17. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

28. januarja 2022

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.");
   }
}
```

17. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

28. januarja 2022

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. Seštevanje ulomkov

V neki butalski družini imajo veliko otrok. Vsakič, ko ima kdo od otrok rojstni dan, slavljenec dobi torto, ostali otroci pa se postavijo v vrsto za njim. Slavljenec prereže torto na pol. Polovico bodisi pojé bodisi posadi v zemljo (da bo drugo leto zrasla nova torta), drugo polovico pa dá naslednjemu otroku v vrsti. Vsak naslednji otrok naredi enako: kos, ki ga prejme, prereže na pol, polovico bodisi pojé bodisi posadi, polovico poda naprej. Izjema je zadnji otrok v vrsti, ki vedno pojé cel kos, ki pride do njega.

Napiši program (ali podprogram oz. funkcijo), ki prebere podatke o več takšnih praznovanjih. V prvi vrstici dobi število praznovanj r (največ 1000) in število otrok n (največ 20), nato pa sledi r vrstic, ki opisujejo praznovanja; v vsaki od teh je niz n znakov, ki opisujejo, kaj naredi posamezni otrok pri tistem praznovanju (`'J'` = pojé, `'S'` = posadi). Podatke lahko bereš s standardnega vhoda ali pa iz datoteke `praznovanja.txt` (karkoli ti je lažje).

Vsaka torta tehta 1 kg. Tvoj program naj izpiše, koliko kilogramov torte, zaokroženo na najbližji celoštevilski kilogram, je na koncu zakopanih v zemlji. (Pol kilograma naj zaokroži navzgor; na primer, če je v zemlji točno 3,5 kg torte, naj izpiše 4.)

Vsi rezi so na vseh praznovanjih so bili popolnoma natančni; fizika pravi, da to ni mogoče, toda Butalci se tega niso nikoli naučili.

Primer vhodnih podatkov:	Pripadajoči izhod:

`5 4`
`SJSJ`
`SSJJ`
`JSJJ`
`JJJJ`
`JJSJ`
	`2`

Komentar: na prvem praznovanju so zakopali $\frac{1}{2} + \frac{1}{8}$ kg, na drugem $\frac{1}{2} + \frac{1}{4}$ kg, na tretjem $\frac{1}{4}$ kg, na četrtem 0 kg in na petem $\frac{1}{8}$ kg. Skupaj je to $\frac{7}{4}$ kg, temu najbližje celo število pa je 2.

2. Slovar

Imamo podan seznam besed iz slovarja. V njem želimo poiskati besede, ki ustrezajo pogoju naslednje oblike: predpisana je dolžina besede in za nekatera mesta v besedi je predpisano, katera črka mora stati na njih; za ostala mesta v besedi pa je vseeno, katere črke stojijo tam. Tak pogoj lahko opišemo z „vzorcem“ — to je niz, v katerem nastopajo črke in zvezdice, pri čemer zvezdice pomenijo, da je za tisto mesto v besedi vseeno, katera črka stoji tam.

Primer: vzorcu `*i*a` ustrezajo med drugim besede `miza`, `zima` in `riba`; ne ustrezajo pa mu na primer besede `mirta` (ker je predolga), `ica` (ker je prekratka) in `prva` (ker na drugem mestu nima `i`, ampak `r`).

(*a*) **Napiši program** (ali podprogram oz. funkcijo), ki kot vhodne podatke dobi seznam besed in vzorec ter izpiše tiste besede, ki ustrezajo temu vzorcu. Podrobnosti tega, v kakšni obliki dobi vhodne podatke, si izberi sam in jih v rešitvi tudi opiši.

(*b*) Recimo, da bi želeli izvesti takšna iskanja za veliko različnih vzorcev, pri čemer bi bil seznam besed ves čas enak. Zato je morda koristno ta seznam najprej nekako predelati ali preurediti, da bomo potem lahko čim hitreje iskali besede, ki ustrezajo različnim vzorcem. **Opiši**, kako bi to naredil in kakšen bi bil potem postopek iskanja besed, ki ustrezajo danemu vzorcu. (Pri tem delu naloge je dovolj opis postopka, ni treba pisati implementacije v kakšnem konkretnem programskem jeziku.)

Podnaloga (*a*) je vredna 13 točk, podnaloga (*b*) pa 7 točk. Pri obeh lahko predpostaviš, da v besedah in vzorcu nastopajo le male črke angleške abecede (od `a` do `z`), v vzorcu pa seveda poleg njih tudi zvezdice. Poleg tega lahko tudi predpostaviš, da gre res za besede iz slovarja kakšnega naravnega jezika (torej besede niso pretirano dolge, ne začnejo se vse na isto črko in podobno).

3. Genialno

Na pravokotni karirasti mreži, visoki h vrstic in široki w stolpcev, je na vsakem polju 0 ali več žetonov. Ko dodamo nov žeton na neko polje, dobimo za to določeno število točk, in sicer takole: iz polja, kamor smo postavili novi žeton, gremo v vse štiri možne smeri (gor, dol, levo, desno) tako daleč, dokler ne naletimo na prazno polje (táko z nič žetoni) ali na rob mreže. Dobimo toliko točk, kolikor je skupno število žetonov na takó obiskanih poljih (med slednja ne štejemo polja, na katero smo postavili novi žeton — glej primer spodaj).

Napiši program (ali podprogram oz. funkcijo), ki ugotovi, na katero polje moramo dati novi žeton, da bomo dobili največ točk. (Če obstaja več enako dobrih rešitev, je vseeno, katero izpiše.) Podrobnosti tega, kako dobiš podatke o mreži (velikost mreže in število žetonov na posameznem polju), si izberi sam in jih v rešitvi tudi opiši. Zaželeno je, da je tvoja rešitev učinkovita, tako da bo delovala hitro tudi za velike mreže (npr. s po nekaj tisoč stolpci in vrsticami).

Primer: recimo, da imamo mrežo, kot jo kaže naslednja slika.

0	0	1	2	0
0	2	3	1	0
0	0	2	1	2
0	0	0	3	1

Če postavimo novi žeton na drugo polje (z leve) v drugi vrstici (od zgoraj), dobimo $3+1 = 4$ točke. Če ga postavimo na drugo polje v tretji vrstici, dobimo $(2)+(2+1+2) = 7$ točk. Če ga postavimo na četrto polje v drugi vrstici, dobimo $(2)+(3+2)+(1+3) = 11$ točk, kar je pri tej mreži tudi najboljša možna poteza.

4. Parkirišče

Ob cesti je parkirišče, na katerem eden za drugim parkirajo tovornjaki. Časi prihodov in odhodov ter dolžine tovornjakov so znane vnaprej; vseh tovornjakov je n, pri čemer je i-ti izmed njih dolg d_i metrov, na parkirišče bo prišel ob času p_i in se na njem zadržal t_i časa, nato pa bo odpeljal. Tovornjaki niso nujno oštevilčeni v nobenem posebnem vrstnem redu.

Ko tovornjak zapusti parkirišče, se tisti, ki so bili parkirani za njim, v hipu pomaknejo naprej, tako da med parkiranimi tovornjaki ni lukenj (pa tudi ne med začetkom parkirišča in prvim tovornjakom). Za prihod ali odhod tovornjaka predpostavimo, da se zgodi v hipu, neskončno hitro. Mogoče je, da ob istem času pride in/ali odide več tovornjakov (v tem primeru moramo ravnati tako, kot da se najprej zgodijo odhodi, nato pa prihodi).

Opiši postopek (ali napiši podprogram oz. funkcijo, če ti je lažje), ki izračuna, kako dolgo parkirišče potrebujemo, da bo na njem vedno dovolj prostora za vse hkrati parkirane tovornjake. Za vhodne podatke bodo veljale naslednje omejitve: tovornjakov je največ milijon, dolžina posameznega tovornjaka je celo število od 1 do 1000, časi p_i in t_i pa so cela števila od 1 do 10^9 (recimo, da so podani v mikrosekundah od nekega izbranega začetnega trenutka naprej).

5. Barvanje stolpnic

Vzdolž ravne ulice so postavili n stolpnic samih različnih višin. Višino i-te stolpnice označimo s h_i. Rekli bomo, da sta stolpnici i in j *povezani*, če so vse stolpnice med njima nižje od njiju, torej če za vsak k z območja $i < k < j$ velja $h_k < h_i$ in tudi $h_k < h_j$. Arhitekti želijo poživiti sive stolpnice s svetlimi barvami fasad, pri tem pa nobeni dve povezani stolpnici ne smeta biti enake barve. **Opiši postopek** (ali napiši podprogram oz. funkcijo, če ti je lažje), ki določi barve stolpnic v skladu s tem pravilom in pri tem uporabi najmanjše možno število različnih barv. (Za rešitev, ki morda včasih uporabi več kot toliko barv, lahko še vedno dobiš delne točke.) V opisu postopka tudi dobro utemelji, zakaj tvoj postopek res vrača pravilne rezultate. Za predstavitev barv uporabi kar zaporedna naravna števila od 1 naprej. Kot vhodne podatke dobi tvoj postopek število stolpnic n in zaporedje višin $h_1, h_2, \ldots, h_n$. Tvoj postopek naj bo učinkovit, da bo deloval hitro tudi za velike n (npr. nekaj tisoč stolpnic).

17. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

28. januarja 2022

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Seštevanje ulomkov

Naloga pravi, da vsak otrok razen zadnjega prereže prejeti kos na pol in potem eno polovico mogoče posadi v zemljo; rezov je torej $n - 1$, zato je najmanjši možni posajeni kos težak $1/2^{n-1}$ kilograma. To maso bomo pri naših izračunih vzeli za osnovno enoto; vse druge mase, s kateri bomo imeli pri tej nalogi opravka, so večkratniki te osnovne enote, zato bomo lahko ves čas računali s celimi števili in nam ne bo treba skrbeti, da bi nam rezultat pokvarile kakšne zaokrožitvene napake.

Za torto, težko 1 kilogram, bomo torej zdaj rekli, da je težka 2^{n-1} enot. Kos, ki ga posadi prvi otrok (če se odloči saditi in ne jesti), je potem težak 2^{n-2} enot; kos, ki ga posadi drugi otrok, je težak 2^{n-3} enot in tako naprej. V splošnem je torej kos, ki ga posadi i-ti otrok, težak 2^{n-1-i} enot.

Podatke o praznovanjih berimo v zanki, pri vsakem praznovanju pa pojdimo v vgnezdeni zanki po otrocih in seštevajmo mase posajenih kosov torte. Na koncu tega postopka imamo pred sabo skupno maso posajenih kosov; recimo, da je to v enot po $1/2^{n-1}$ kilograma; če delimo v z 2^{n-2} in rezultat zaokrožimo navzdol, dobimo skupno maso v enotah po pol kilograma. Če je ta rezultat lih, to pomeni, da je od zadnjega kilograma prisotna vsaj polovica, torej moramo pri zaokrožanju na kilogram zaokrožiti navzgor (kar lahko naredimo tako, da masi v polkilogramskih enotah prištejemo 1, preden jo nazadnje delimo z 2, da dobimo maso v kilogramskih enotah).

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

int main()
{
  // Preberimo število praznovanj in otrok.
  int r, n; cin >> r >> n;

  // Obdelajmo vsa praznovanja.
  int vsota = 0;
  while (r-- > 0)
  {
    // Preberimo niz z opisom praznovanja.
    string s; cin >> s;

    // Prištejmo posajene kose k vsoti.
    for (int i = 0; i < n; ++i)
      if (s[i] == 'S') vsota += 1 << (n - 2 - i);
  }
  // Zaokrožimo vsoto navzdol na polovico kilograma.
  vsota >>= n - 2;

  // Če je število polkilogramskih enot liho, moramo pri
  // pretvorbi v kilograme zaokrožiti navzgor.
  vsota = (vsota + (vsota & 1)) >> 1;

  // Izpišimo rezultat.
  cout << vsota << endl; return 0;
}
```

Lahko pa nalogo rešimo tudi z uporabo aritmetike s plavajočo vejico, saj imajo ulomki, s kakršnimi delamo pri tej nalogi, vsi za imenovalec neko potenco števila 2, take pa

lahko v predstavitvi s plavajočo vejico predstavimo brez napak, dokler je v mantisi dovolj prostora. Pri naši nalogi imamo opravka z vrednostmi od 0 do r in potrebujemo natančnost do $1/2^{n-1}$, kar pri $r = 1000$ in $n = 20$ pomeni kakšnih 29 bitov; tip **double** ima mantiso dolgo 52 bitov, torej več kot dovolj, da do zaokrožitvenih napak ne bo prihajalo. Za izpis vsote v kilogramih na koncu lahko uporabimo funkcijo round iz standardne knjižnice, ki polovice zaokroža stran od 0, prav to pa naša naloga tudi zahteva.

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

int main()
{
  // Preberimo število praznovanj in otrok.
  int r, n; cin >> r >> n;

  // Obdelajmo vsa praznovanja.
  double vsota = 0;
  while (r-- > 0)
  {
    // Preberimo niz z opisom praznovanja.
    string s; cin >> s;

    // Prištejmo posajene kose k vsoti.
    double kos = 1;
    for (int i = 0; i < n; ++i) {
      kos /= 2;
      if (s[i] == 'S') vsota += kos; }
  }

  // Izpišimo vsoto, zaokroženo na celo število kilogramov.
  cout << round(vsota) << endl; return 0;
}
```

Zapišimo še rešitev v pythonu. Lahko bi računali s celimi števili kot pri prvi od gornjih dveh rešitev ali pa s plavajočo vejico (v pythonu sta primerna tipa float in Decimal) kot pri drugi, lahko pa namesto tega uporabimo tip Fraction, s katerim lahko brez zaokrožitvenih napak računamo z racionalnimi števili (ulomki):

```
import sys, fractions, math

# Preberimo število praznovanj in otrok.
r, n = (int(s) for s in sys.stdin.readline().split())

# Obdelajmo vsa praznovanja.
vsota = 0
for ri in range(r):

  # Preberimo niz z opisom praznovanja.
  s = sys.stdin.readline()

  # Prištejmo posajene kose k vsoti.
  for i in range(n):
    if s[i] == 'S': vsota = vsota + fractions.Fraction(1, 2**(i + 1))

# Vsota je zdaj racionalno število v kilogramih.
# Zaokrožimo jo navzdol na celo število v polovicah kilograma.
vsota = math.floor(2 * vsota)

# Če je število polkilogramskih enot liho, moramo pri
# pretvorbi v kilograme zaokrožiti navzgor.
print((vsota + (vsota % 2)) // 2)
```

Za zaokrožanje nismo uporabili funkcije round, kajti ta v pythonu zaokroža polovice na najbližje sodo število (npr. 3,5 $\mapsto$ 4 in tudi 4,5 $\mapsto$ 4) in ne vedno navzgor, kot zahteva naša naloga.

To rešitev lahko zapišemo tudi krajše:

```
import sys, fractions, math

# Preberimo število praznovanj in otrok.
r, n = (int(s) for s in sys.stdin.readline().split())

# Obdelajmo vsa praznovanja.
vsota = math.floor(sum(fractions.Fraction(1, 2**i)
                       for ri in range(r) for i, c in enumerate(sys.stdin.readline()) if c == 'S'))

# Vsota je zdaj zaokrožena navzdol na celo število polkilogramskih enot.
# Če je to število liho, moramo pri pretvorbi v kilograme zaokrožiti navzgor.
print((vsota + (vsota % 2)) // 2)
```

2. Slovar

(a) Prvi del naloge je enostaven: pojdimo v zanki po vseh besedah slovarja; pri vsaki besedi najprej preverimo, če je enako dolga kot vzorec; če ni, lahko takoj zaključimo, da se ne bo ujemala z danim vzorcem; sicer pa pojdimo v vgnezdeni zanki po črkah besede in vzorca hkrati ter preverjajmo, če se ujemajo. Slednje pomeni, da mora imeti bodisi vzorec na tistem mestu zvezdico bodisi morata imeti vzorec in beseda na tistem mestu enako črko. Čim opazimo kakšno neujemanje, lahko preverjanje trenutne besede prekinemo, saj že vemo, da ne bo ustrezala našemu vzorcu. Če pa pridemo do konca besede, ne da bi opazili kakšno neujemanje, jo izpišimo.

Primer implementacije take rešitve v C++ (recimo, da slovar dobimo kot vektor nizov):

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

void PoisciA(const vector<string>& slovar, const string& vzorec)
{
  int d = vzorec.length();
  for (const auto &beseda : slovar)
  {
    // Preverimo, če je beseda enako dolga kot vzorec.
    if (beseda.length() != d) continue;

    // Preverimo, če se znaki besede ujemajo z istoležnimi znaki vzorca.
    int i = 0; while (i < d && (vzorec[i] == '*' || vzorec[i] == beseda[i])) ++i;

    // Če smo prišli do konca, ne da bi opazili kakšno neujemanje,
    // je beseda ustrezna in jo izpišimo.
    if (i >= d) cout << beseda << endl;
  }
}
```

Zapišimo to rešitev še v pythonu; slovar tu pričakujemo kot seznam (pythonov tip list) oz. kot karkoli, po čemer je mogoče iterirati:

```
def PoisciA(slovar, vzorec):
  d = len(vzorec)
  for beseda in slovar:

    # Preverimo, če je beseda enako dolga kot vzorec.
    if len(beseda) != d: continue

    # Preverimo, če se znaki besede ujemajo z istoležnimi znaki vzorca.
    i = 0
    while i < d and (vzorec[i] == '*' or vzorec[i] == beseda[i]): i += 1

    # Če smo prišli do konca, ne da bi opazili kakšno neujemanje,
    # je beseda ustrezna in jo izpišimo.
    if i >= d: print(beseda)
```

Še ena možnost je, da uporabimo regularne izraze (ti so, mimogrede, na voljo tudi v C++ovi standardni knjižnici), pri čemer pa pazimo na to, da se v njih za ujemanje s poljubnim znakom uporablja pika in ne zvezdica:

```
import re
def PoisciA2(slovar, vzorec):
  r = re.compile(vzorec.replace("*", "."))
  for beseda in slovar:
    if r.match(beseda): print(beseda)
```

Še ena možnost v pythonu pa je modul fnmatch, pri katerem se za ujemanje s poljubnim znakom uporablja vprašaj namesto zvezdice:

```
import fnmatch
def PoisciA3(slovar, vzorec):
  vzorec = vzorec.replace("*", "?")
  for beseda in slovar:
    if fnmatch.fnmatch(beseda, vzorec): print(beseda)
```

(b) Razmislimo zdaj o drugem delu naloge, pri katerem bi radi slovar vnaprej predelali tako, da bomo lahko po njem čim hitreje iskali besede, ki se ujemajo z vzorci. Za začetek je koristno besede slovarja razdeliti na več seznamov po dolžini, saj se lahko vzorec dolžine d ujema samo z besedami dolžine d, daljših ali krajših besed pa sploh nima smisla pregledovati.

Recimo zdaj, da i-ti znak vzorca ni zvezdica, ampak neka črka, recimo c. Potem pridejo v poštev le tiste besede iz slovarja, ki imajo na i-tem mestu prav to črko in ne kakšne druge. Koristno bi bilo torej za vsako dolžino d, za vsak indeks i od 1 do d in za vsako črko abecede c pripraviti seznam tistih besed iz slovarja, ki so dolge natanko d znakov in imajo na i-tem mestu črko c; recimo temu seznamu $A(d, i, c)$.

Ko hočemo potem za neki vzorec (dolžine d) poiskati besede, ki se ujemajo z njim, poiščimo v njem poljubno mesto i, kjer ni zvezdice, ampak neka črka c; potem moramo le pregledati vse besede iz $A(d, i, c)$ in za vsako od njih preveriti, ali se ujema tudi s preostankom vzorca. Za to lahko uporabimo enak postopek kot pri podnalogi (a). Če je ima vzorec na več mestih črko (in ne zvezdice), imamo torej na voljo več primernih $A(d, i, c)$ in je smiselno pregledati najkrajšega med njimi, da bomo imeli čim manj dela.

Poseben primer nastopi, če v vzorcu ni nobene črke, ampak same zvezdice. Tedaj nam razmislek iz prejšnjega odstavka ne ponudi nobenega seznama možnih kandidatov, pač pa gremo lahko pri poljubnem i po vseh črkah abecede in za vsako črko c izpišemo vse besede iz $A(d, i, c)$; tako bomo sčasoma izpisali prav vse besede dolžine d, to pa je pri takem vzorcu tudi pravilni odgovor.

```
int maxDolzina;
vector<vector<string>> seznami;

void Pripravi(const vector<string>& slovar)
{
  // Poglejmo, kako dolga je najdaljša beseda.
  maxDolzina = 0;
  for (const auto &beseda : slovar) maxDolzina = max(maxDolzina, int(beseda.length()));
  // Pripravimo si dovolj seznamov.
  seznami.clear(); seznami.resize((maxDolzina + 1) * (maxDolzina + 1) * 26);
  // Pri vsaki besedi pojdimo po vseh njenih črkah in jo dodajmo
  // v ustrezni seznam.
  for (const auto &beseda : slovar)
    for (int d = beseda.size(), i = 0; i < d; ++i)
      seznami[(d * (maxDolzina + 1) + i) * 26 + (beseda[i] - 'a')].push_back(beseda);
}

void PoisciB(const string& vzorec) const
{
  // Morda je ta vzorec daljši od vseh besed v slovarju.
  int d = vzorec.length(); if (d > maxDolzina) return;
  // Poglejmo, pri katerem i dobimo najkrajši seznam kandidatov.
  int najIdx = -1, najDolz = 0;
  for (int i = 0; i < d; ++i) if (vzorec[i] != '*') {
```

```
    int idx = (d * (maxDolzina + 1) + i) * 26 + (vzorec[i] - 'a');
    int dolzina = seznami[idx].size();
    if (najIdx < 0 || dolzina < najDolz) najIdx = idx, najDolz = dolzina; }
  // Če smo tak seznam našli, ga preiščimo.
  if (najIdx >= 0) { PoisciA(seznami[najIdx], vzorec); return; }
  // Sicer ima vzorec same zvezdice in moramo izpisati vse nize dolžine d.
  for (int c = 0; c < 26; ++c)
    for (const auto &beseda : seznami[d * (maxDolzina + 1) * 26 + c])
      cout << beseda << endl;
}
```

Še podobna rešitev v pythonu:

```
def Pripravi(slovar):
  global maxDolzina, seznami
  # Poglejmo, kako dolga je najdaljša beseda.
  maxDolzina = max(len(beseda) for beseda in slovar)
  # Pripravimo si dovolj seznamov.
  seznami = [[[[] for c in range(26)] for i in range(d)] for d in range(maxDolzina + 1)]
  # Pri vsaki besedi pojdimo po vseh njenih črkah in jo dodajmo v ustrezni seznam.
  for beseda in slovar:
    d = len(beseda)
    for i in range(d):
      seznami[d][i][ord(beseda[i]) - ord('a')].append(beseda)

def PoisciB(vzorec):
  # Morda je ta vzorec daljši od vseh besed v slovarju.
  d = len(vzorec)
  if d > maxDolzina: return
  # Med seznami kandidatov za vsako črko vzorca preiščimo najkrajšega.
  kandidati = min((seznami[d][i][ord(c) - ord('a')] for i, c in enumerate(vzorec) if c != '*'),
                  key = len, default = None)
  if kandidati is not None: PoisciA(kandidati, vzorec); return
  # Če ni nobenega takega seznama, ker ima vzorec same zvezdice,
  # moramo izpisati vse nize dolžine d.
  for seznam in seznami[d][0]:
    for beseda in seznam: print(beseda)
```

3. Genialno

Preprosta, vendar manj učinkovita rešitev je, da gremo v zanki po vseh poljih mreže in pri vsakem od njih izračunamo število točk po pravilu iz besedila naloge. To pomeni, da potrebujemo še eno vgnezdeno zanko, ki gre po vseh štirih možnih smereh (gor, dol, levo in desno), za vsako smer pa imamo potem še eno zanko, ki se premika od trenutnega polja v tisti smeri in sešteva število žetonov na tako obiskanih poljih, ustavi pa se, ko bodisi pride do praznega polja (takega z nič žetoni) ali pa doseže rob mreže.

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

// a[y * w + x] = število žetonov na polju (x, y) za 0 ≤ x < w, 0 ≤ y < h.
pair<int, int> KamPostaviti(int w, int h, const vector<int> &a)
{
  int xNaj = -1, yNaj = -1, najTock = -1;
  const int DX[] = { -1, 1, 0, 0 }, DY[] = { 0, 0, 1, -1 };
  for (int y = 0; y < h; ++y) for (int x = 0; x < w; ++x)
  {
    int tocke = 0;
    // Pojdimo iz polja (x, y) v vse štiri smeri.
    for (int smer = 0; smer < 4; ++smer) for (int xx = x, yy = y; ; )
```

```
    {
      // Naredimo korak naprej v trenutno smer.
      xx += DX[smer]; yy += DY[smer];
      // Če smo padli čez rob mreže, končajmo.
      if (xx < 0 || xx >= w || yy < 0 || yy >= h) break;
      // Če smo na praznem polju, tudi končajmo.
      int zetoni = a[yy * w + xx]; if (zetoni == 0) break;
      // Sicer bomo dobili toliko točk, kolikor je tu žetonov.
      tocke += zetoni;
    }
    // Najboljši rezultat doslej si zapomnimo.
    if (tocke > najTock) najTock = tocke, xNaj = x, yNaj = y;
  }
  return {xNaj, yNaj};
}
```

Zapišimo podobno rešitev še v pythonu:

```
# a[y * w + x] = število žetonov na polju (x, y) za 0 ≤ x < w, 0 ≤ y < h.
def KamPostaviti(w, h, a):
  # Vrne točke, ki jih dobimo iz ene od smeri, če postavimo žeton na (x, y).
  def TockeS(x, y, smer):
    DX = [−1, 1, 0, 0][smer]; DY = [0, 0, 1, −1][smer]
    tocke = 0
    while True:
      # Naredimo korak naprej v trenutno smer.
      x += DX; y += DY
      # Če smo padli čez rob mreže, končajmo.
      if x < 0 or x >= w or y < 0 or y >= h: break
      # Če smo na praznem polju, tudi končajmo.
      zetoni = a[y * w + x]
      if zetoni == 0: break
      # Sicer bomo dobili toliko točk, kolikor je tu žetonov.
      tocke += zetoni
    return tocke
  # Vrne skupno število točk, ki ga dobimo, če postavimo žeton na (x, y).
  def Tocke(x, y): return sum(TockeS(x, y, smer) for smer in range(4))
  # Izračunajmo najboljši položaj novega žetona in ga vrnimo.
  return max((Tocke(x, y), (x, y)) for y in range(h) for x in range(w))[1]
```

Slabost dosedanje rešitve je, da je razmeroma počasna. V najslabšem primeru (če so vsa polja mreže neprazna) se bo najbolj notranja zanka vedno premikala vse do roba mreže, zato bomo pri vsakem (x, y) pregledali celotno vrstico y in celoten stolpec x, skupaj torej $w + h$ polj; časovna zahtevnost te rešitve je torej $O(wh(w + h))$.

Toda predstavljajmo si maksimalno strnjeno skupino nepraznih polj, ki so vsa v isti vrstici, recimo od (x_1, y) do (x_2, y). Ko pravimo, da naj bo maksimalna, hočemo s tem reči, naj na levi in desni mejí bodisi na rob mreže bodisi na prazno polje. Recimo, da razmišljamo o tem, da bi žeton postavili na neko polje (x, y) te skupine (torej za $x_1 \leq x \leq x_2$); s premikanjem levo in desno od njega bomo obiskali ravno celo skupino, preden se bomo ustavili. To pomeni, da vsota vseh žetonov te skupine prispeva k točkam vsakega polja v skupini. Podobno je tudi, če razmišljamo o postavitvi žetona tik levo od skupine, na $(x_1 - 1, y)$; s premikanjem desno od tam bomo obiskali ravno celo skupino. Podobno je tudi pri postavljanju žetona tik desno od skupine, na (x_2+1, y), ko obiščemo celo skupino med premikanjem levo.

Vidimo torej, da ko enkrat najdemo tako skupino, lahko izračunamo vsoto žetonov na njej in jo prištejemo k točkam vseh polj skupine in še sosednjih polj pred in za skupino. Če je skupina dolga d polj, smo imeli z njo $O(d)$ dela; in ker pripada vsako neprazno polje natanko eni taki skupini, so vse skupine skupaj dolge $O(wh)$ polj, zato imamo tudi z vsemi skupaj le $O(wh)$ dela. Tako sčasoma dobimo vse točke (za vse možne položaje

novega žetona), ki nastanejo zaradi premikanja levo in desno od položaja novega žetona. V nadaljevanju moramo popolnoma enak razmislek ponoviti še po stolpcih namesto po vrsticah, da bomo upoštevali še točke, ki nastanejo zaradi premikanja gor in dol, pa bomo na koncu dobili pravo število točk vsake celice.

Tako imamo rešitev s časovno zahtevnostjo $O(wh)$, kar je veliko manj od prejšnje; njena slabost pa je v večji porabi pomnilnika, saj potrebujemo tabelo $w \cdot h$ elementov, v kateri računamo točke vsake celice.

```
pair<int, int> KamPostaviti2(int w, int h, const vector<int> &a)
{
  vector<int> tocke(w * h, 0);
  for (int obrni = 0; obrni < 2; ++obrni)
  {
    // Spodnji postopek je tak, kot da iščemo vodoravne skupine neničelnih polj;
    // toda pri obrni = 1 koordinatne osi obrnemo, tako da v resnici iščemo navpične skupine.
    int W = w, H = h; if (obrni) swap(W, H);
    auto A = [=, &a] (int X, int Y) { return a[obrni ? X * w + Y : Y * w + X]; };

    for (int Y = 0; Y < H; ++Y) for (int X = 0; X < W; ++X) if (A(X, Y) != 0)
    {
      // Tu se začne skupina neničelnih polj.
      // Kako daleč sega in kakšna je njihova vsota?
      int XX = X + 1, vsota = A(X, Y);
      while (XX < W && A(XX, Y) != 0) vsota += A(XX++, Y);

      // Prištejmo to vsoto k točkam polj v skupini in še tistih tik pred in za njo.
      for (int t = max(X - 1, 0); t < min(XX + 1, W); ++t)
        tocke[obrni ? t * w + Y : Y * w + t] += vsota - A(t, Y);

      X = XX; // Nadaljujmo za koncem te skupine.
    }
  }
  // Vrnimo najboljši položaj novega žetona.
  int naj = 0; for (int i = 1; i < w * h; ++i) if (tocke[i] > tocke[naj]) naj = i;
  return {naj % w, naj / w};
}
```

Zapišimo to rešitev še v pythonu:

```
def KamPostaviti2(w, h, a):
  tocke = [0] * (w * h)
  for obrni in range(2):
    # Spodnji postopek je tak, kot da iščemo vodoravne skupine neničelnih polj;
    # toda pri obrni = 1 koordinatne osi obrnemo, tako da v resnici iščemo navpične skupine.
    W, H = (h, w) if obrni else (w, h)
    def A(X, Y): return a[X * w + Y if obrni else Y * w + X]

    for Y in range(H):
      X = 0
      while X < W:
        vsota = A(X, Y)
        if vsota == 0: X += 1; continue

        # Tu se začne skupina neničelnih polj.
        # Kako daleč sega in kakšna je njihova vsota?
        XX = X + 1
        while XX < W and A(XX, Y) != 0: vsota += A(XX, Y); XX += 1

        # Prištejmo to vsoto k točkam polj v skupini in še tistih tik pred in za njo.
        for t in range(max(0, X - 1), min(XX + 1, W)):
          tocke[t * w + Y if obrni else Y * w + t] += vsota - A(t, Y)

        X = XX + 1 # nadaljujmo za to skupino
  # Izračunajmo najboljši položaj novega žetona in ga vrnimo.
  naj = max((tocke[i], i) for i in range(w * h))[1]
  return naj % w, naj // w
```

4. Parkirišče

Naloga pravzaprav sprašuje po tem, kakšna je največja skupna dolžina tovornjakov, ki so hkrati parkirani na našem parkirišču. Ta skupna dolžina se spremeni le ob prihodu ali odhodu kakšnega tovornjaka, torej (za vsak i od 1 do n) ob času p_i (ko se skupna dolžina poveča za d_i zaradi prihoda tovornjaka številka i) in ob času p_i+t_i (ko se skupna dolžina zmanjša za d_i zaradi odhoda tovornjaka številka i). Dovolj bo torej, če za vsakega od teh časov izračunamo, kakšna bo skupna dolžina parkiranih tovornjakov po tej spremembi. To pa je najlaže računati po naraščajočih časih, saj lahko dobimo novo dolžino iz stare preprosto tako, da slednji prištejemo d_i ali ga od nje odštejemo.

Pripravimo torej seznam parov (p_i, d_i) in $(p_i + t_i, -d_i)$ za vse $i = 1, \ldots, n$. Prva komponenta pove čas, druga pa spremembo dolžine. Uredimo jih naraščajoče po času, tiste z enakim pasom pa po drugi komponenti. V tako urejenem zaporedju moramo zdaj le seštevati druge komponente, pa bomo imeli pred seboj skupno dolžino po vsaki od sprememb.

Ker smo rekli, da pare z enakim časom uredimo po drugi komponenti, bomo v primerih, ko hkrati neki tovornjak pride in neki drug tovornjak odide, najprej obdelali odhode (kajti pri teh je druga komponenta negativna) in šele potem prihode (pri katerih je druga komponenta pozitivna), prav to pa naloga tudi zahteva.

Če nastopi več sprememb ob istem času, sicer skupna dolžina „med“ temi spremembami nima nobenega smisla (ker so spremembe pač istočasne in naloga tudi pravi, da se zgodijo v hipu), vendar tudi ne bo pokvarila naših rezultatov, saj nas zanima le največja možna skupna dolžina, ta pa bo nastopila bodisi po vseh teh spremembah (ko bodo upoštevani že vsi prihodi ob tem času) bodisi pred njimi (ko ne bo upoštevan še nobeden izmed odhodov ob tem času), ni pa mogoče, da bi bila kakšna od teh nesmiselnih vmesnih skupnih dolžin večja od obeh prej omenjenih.[1]

Oglejmo si še implementacijo te rešitve v C++:

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

struct Tovornjak { int dolzina, prihod, trajanje; };

int Parkirisce(const vector<Tovornjak> &tovornjaki)
{
  // Pripravimo podatke o spremembah skupne dolžine.
  vector<pair<int, int>> spremembe;
  for (const auto &t : tovornjaki) {
    spremembe.emplace_back(t.prihod, t.dolzina);
    spremembe.emplace_back(t.prihod + t.trajanje, -t.dolzina); }
  // Uredimo spremembe po času.
  sort(spremembe.begin(), spremembe.end());
  // Računajmo skupno dolžino po vsaki spremembi.
  int maxDolzina = 0, dolzina = 0;
  for (auto [cas, sprememba] : spremembe) {
    dolzina += sprememba;
    // Najdaljšo skupno dolžino si zapomnimo.
    maxDolzina = max(maxDolzina, dolzina); }
  return maxDolzina;
}
```

Zapišimo to rešitev še v pythonu:

```
# Podprogram Parkirisce pričakuje kot parameter seznam takšnih objektov:
class Tovornjak: __slots__ = ["dolzina", "prihod", "trajanje"]
```

[1] Natančneje povedano: če smo upoštevali že nekaj prihodov ob tem času, ne pa še vseh, se bo skupna dolžina še povečala, ko bomo upoštevali še preostale prihode ob tem času. Če pa smo upoštevali že nekaj odhodov ob tem času, ne pa še vseh, potem je bila skupna dolžina večja, še preden smo upoštevali tistih dosedanjih nekaj odhodov.

```
def Parkirisce(tovornjaki):
  # Pripravimo podatke o spremembah skupne dolžine.
  spremembe = [(t.prihod, t.dolzina) for t in tovornjaki]
  spremembe += [(t.prihod + t.trajanje, −t.dolzina) for t in tovornjaki]

  # Uredimo spremembe po času.
  spremembe.sort()

  # Računajmo skupno dolžino po vsaki spremembi.
  maxDolzina = 0; dolzina = 0
  for cas, sprememba in spremembe:
    dolzina += sprememba
    maxDolzina = max(maxDolzina, dolzina)    # Najdaljšo skupno dolžino si zapomnimo.

  return maxDolzina
```

5. Barvanje stolpnic

Povezanost med stolpnicami, kot je definirana v tej nalogi, ima naslednjo zanimivo lastnost: med stolpnicami, ki so povezane s stolpnico i, sta lahko največ dve taki, ki sta višji od i, in sicer ena levo od i ter ena desno od i; vse ostale stolpnice, s katerimi je i povezana, morajo biti nižje od nje.

O tem se lahko prepričamo takole: recimo, da se od i pomikamo desno; naj bo j prva višja stolpnica (višja od i-te), na katero naletimo. Ker so vmes vse stolpnice nižje od i-te, sta i in j povezani. Recimo zdaj, da bi bila nekje še bolj desno neka stolpnica k, ki bi bila tudi višja od i in povezana z njo; toda to bi pomenilo, da so vse stolpnice med i in k nižje od i (in tudi od k), kar pa ni res, saj je med temi tudi stolpnica j, ki je višja od i. Torej je lahko desno od i največ ena stolpnica, ki je višja od i in povezana z njo (in sicer je to kar prva višja stolpnica, ki stoji desno od i-te). Podoben razmislek lahko potem seveda opravimo tudi za stolpnice levo od i in se prepričamo, da je tudi tam i povezana z največ eno tako, ki je višja od nje (in sicer je to kar prva višja stolpnica levo od i-te). □

Zaradi te lastnosti je koristno barvati stolpnice od višjih proti nižjim. Ko pride na vrsto recimo stolpnica i, so bile doslej že pobarvane le tiste stolpnice, ki so višje od nje; med njimi pa sta z i povezani največ dve. Dodelimo i-ju najmanjšo tako številko barve (spomnimo se, da označujemo barve z naravnimi števili od 1 naprej), ki je še nima nobena od tistih (največ) dveh z njo povezanih stolpnic, ki smo ju že pobarvali. Ena od barv 1, 2 in 3 bo torej gotovo primerna, saj nam tisti dve stolpnici lahko prepovesta največ dve od teh treh barv. Tako torej vidimo, da lahko stolpnice zagotovo pobarvamo s tremi barvami.

Za učinkovito izvedbo tega postopka je koristno razmisliti še o tem, kako naj ugotovimo, s katerima višjima stolpnicama je i povezana; z drugimi besedami, katera je prva višja stolpnica levo od i (recimo ji $L[i]$) in katera desno od i (recimo ji $D[i]$). Preprosta rešitev je, da gremo pri vsakem i v zanki po stolpnicah levo od i, dokler ne naletimo na prvo višjo od i; in potem podobno še v zanki po stolpnicah desno od i. Slabost te rešitve je, da v najslabšem primeru porabi $O(n^2)$ časa.

Boljša možnost je, da ko barvamo stolpnice padajoče po višini, vsako že pobarvano stolpnico dodamo v neko primerno uravnoteženo drevesasto podatkovno strukturo (na primer rdeče-črno drevo; v C++ lahko uporabimo razred `set`), v kateri bodo stolpnice urejene po svojem indeksu. Ko barvamo stolpnico i, so v tem drevesu že vse stolpnice, višje od i (in samo one); poiščimo torej v njem prvo stolpnico z indeksom $> i$, pa bo to ravno prva višja stolpnica desno od i, torej $D[i]$; podobno lahko v drevesu poiščemo zadnjo stolpnico z indeksom $< i$, pa bo to ravno prva višja stolpnica levo od i, torej $L[i]$. Tako imamo pri vsaki stolpnici $O(\log n)$ dela za tidve iskanji po drevesu (pa tudi za dodajanje stolpnice i v drevo, ko jo pobarvamo) in časovna zahtevnost celotnega postopka je $O(n \log n)$.

Še hitreje pa gre, če računamo vrednosti $L[i]$ od leve proti desni. Najbolj leva stolpnica seveda sploh nima nobene višje na svoji levi, zato postavimo $L[1] = 0$. Od tam naprej pri vsakem i razmišljajmo takole: en kandidat za $L[i]$ je i-jeva leva soseda, torej $i - 1$. Če je ta višja od i-te stolpnice, se ustavimo in jo razglasimo za $L[i]$; če pa je $i - 1$ prenizka, je potem naslednji primerni kandidat za $L[i]$ šele prva taka stolpnica, ki stoji levo od $i - 1$ in je višja od stolpnice $i - 1$ — to pa je stolpnica $L[i - 1]$. Če je

tudi ta prenizka (nižja od i-te stolpnice), je naslednji kandidat potem $L[L[i-1]]$ in tako naprej. Zapišimo ta postopek s psevdokodo:

```
for i := 1 to n:
  c := i − 1;
  while c > 0:
    if h_c > h_i then break
    else c := L[c];
  L[i] := c;
```

Na podoben način lahko računamo tudi $D[i]$, le ta gremo tam od desne proti levi. Kakšna je časovna zahtevnost tega postopka? Predstavljajmo si, da bi med izvajanjem našega postopka vzdrževali sklad, na katerem so stolpnice i, $L[i]$, $L[L[i]]$ in tako naprej, vse višje in vse bolj leve, dokler se pač to zaporedje ne konča pri neki stolpnici j, ki ima $L[j] = 0$. Zgornji postopek potem pravzaprav pri i naredi naslednje: z vrha sklada pobere stolpnice, nižje od h_i, in nato na vrh sklada doda stolpnico i. Ker torej vsako stolpnico enkrat dodamo na sklad, jo tudi največ enkrat poberemo s sklada, zato je časovna zahtevnost vseh teh operacij skupaj le $O(n)$.

S tem, ko smo vse $L[i]$ in $D[i]$ izračunali v $O(n)$ časa namesto $O(n \log n)$ kot pri rešitvi z drevesom, sicer nismo veliko pridobili, saj bomo stolpnice kasneje še vedno barvali padajoče po višini, torej jih moramo še vedno urediti in to nam bo še vedno vzelo $O(n \log n)$ časa. Ima pa zadnji postopek za izračun $L[i]$ in $D[i]$ vseeno to prednost, da ne potrebuje drevesa (kar je koristno npr. v pythonu, kjer v standardni knjižnici takega drevesa nimamo).[2]

Zdaj torej znamo pobarvati stolpnice s kvečjemu tremi barvami, ostane pa še vprašanje, ali obstaja kakšno barvanje z manj barvami od tistega, ki ga najde naš dosedanji postopek. Pri $n = 0$ ali $n = 1$ je možno seveda barvanje z n barvami in naš postopek ga bo tudi našel; pri $n \geq 2$ pa gotovo potrebujemo vsaj dve barvi, saj sta najvišji dve stolpnici med seboj povezani. Pa recimo, da pri nekem razporedu stolpnic zadoščata že samo dve barvi, naš postopek pa jih pobarva s tremi. Naš postopek najvišji stolpnici vedno dá barvo 1; vzemimo neko barvanje z dvema barvama (seveda takšno, v katerem nobeni dve povezani stolpnici nista iste barve) in če najvišja stolpnica v njem nima barve 1, preprosto obrnimo barve vseh stolpnic v njem. Naj bo zdaj $d = (d_1, \ldots, d_n)$ zaporedje barv stolpnic v tem barvanju z dvema barvama, $t = (t_1, \ldots, t_n)$ pa v barvanju s tremi barvami, ki ga je našel naš postopek.

Pojdimo zdaj po stolpnicah od višjih proti nižjim in primerjajmo d_i in t_i; pri prvih nekaj stolpnicah se barve mogoče ujemajo, prej ali slej pa mora nastopiti razlika, recimo pri stolpnici i. Neujemanje med t_i in d_i lahko nastopi na dva načina: (1) lahko da je $t_i = 3$; toda če je naš postopek uporabil barvo 3, to pomeni, da je i povezana z dvema višjima stolpnicama j in k, ki sta bili barv 1 in 2; in ker se pri višjih stolpnicah t in d ujemata, imata j in k barvi 1 in 2 tudi v d; in ker je v d tudi stolpnica i ene od teh dveh barv, sta tam dve povezani stolpnici enake barve, kar je protislovje. (2) Neujemanje lahko nastopi torej le tako, da je $t_i = 3 - d_i$; toda ker i ni najvišja, je gotovo povezana z vsaj eno višjo stolpnico, recimo j; ta ima v obeh barvanjih enako barvo, $t_j = d_j$; ker sta i in j povezani, naš postopek ni pobarval i z enako barvo kot j, torej je $t_i = 3 - t_j = 3 - d_j$, in ko to združimo z $t_i = 3 - d_i$, vidimo, da mora biti $d_i = d_j$, torej sta v d dve povezani stolpnici enake barve, kar je spet protislovje.

Vidimo torej, da nas predpostavka, da je naš postopek porabil tri barve na takem razporedu, ki ga je mogoče pobarvati že z dvema, neizogibno pripelje v protislovje, torej naš postopek na takem razporedu res porabi samo dve barvi. □

Čeprav naloga zahteva le opis postopka, si vseeno oglejmo še primer implementacije v C++; najprej rešitev z drevesom:

```
#include <set>
#include <vector>
```

[2] Poleg tega, če morda višine stolpnic tvorijo permutacijo števil od 1 do n ali kaj podobnega (česar sicer naloga ne zagotavlja), jih lahko uredimo že v $O(n)$ časa in je zato naš zadnji postopek za izračun $L[i]$ in $D[i]$ še tem koristnejši.

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

void Pobarvaj(const vector<int> &h, vector<int> &barva)
{
  int n = h.size(); barva.resize(n);

  // Uredimo stolpnice padajoče po višini.
  vector<pair<int, int>> vrstniRed(n);
  for (int i = 0; i < n; ++i) vrstniRed[i] = {h[i], i};
  sort(vrstniRed.begin(), vrstniRed.end(), greater<pair<int, int>>());
  set<int> pobarvane;

  for (auto [hi, i] : vrstniRed)
  {
    int b = 0; // V b bomo s prižiganjem bitov označili, katerih barv ne smemo uporabiti.

    // Kakšne barve je naslednja višja stolpnica desno od i?
    auto it = pobarvane.upper_bound(i);
    if (it != pobarvane.end()) b |= 1 << barva[*it];

    // Kakšne barve je naslednja višja stolpnica levo od i?
    if (it != pobarvane.begin()) b |= 1 << barva[*--it];

    // Pripišimo stopnici i prvo prosto barvo.
    barva[i] = (b & 2) == 0 ? 1 : (b & 4) == 0 ? 2 : 3;
    pobarvane.emplace(i);
  }
}
```

In še rešitev, ki vse vrednosti $L[i]$ in $D[i]$ izračuna v $O(n)$ časa:

```
void Pobarvaj2(const vector<int> &h, vector<int> &barva)
{
  int n = h.size(); barva.resize(n);

  // Za vsako stolpnico poiščimo naslednjo višjo na levi in desni.
  vector<int> L(n), D(n);
  for (int i = 0; i < n; ++i) for (L[i] = i - 1; L[i] >= 0 && h[L[i]] < h[i]; L[i] = L[L[i]]) ;
  for (int i = n-1; i >= 0; --i) for (D[i] = i+1; D[i] < n && h[D[i]] < h[i]; D[i] = D[D[i]]) ;

  // Uredimo stolpnice padajoče po višini.
  vector<pair<int, int>> vrstniRed(n);
  for (int i = 0; i < n; ++i) vrstniRed[i] = {h[i], i};
  sort(vrstniRed.begin(), vrstniRed.end(), greater<pair<int, int>>());

  for (auto [hi, i] : vrstniRed)
  {
    int b = 0; // V b bomo s prižiganjem bitov označili, katerih barv ne smemo uporabiti.

    // Kakšne barve je naslednja višja stolpnica levo od i?
    if (L[i] >= 0) b |= 1 << barva[L[i]];
    // Kakšne barve je naslednja višja stolpnica desno od i?
    if (D[i] < n) b |= 1 << barva[D[i]];

    // Pripišimo stopnici i prvo prosto barvo.
    barva[i] = (b & 2) == 0 ? 1 : (b & 4) == 0 ? 2 : 3;
  }
}
```

Zapišimo še primer implementacije te druge rešitve v pythonu.

```
def Pobarvaj2(visine):
  n = len(visine); barve = [-1] * n

  # Za vsako stolpnico poiščimo naslednjo višjo na levi in desni.
  L = [i - 1 for i in range(n)]; D = [i + 1 for i in range(n)]
  for i in range(n):
    while L[i] >= 0 and visine[L[i]] < visine[i]: L[i] = L[L[i]]
  for i in range(n - 1, -1, -1):
    while D[i] < n and visine[D[i]] < visine[i]: D[i] = D[D[i]]
```

```
# Preglejmo stolpnice padajoče po višini.
for (hi, i) in sorted(((visine[i], i) for i in range(n)), reverse = True):
    b = 0 # V b bomo s prižiganjem bitov označili, katerih barv ne smemo uporabiti.
    # Kakšne barve je naslednja višja stolpnica levo od i?
    if L[i] >= 0: b |= 1 << barve[L[i]]
    # Kakšne barve je naslednja višja stolpnica desno od i?
    if D[i] < n: b |= 1 << barve[D[i]]
    # Pripišimo stopnici i prvo prosto barvo.
    barve[i] = 1 if (b & 2) == 0 else 2 if (b & 4) == 0 else 3
return barve
```

17. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

28. januarja 2022

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. Seštevanje ulomkov

- Ker je ta naloga mišljena kot lahka, ne pričakujemo, da bodo tekmovalci kaj dosti razmišljali o tem, ali lahko pride pri uporabi aritmetike s plavajočo vejico do kakšnih napak zaradi omejene natančnosti pri predstavitvi ne-celih števil. Rešitev, ki uporablja tip **double** (ali `float` v pythonu), lahko dobi vse točke prav tako kot rešitev, ki računa le s celimi števili.

- Naloga zahteva, da rezultat zaokrožimo na najbližji kilogram. Rešitvam, ki rezultata ne zaokrožijo ali pa vedno zaokrožijo navzdol, naj se zaradi tega odšteje pet točk. Rešitvam, ki napačno zaokrožijo le polovične kilograme (npr. navzdol namesto navzgor; ali pa če uporabijo pythonovo funkcijo round, ki zaokroža k najbližjemu sodemu številu), naj se zaradi tega odšteje dve točki.

2. Slovar

- V nekaterih programskih jezikih je morda že v standardni knjižnici kakšna funkcija za preverjanje, ali se niz ujema s takim vzorcem, kakršne uporabljamo pri tej nalogi; v tem primeru ni nič narobe, če si rešitev pomaga s takšno funkcijo, namesto da bi sama v zanki primerjala znake niza z znaki vzorca. (Primer: v pythonu lahko uporabimo modul fnmatch, le da moramo v naših vzorcih spremeniti zvezdice v vprašaje; ali pa spremenimo zvezdice v pike in potem uporabimo razrede za delo z regularnimi izrazi, ki so na voljo tako v C++ kot v pythonu).
- Pri podnalogi (b) je težko vnaprej reči, česa vsega se utegnejo tekmovalci domisliti. Želimo si predvsem, da se uspejo nekako izogniti potrebi po preiskovanju celega slovarja pri vsaki poizvedbi. Rešitev, ki vedno pregleda vse tiste besede, ki so enako dolge kot vzorec, naj dobi pri tej podnalogi 4 točke od sedmih.
- V primerih, ko ima vzorec več zaporednih črk, bi se dalo dosedanjo rešitev podnaloge (b) še izboljšati tako, da bi besede slovarja oz. njihove sufikse zlagali v drevesa po črkah (*trie*) in s pomočjo takih dreves prišli do še manjših seznamov kandidatov (dobili bi vse besede, ki se ujemajo z vzorcem v več zaporednih črkah, ne le v eni črki kot pri dosedanji rešitvi). Vendar pa od tekmovalcev ne pričakujemo, da bodo razmišljali o čem takem.
- Rešitev sme narediti razumne predpostavke o tem, kakšne besede so v slovarju; na primer, da niso daljše od nekaj 10 znakov.
- Od tekmovalcev ne pričakujemo, da vedo, koliko črk ima abeceda ali kakšne so njihove kode v tabeli ASCII; rešitvam, ki glede teh stvari vsebujejo napačne konstante, naj se zaradi tega odšteje največ eno točko.

3. Genialno

- Pri tej nalogi pričakujemo večinoma rešitve s časovno zahtevnostjo $O(wh(w+h))$; take naj dobijo največ 15 točk, če so sicer pravilne. Rešitve z manjšo časovno zahtevnostjo, kot je na primer naša rešitev v času $O(wh)$, naj dobijo vse točke, če so sicer pravilne.
- Če bi rešitev pri računanju, koliko točk dobimo ob postavitvi žetona na določeno mesto, prištela zraven tudi število dosedanjih žetonov na tem mestu, naj se ji zaradi tega odšteje tri točke.
- Rešitev lahko prebere opis mreže s standardnega vhoda, iz datoteke ali pa predpostavi, da ga dobi v neki tabeli, seznamu, vektorju ali čem podobnem; vse te možnosti so enako dobre.

4. Parkirišče

- Pri tej nalogi je pomembna predvsem ideja, da sledimo spremembam (prihodom in odhodom tovornjakov) naraščajoče po času. Ni pa mišljeno, da bi se tekmovalčeva rešitev kaj ukvarjala s tem, kakšen postopek uporabiti za urejanje časov sprememb.
- Namesto urejanja časov bi se dalo tudi vsakič preiskati celoten seznam tovornjakov, da ugotovimo, ob katerem času pride do naslednje spremembe. To je ekvivalentno rešitvi, ki bi čase prihodov in odhodov uredila z urejanjem z izbiranjem ali kakšnim podobnim postopkom, ki ima časovno zahtevnost $O(n^2)$. Ker učinkovitost urejanja ni predmet te naloge, lahko tudi take rešitve dobijo vse točke, če so sicer pravilne.

- Naloga posebej omenja, da lahko hkrati pride in odide več tovornjakov in da se v takem primeru šteje, kot da so se odhodi zgodili pred prihodi. Če kakšna rešitev tega ne upošteva in lahko zato dobi napačen rezultat (npr. ker se za hip zdi, da so novi tovornjaki prišli, stari pa še niso odšli in da zato potrebujemo daljše pakririšče), naj se ji zaradi tega odšteje pet točk.

- Cela števila, s katerimi so predstavljeni časi, so razmeroma velika, da bi spodbudili reševalce k urejanju časov. Če bi kakšna rešitev poskušala uporabiti te čase kot indekse v neko gromozansko tabelo (in v njej označevati, kdaj pride do sprememb v dolžini parkirišča), naj se ji zaradi tega odšteje štiri točke.

5. Barvanje stolpnic

- Pri tej nalogi je poudarek predvsem na opažanju, da je koristno stolpnice barvati od višjih proti nižjim. Zaželeno je, da vsebuje tekmovalčev odgovor tudi nekakšno utemeljitev, zakaj je to res, vendar prav veliko v tej smeri ne moremo pričakovati. Če rešitev ne vsebuje niti poskusa take utemeljitve, naj se ji zaradi tega odšteje štiri točke. V naši rešitvi je tudi dokaz, da naš postopek vedno, ko je to mogoče, najde barvanje z dvema barvama namesto s tremi, vendar tovrstnega dokaza od tekmovalčevega odgovora ne pričakujemo.

- Če barvamo stolpnice v kakšnem drugem vrstnem redu, na primer od leve proti desni, se lahko zgodi, da bomo porabili več kot tri barve; takim rešitvam, ki sicer vrnejo veljavno barvanje (táko, pri katerem nista nobeni dve povezani stolpnici pobarvani z isto barvo), vendar porabijo preveč barv, naj se zaradi tega odšteje 5 točk. (To velja seveda za rešitve, ki se vsaj trudijo uporabiti čim manj barv; ni pa mišljeno, da bi dobili ljudje netrivialno število točk npr. za rešitev, ki preprosto pobarva vsako stolpnico s svojo barvo.)

- Rešitve s časovno zahtevnostjo $O(n^2)$ naj dobijo največ 18 točk, če so drugače pravilne; tiste s časovno zahtevnostjo $O(n^3)$ največ 16 točk; in tiste z eksponentno časovno zahtevnostjo največ 10 točk.

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. Seštevanje ulomkov	lažja naloga v prvi skupini
2. Slovar	srednje težka naloga v prvi ali lažja v drugi skupini[3]
3. Genialno	srednje težka naloga v prvi ali lažja v drugi skupini[4]
4. Parkirišče	srednje težka naloga v drugi ali lažja v tretji skupini
5. Barvanje stolpnic	težja naloga v drugi ali lažja v tretji skupini

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

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

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

[3] Podnaloga (a) sama po sebi bi lahko bila lažja naloga za prvo skupino, podnaloga (b) pa bi bila bolj za drugo skupino.

[4] Težavnost te naloge je odvisna od tega, kako točkujemo rešitve odvisno od njihove učinkovitosti. Naloga bi postala težja, če bi zahtevali rešitev v času $O(wh)$, ker pa smo predvideli 15 točk (od dvajsetih) že za veliko preprostejšo rešitev v času $O(wh(w+h))$, ta naloga ni tako težka.