretjo skupino. V navodilih za letošnje šolsko tekmovanje smo ubrali srednjo pot: za poizvedbe v času $O(v^2)$ se dobi polovico točk, za karkoli hitrejšega od tega pa vse točke.