21. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

23. januarja 2026

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.");
   }
}
```

21. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

23. januarja 2026

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 `https://rtk.ijs.si/`.

1. Napredek

Podan imamo vrstni red istih tekmovalcev na dveh tekmovanjih. Tekmovalcev je n in namesto z imeni so predstavljeni z enoličnimi številkami od 1 do n. Vsak izmed tekmovalcev se pojavlja v obeh vrstnih redih. Primer:

prvo tekmovanje:	$4, 2, 1, 3, 5$
drugo tekmovanje:	$4, 5, 2, 1, 3$

Napiši program, ki bo izračunal, kdo je med prvim in drugim tekmovanjem najbolj napredoval (ima največjo razliko med uvrstitvama). V gornjem primeru je to tekmovalec s številko 5, ki je napredoval za tri mesta. Če je možnih odgovorov več, je vseeno, katerega od njih izpišeš. Tvoja rešitev naj bo učinkovita, da bo delovala hitro tudi za velike n. Podrobnosti tega, v kakšni obliki dobi tvoj program vhodne podatke in kako vrne ali izpiše rezultate, si izberi sam in jih v svoji rešitvi tudi opiši.

2. Tabela medalj

Dan je seznam medalj, ki so jih dobile različne države na nekem mednarodnem tekmovanju. V vsaki vrstici je najprej ime države, nato pa barva medalje (`zlato`, `srebro` ali `bron`). Da bo naloga lažja, predpostavimo, da so imena držav le enobesedna, brez presledkov. Primer vhoda:

```
Kitajska zlato
VB bron
Francija srebro
VB zlato
...
```

Napiši program, ki prebere tak seznam in izpiše tabelo, v kateri bo po ena vrstica za vsako državo, v tej vrstici pa bodo ime države in število medalj (zlatih, srebrnih, bronastih in vseh skupaj). Med stolpci naj bo po en presledek; imena držav naj bodo poravnana levo, medalje pa desno. Posamezni stolpec naj ne bo širši, kot je treba (glede na podatke v njem). Države naj bodo urejene padajoče po številu zlatih medalj; tiste z enakim številom zlatih naj bodo urejene padajoče po številu srebrnih; tiste, ki se ujemajo tako v številu zlatih kot v številu srebrnih, pa naj bodo urejene padajoče po številu bronastih. Primer takšnega izpisa:

```
ZDA        40 44 42 126
Kitajska   40 27 24  91
Japonska   20 12 13  45
Avstralija 18 19 16  53
Francija   16 26 22  64
Nizozemska 15  7 12  34
...
```

Predpostaviš lahko, da je različnih držav manj kot 200, da je vseh medalj skupaj manj kot 2000 in da so imena držav dolga kvečjemu 20 znakov. Tvoj program lahko bere s standardnega vhoda in piše na standardni izhod ali pa bere iz datoteke `vhod.txt` in piše v datoteko `izhod.txt` (karkoli ti je lažje).

3. Okrožnica

Ena od obveznosti dežurnega dijaka je, da po šoli raznese okrožnico. Prebrati jo mora vsakemu razredu, čas obiska posamičnega razreda in vrstni red obiskov pa si lahko izbere poljubno. Pri tem se hoče izogniti obiskovanju učilnic, v katerih poučujejo nekateri učitelji, saj ima z njimi slabe izkušnje.

Napiši program (ali podprogram oz. funkcijo), ki bo sprejel podatke o šolskem urniku za en dan (torej v kateri učilnici so posamični razredi in učitelji vsako šolsko uro) ter seznam učiteljev, ki se jih hočemo izogniti. Program oz. podprogram naj pripravi razpored obiskovanja razredov, s katerim se ne oglasimo pri nobenem neželenem učitelju, ali pa ugotovi, da takšen razpored ne obstaja. Število obiskov na šolsko uro ni omejeno. Posamezen razred smemo obiskati največ enkrat.

Predpostaviš lahko, da so učitelji, razredi, učilnice in šolske ure predstavljene z majhnimi naravnimi števili (recimo od 1 do 100), ne npr. z imeni ali kakšnimi drugimi bolj zapletenimi oznakami. Urnik je podan kot zaporedje četveric oblike (u, r, p, t), ki povedo, da učitelj u poučuje razred r v učilnici p na t-to šolsko uro dneva. Podrobnosti tega, v kakšni obliki tvoj (pod)program dobi vhodne podatke in vrne ali izpiše rezultate, si izberi sam in jih v svoji rešitvi tudi opiši.

4. Tridimenzionalni labirint

Rok se je naveličal reševanja labirintov, objavljenih v časopisih, in je presedlal na tridimenzionalne. Labirint je torej kvader, sestavljen iz $A \times B \times C$ enotskih kockic, vsaka pa je lahko bodisi prazna bodisi zid. Števila A, B in C so manjša ali enaka 100. Položaj posamezne kockice v labirintu lahko opišemo s trojico koordinat (x, y, z), kjer je $x \in \{0, \ldots, A-1\}$, $y \in \{0, \ldots, B-1\}$ in $z \in \{0, \ldots, C-1\}$. Po labirintu se lahko premikaš tako, da greš iz ene kockice v drugo, če sta obe prazni in imata skupno ploskev. Kockica je *izhod* iz labirinta, če s kakšno od svojih ploskev meji na zunanjost kvadra (labirinta).

Opiši postopek (ali napiši podprogram oz. funkcijo, če ti je lažje), ki dobi opis labirinta in začetne koordinate, vrne pa koordinate najbližjega in najbolj oddaljenega dosegljivega izhoda. (Če je možnih več rešitev, je vseeno, katero od njih vrne.) Pri tem se seveda oddaljenost do izhoda meri kot dolžina najkrajše poti od začetnega položaja do tistega izhoda; dolžina poti po labirintu pa je definirana kot število korakov na njej (število premikov iz ene kockice v sosednjo kockico). Če pišeš podprogram, si podrobnosti tega, v kakšni obliki tvoj podprogram dobi vhodne podatke in kako vrne ali izpiše rezultate, izberi sam in jih v svoji rešitvi tudi opiši.

Če ti je naloga pretežka, jo lahko za 13 točk od 20 rešiš z dodatno predpostavko, da v labirintu ni ciklov — z drugimi besedami, da je mogoče od ene prazne kockice do druge priti po kvečjemu eni poti, nikoli po dveh ali več različnih poteh.

5. Skladišče

Neko podjetje ima v skladišču n zabojev (zaboji so oštevilčeni od 0 do $n-1$). Zaboji so različno težki (nobena dva nista enako težka), vendar do njihovih tež ne moremo neposredno dostopati.

Posamezni zaboj je v posameznem trenutku v enem od dveh možnih stanj: *izbran* ali *neizbran*. Podjetje bi rado po vsaki spremembi stanja vedelo, kateri zaboj je zdaj srednji po teži med vsemi izbranimi zaboji. „Srednji po teži“ pomeni naslednje: če je izbranih $2k$ ali $2k-1$ zabojev, potem je srednji po teži tisti, ki je k-ti najlažji med njimi. (Na primer: če je izbranih 7 ali 8 zabojev, je srednji po teži četrti najlažji med njimi; če je izbranih 9 ali 10 zabojev, je srednji po teži peti najlažji med njimi; in tako naprej.)

Skladišče je razdeljeno na tri dvorane: A, B in C. V vsaki dvorani je dovolj prostora za vse zaboje. Na voljo imaš naslednje podprograme za premikanje zabojev med dvoranami in primerjanje zabojev po teži:

- PremaknilzAvB(z) — premakne zaboj s številko z iz dvorane A v dvorano B;
- PremaknilzAvC(z) — premakne zaboj s številko z iz dvorane A v dvorano C;
- PremaknilzB() — premakne iz dvorane B *najtežji* zaboj (izmed vseh, ki so takrat v dvorani B) v dvorano A in vrne številko tega zaboja;
- PremaknilzC() — premakne iz dvorane C *najlažji* zaboj (izmed vseh, ki so takrat v dvorani C) v dvorano A in vrne številko tega zaboja;
- JeLazjiOdB(z) — zaboj z mora biti iz dvorane A; funkcija vrne logično vrednost, ki pove, ali je zaboj z lažji od najtežjega zaboja dvorane B ali ne;
- JeLazjiOdC(z) — zaboj z mora biti iz dvorane A; funkcija vrne logično vrednost, ki pove, ali je zaboj z lažji od najlažjega zaboja dvorane C ali ne.

Teh podprogramov torej ne pišeš ti, ampak predpostavi, da že obstajajo in da jih lahko pokličeš iz svoje kode. Če podaš gornjim podprogramom kot parameter z neki zaboj, ki se takrat ne nahaja v dvorani A, se bo tvoj program sesul; enako tudi, če pokličeš PremaknilzB ali JeLazjiB takrat, ko je dvorana B prazna, ali pa če pokličeš PremaknilzC ali JeLazjiC takrat, ko je dvorana C prazna.

Še enkrat poudarimo, da z zaboji ne moreš delati drugače kot tako, da kličeš gornje podprograme.

Opiši, kako bi implementiral naslednja podprograma (oz. funkciji), ki bosta pomagala podjetju delati z izbranimi zaboji:

- Inicializacija(n) — sistem ga bo poklical na začetku izvajanja, da si lahko inicializiraš morebitne globalne spremenljivke in podobne stvari; kot parameter dobi število zabojev n.
- SpremembaStanja(z) — sistem ga bo poklical, ko je treba zaboju z spremeniti stanje (iz izbranega v neizbranega ali obratno). Funkcija naj vrne številko zaboja, ki je (po spremembi) srednji po teži med izbranimi zaboji. Če po spremembi ni izbran noben zaboj, naj funkcija vrne -1.

Poleg teh dveh podprogramov lahko tvoja rešitev deklarira in uporablja tudi svoje globalne spremenljivke in pomožne podprograme. Če ti je lažje, lahko namesto opisa postopka napišeš implementacijo v kakšnem konkretnem programskem jeziku (je pa s tem pri tej nalogi razmeroma dosti dela).

Predpostavi, da se na začetku izvajanja nahajajo vsi zaboji v dvorani A in da ni noben zaboj izbran. Operaciji JeLazjiOdB in JeLazjiOdC sta dragi, zato je pomembno, da ju uporabiš čim manjkrat.

Če ti je naloga pretežka, lahko za 13 točk od 20 rešiš lažjo različico, pri kateri se zabojem stanje spreminja le iz neizbranega v izbrano, nikoli pa obratno (ko je torej zaboj enkrat izbran, odtlej vedno ostane izbran).

21. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

23. januarja 2026

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Napredek

Ker so tekmovalci predstavljeni s številkami namesto z imeni, lahko številko tekmovalca uporabimo kot indeks v tabelo ali vektor. Najprej v zanki preberimo vrstni red s prvega tekmovanja in si v neki tabeli (v spodnji rešitvi je to mesto1) za vsakega tekmovalca zapišimo, na katerem mestu je bil (to je ravno števec zanke, s katero beremo tekmovalce). Nato v še eni zanki berimo vrstni red z drugega tekmovanja; pri vsakem prebranem tekmovalcu vemo, na katerem mestu je v tem drugem vrstnem redu, v prej omenjeni tabeli pa imamo podatek o tem, na katerem mestu je bil v prvem vrstnem redu. Tako ni težko izračunati razlike med obema uvrstitvama, to pa je napredek opazovanega tekmovalca. Imejmo še dve spremenljivki, ki povesta največji doslej najdeni napredek (maxNapredek) in številko tekmovalca, ki je ta napredek dosegel (maxKdo); pri vsakem novem tekmovalcu poglejmo, če je njegov napredek večji od največjega doslej, in če je, si ga zapomnimo. Na koncu izpišimo številko tekmovalca z največjim napredkom.

Oglejmo si implementacijo takšne rešitve v C++. Predpostavili bomo, da podatke dobimo na standardnem vhodu v treh vrsticah; v prvi je n, v preostalih dveh pa sta oba vrstna reda, pri čemer so števila ločena s presledkom. Tudi rezultat bomo izpisali na standardni izhod.

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

int main()
{
  // Preberimo število tekmovalcev.
  int n; cin >> n;

  // Preberimo prvi vrstni red in si za vsakega tekmovalca zapomnimo,
  vector<int> mesto1(n + 1);       // na katerem mestu je bil.
  for (int i = 0; i < n; ++i) {
    int tekmovalec; cin >> tekmovalec;
    mesto1[tekmovalec] = i; }

  // Preberimo drugi vrstni red, sproti računajmo napredek vsakega
  // tekmovalca in si zapomnimo najboljšega.
  int maxNapredek = -1, maxKdo = -1;
  for (int i = 0; i < n; ++i) {
    int tekmovalec; cin >> tekmovalec;

    // Za koliko mest je ta tekmovalec napredoval?
    int napredek = mesto1[tekmovalec] - i;

    // Če je to največji napredek doslej, si ga zapomnimo.
    if (maxKdo < 0 || napredek > maxNapredek)
      maxNapredek = napredek, maxKdo = tekmovalec; }

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

Opozorimo še na dve podrobnosti: za mesto1 smo alocirali $n+1$ elementov dolg vektor, da bomo lahko kot indekse uporabljali številke tekmovalcev, ki gredo od 1 do n namesto od 0 do $n-1$. (Lahko bi namesto tega seveda od številke tekmovalca odšteli 1, preden jo uporabimo kot indeks v vektor.) Pri računanju napredka pa pazimo na to, da nižja

številka mesta pomeni višjo uvrstitev; napredek je torej tem večji, čim bolj se je tekmovalcu številka mesta znižala. Zato pri izračunu napredka odštejemo mesto na drugem tekmovanju od mesta na prvem tekmovanju (in ne obratno).

Oglejmo si še implementacijo takšne rešitve v pythonu:

```
n = int(input()) # Preberimo število tekmovalcev.

# Preberimo prvi vrstni red in si za vsakega tekmovalca zapomnimo,
mesto1 = [-1] * (n + 1)       # na katerem mestu je bil.
vrstniRed = [int(s) for s in input().split()]
for i in range(n): mesto1[vrstniRed[i]] = i;

# Preberimo drugi vrstni red.
vrstniRed = [int(s) for s in input().split()]

# Za vsakega tekmovalca izračunajmo napredek in si zapomnimo najboljšega.
maxNapredek = -1; maxKdo = -1
for i in range(n):
  tekmovalec = vrstniRed[i]

  # Za koliko mest je ta tekmovalec napredoval?
  napredek = mesto1[tekmovalec] - i

  # Če je to največji napredek doslej, si ga zapomnimo.
  if maxKdo < 0 or napredek > maxNapredek:
    maxNapredek = napredek; maxKdo = tekmovalec

print(maxKdo) # Izpišimo rezultat.
```

2. Tabela medalj

Za vsako državo bomo vzdrževali zapis, ki bo vseboval ime države, število medalj vsake barve (zlate, srebrne, bronaste) in skupno število medalj. Vhodne podatke bomo brali v zanki; pri vsaki prebrani medalji poiščemo zapis za to državo (če ga še nimamo, ga zdaj dodamo) in v njej povečamo za 1 števec medalj tiste barve ter skupni števec medalj.[1]

Ko pridemo do konca vhodnih podatkov, uredimo zapise o državah tako, kot naloga zahteva za izpis tabele, torej padajoče po številu zlatih medalj, države z enakim številom zlatih uredimo padajoče po številu srebrnih in tako naprej. V spodnji rešitvi smo uporabili funkcijo sort iz C++ove standardne knjižnice, napisali pa smo svoj primerjalni operator.

Preden lahko začnemo tabelo zares izpisovati, moramo določiti širine stolpcev, kajti od tega je odvisno, koliko presledkov bo treba vriniti, da bo izpis lepo poravnan. Za prvi stolpec je treba pogledati, kako dolgo je najdaljše ime kakšne države; za ostale stolpce pa je dovolj, če poiščemo največje število medalj v vsakem stolpcu in preštejemo, koliko znakov je to število dolgo, če ga pretvorimo v niz.

Nato gremo lahko še enkrat v zanki po državah in sproti izpisujemo vrstice naše tabele. Pri tem pazimo na to, da bodo stolpci primerno široki in poravnani (imena levo, medalje desno). V spodnji rešitvi smo si pomagali z I/O manipulatorji iz C++ove standardne knjižnice (left in right za poravnavanje, setw za širino stolpca).

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

struct Drzava
{
  string ime;
```

[1] Ker naloga pravi, da je držav (in medalj) malo, bo dovolj dobro, če hranimo zapise o državah v tabeli ali vektorju in se pri vsaki prebrani medalji zapeljemo v zanki po zapisih, da najdemo pravega (za trenutno državo); če pa bi bilo držav lahko veliko, bi bilo zapise bolje hraniti v slovarju oz. razpršeni tabeli, kjer bi ime države uporabljali kot ključ (v C++ bi na primer uporabili unordered_map iz standardne knjižnice).

```
  int medalje[4]; // zlate, srebrne, bronaste, skupaj
  // Spodnji operator primerja zapise glede na vrstni red v izpisu: prej pridejo
  // države z več zlatimi medaljami; tiste z enakim številom zlatih se uredi
  // padajoče po srebrnih itd.
  bool operator < (const Drzava &D) const {
    for (int b = 0; b < 3; ++b)
      if (medalje[b] != D.medalje[b]) return medalje[b] > D.medalje[b];
    return false; }
};

int main()
{
  vector<Drzava> drzave;
  while (true)
  {
    // Preberimo naslednjo medaljo.
    string ime, barva; cin >> ime >> barva; if (! cin) break;
    // Poiščimo zapis za to državo.
    int i = 0; while (i < drzave.size() && drzave[i].ime != ime) ++i;
    // Če to državo vidimo prvič, zapis zanjo zdaj dodajmo.
    if (i == drzave.size()) drzave.push_back({ime, 0, 0, 0, 0});
    // Povečajmo števec medalj ustrezne barve in skupni števec vseh medalj.
    ++drzave[i].medalje[barva == "zlato" ? 0 : barva == "srebro" ? 1 : 2];
    ++drzave[i].medalje[3];
  }
  // Uredimo države po medaljah.
  sort(drzave.begin(), drzave.end());
  // Določimo širine stolpcev. Za začetek v sirine[0] pripravimo dolžino
  // najdaljšega imena, v sirine[1..4] pa največje število medalj vsake barve in skupaj.
  int sirine[5] = {0, 0, 0, 0};
  for (auto &D : drzave) {
    sirine[0] = max(sirine[0], (int) D.ime.length());
    for (int b = 1; b < 5; ++b) sirine[b] = max(sirine[b], D.medalje[b - 1]); }
  // Iz največjega števila medalj določimo širino stolpca.
  for (int b = 1; b < 5; ++b) sirine[b] = to_string(sirine[b]).length();
  // Izpišimo tabelo.
  for (auto &D : drzave) {
    cout << left << setw(sirine[0]) << D.ime << right;
    for (int b = 1; b < 5; ++b) cout << " " << setw(sirine[b]) << D.medalje[b - 1];
    cout << endl; }
  return 0;
}
```

Oglejmo si še primer rešitve v pythonu; tu bodo nekatere stvari malo lažje in krajše kot v C++. Med branjem vhodnih podatkov bomo zapise o državah hranili v slovarju namesto v seznamu, tako da bo zelo enostavno najti pravi zapis (oz. ugotoviti, da ga še nimamo). Ko pa z branjem vhodnih podatkov končamo in je treba države urediti, jih bomo predstavili s pari oblike (medalje, ime), pri čemer je medalje seznam, ki ima na prvem mestu število zlatih medalj, na drugem število srebrnih itd.; če seznam takšnih parov uredimo padajoče, bomo dobili točno tak vrstni red, kot ga potrebujemo za izpis v naši tabeli.

Za izpis stolpcev s primerno širino pride prav pythonov mehanizem za formatiranje nizov s predpono f. Če bi hoteli na primer spremenljivko x pretvoriti v niz širine 5 znakov, bi lahko uporabili `f"{x:5}"`; če pa širine vnaprej ne poznamo in jo imamo v spremenljivki y, lahko uporabimo `f"{x:{y}}"`. Če je x niz, bo poravnan levo, če je število, pa desno, kar je za potrebe naše naloge ravno prav, torej se nam s smerjo poravnavanja ni treba ukvarjati.

```
import sys
drzave = {}
```

```
for vrstica in sys.stdin:
    ime, barva = vrstica.strip().split()

    # Če to državo vidimo prvič, zapis zanjo zdaj dodajmo.
    if ime not in drzave: drzave[ime] = [0, 0, 0, 0]
    barva = 0 if barva == "zlato" else 1 if barva == "srebro" else 2

    # Povečajmo števec medalj ustrezne barve in skupni števec vseh medalj.
    drzave[ime][barva] += 1; drzave[ime][3] += 1

# Uredimo države po številu medalj.
drzave = [(medalje, ime) for (ime, medalje) in drzave.items()]
drzave.sort(reverse = True)

# Določimo širine stolpcev. Za začetek v sirine[0] pripravimo dolžino
# najdaljšega imena, v sirine[1..4] pa največje število medalj vsake barve in skupaj.
sirine = [0] * 5
for (medalje, ime) in drzave:
    sirine[0] = max(sirine[0], len(ime))
    for b in range(4): sirine[b + 1] = max(sirine[b + 1], medalje[b])

# Iz največjega števila medalj določimo širino stolpca.
for b in range(1, 5): sirine[b] = len(str(sirine[b]))

# Izpišimo tabelo.
for (medalje, ime) in drzave:
    sys.stdout.write(f"{ime:{sirine[0]}}")
    for b in range(1, 5): sys.stdout.write(f" {medalje[b - 1]:{sirine[b]}}")
    sys.stdout.write("\n")
```

3. Okrožnica

Ker smemo v isti uri obiskati poljubno mnogo razredov, je sestavljanje razporeda preprosto. V zanki pojdimo po zapisih, iz katerih je sestavljen urnik. Če v zapisu nastopa kakšen od učiteljev, ki se jim izogibamo, ta zapis preskočimo. Podobno, če v zapisu nastopa kakšen od razredov, ki smo jih v preteklosti že obvestili, tudi ta zapis preskočimo (saj istega razreda ne smemo obiskati večkrat). Sicer pa nas nič ne ovira, da ne bi tega razreda obvestili zdaj, torej dodajmo obisk tega razreda to uro v naš razpored. Na koncu moramo le še preveriti, če smo uspeli obiskati vse razrede (načeloma bi se lahko zgodilo, da bi bil neki razred v urniku vedno prisoten skupaj z enim od tistih učiteljev, ki se jim izogibamo; takšnega razreda ne moremo obvestiti, torej razpored, po kakršnem sprašuje naloga, ne obstaja).

Za lažje preverjanje, ali se nekemu učitelju izogibamo in ali smo neki razred že obvestili, bomo vzdrževali dve tabeli oz. vektorja logičnih vrednosti; naloga pravi, da so učitelji in razredi predstavljeni s števili od 1 do 100, zato lahko rezerviramo tabeli 101 elementov in uporabimo številko učitelja ali razreda kot indeks vanjo.

Oglejmo si primer implementacije takšne rešitve v C++. Spodnja funkcija PripraviRazpored dobi kot vhodne podatke dva vektorja, urnik in izogibajSe (slednji je seznam številk učiteljev, ki se jim izogibamo), razpored obiskov pa vrne v vektorju razpored. Funkcija PripraviRazpored vrne logično vrednost, ki pove, ali je ustrezen razpored uspela pripraviti ali ne.

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

struct Ura { int ucitelj, ucilnica, razred, ura; };
struct Obisk { int ucilnica, ura; };

bool PripraviRazpored(const vector<Ura> &urnik, const vector<int> &izogibajSe,
                      vector<Obisk> &razpored)
{
    // Pripravimo si vektor, kjer za vsakega učitelja piše, ali se mu izogibamo ali ne.
    vector<bool> izogibajSeB(101, false);
    for (int ucitelj : izogibajSe) izogibajSeB[ucitelj] = true;

    // Preglejmo urnik in pripravimo razpored obiskov.
    vector<bool> razredObvescen(101, false); razpored.clear();
```

```
  for (auto &U : urnik)
  {
    if (izogibajSeB[U.ucitelj]) continue;
    if (razredObvescen[U.razred]) continue;
    // Ta razred še ni bil obveščen, torej ga obvestimo zdaj.
    razredObvescen[U.razred] = true;
    razpored.push_back({U.ucilnica, U.ura});
  }
  // Poglejmo, ali smo uspeli obvestiti vse razrede.
  for (auto &U : urnik) if (! razredObvescen[U.razred]) return false;
  return true;
}
```

Oglejmo si še primer podobne rešitve v pythonu. Tokrat bomo namesto tabel logičnih vrednosti uporabili množice (pythonov tip set). Poleg množice obveščenih razredov bomo med pregledovanjem urnika pripravili tudi množico vseh razredov; na koncu lahko obe množici primerjamo in če sta enaki, vemo, da smo uspeli obvestiti vse razrede, sicer pa ne. Podprogram PripraviRazpored v spodnji rešitvi vrne razpored kot seznam zapisov tipa Obisk, če pa primernega razporeda ni, vrne None.

```
from collections import namedtuple
Ura = namedtuple("Ura", ["ucitelj", "ucilnica", "razred", "ura"])
Obisk = namedtuple("Obisk", ["ucilnica", "ura"])

def PripraviRazpored(urnik: list[Ura], izogibajSe: list[int]) -> list[Obisk]:
  # Seznam učiteljev, ki se jim izogibamo, predelajmo v množico.
  izogibajSe = set(izogibajSe)
  # Preglejmo urnik in pripravimo razpored obiskov.
  razpored = []; obvesceniRazredi = set(); vsiRazredi = set()
  for U in urnik:
    vsiRazredi.add(U.razred)
    if U.ucitelj in izogibajSe: continue
    if U.razred in obvesceniRazredi: continue
    # Tega razreda še nismo obvestili, torej ga obiščimo zdaj.
    razpored.append(Obisk(U.ucilnica, U.ura))
    obvesceniRazredi.add(U.razred)
  # Preverimo, ali smo obvestili vse razrede.
  return razpored if obvesceniRazredi == vsiRazredi else None
```

4. Tridimenzionalni labirint

Naloga je zelo primerna za reševanje z iskanjem v širino. Kockice, dosegljive iz začetne, bomo pregledovali po naraščajoči oddaljenosti. Pri tem bomo vzdrževali vrsto kockic, za katere že vemo, da so dosegljive, nismo pa še pregledali, kako je mogoče pot nadaljevati iz njih; postopek se začne s tem, da dodamo v vrsto le začetno kockico. Glavnino postopka tvori zanka, kjer v vsaki iteraciji vzamemo eno kockico z začetka vrste in dodamo na konec vrste tiste njene prazne sosede, ki jih doslej še nismo videli (in dodali v vrsto). (Potrebovali bomo torej tudi tabelo, v kateri bomo označevali, katere kockice smo že videli in dodali v vrsto; v spodnji rešitvi je to zeVidena.) Sosede kockice (x, y, z) so $(x \pm 1, y, z)$, $(x, y \pm 1, z)$ in $(x, y, z \pm 1)$.

Ko vzamemo kockico iz vrste, lahko tudi preverimo, če je tam izhod, torej če leži na eni od zunanjih ploskev kvadra; to je takrat, ko je ena od koordinat enaka 0 ali pa je $x = A - 1$ ali $y = B - 1$ ali $z = C - 1$. Ker pregledujemo kockice po naraščajoči oddaljenosti od začetne, bo prvi izhod, ki ga bomo našli, tudi najbližji, zadnji najdeni izhod pa bo najbolj oddaljen od začetne kockice. Tako si torej ne bo težko zapomniti najbližjega in najbolj oddaljenega izhoda.

Oglejmo si primer implementacije te rešitve v C++. Položaj kockice lahko namesto s trojico koordinat (x, y, z) predstavimo tudi z enim samim številom $x \cdot BC + y \cdot C + z$, kar je prikladno, ker ga lahko potem uporabimo tudi kot indeks v tabelo ali vektor. Spodnji podprogram uporablja to pri vektorjih jePrazna (ki ga dobi kot parameter; v njem za

vsako kockico piše, ali je prazna ali ne) in zeVidena (za označevanje, katere kockice smo že dodali v vrsto). Naš podprogram vrne logično vrednost, ki pove, ali je iz začetnega položaja sploh dosegljiv kak izhod.

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

struct Tocka { int x, y, z; };

bool Labirint(int A, int B, int C,
              // jePrazna[x * B * C + y * C + z] pove, ali je kockica (x, y, z) prazna
              vector<bool> &jePrazna,
              Tocka zacetek, Tocka &najblizjiIzhod, Tocka &najboljOddaljenIzhod)
{
  // zeVidena hrani podatke o tem, katere kockice smo med iskanjem v širino že videli.
  vector<bool> zeVidena(A * B * C, false);

  // Iskanje v širino bomo začeli pri kockici „zacetek“.
  int u0 = (zacetek.x * B + zacetek.y) * C + zacetek.z;
  zeVidena[u0] = true; queue<int> vrsta; vrsta.emplace(u0);
  bool naseIlzhod = false;

  // Preiščimo vse prazne kockice, dosegljive iz začetnega položaja.
  while (! vrsta.empty())
  {
    int u = vrsta.front(); vrsta.pop();
    int x = u / (B * C), y = (u / C) % B, z = u % C;

    // Kockica (x, y, z) je prazna in dosegljiva. Ali je tudi izhod?
    if (x == 0 || y == 0 || z == 0 || x == A - 1 || y == B - 1 || z == C - 1) {

      // Zadnji najdeni izhod bo najbolj oddaljen.
      najboljOddaljenIzhod = Tocka{x, y, z};

      // Prvi najdeni izhod je tudi najbližji.
      if (! naseIlzhod) { naseIlzhod = true; najblizjiIzhod = Tocka{x, y, z}; } }

    // Preglejmo sosede trenutne kockice. Po eni od koordinat se lahko premaknemo za ±1.
    for (int dim = 0; dim < 3; ++dim) for (int d = 0; d < 2; ++d)
    {
      int xx = x, yy = y, zz = z; (dim == 0 ? xx : dim == 1 ? yy : zz) += 2 * d - 1;

      // Kockica (xx, yy, zz) je soseda kockice (x, y, z).
      // Ali smo morda padli ven iz kvadra?
      if (xx < 0 || xx >= A || yy < 0 || yy >= B || zz < 0 || zz >= C) continue;

      // Če smo to sosednjo kockico že videli ali pa je zazidana, jo preskočimo.
      int v = (xx * B + yy) * C + zz;
      if (! jePrazna[v] || zeVidena[v]) continue;

      // Sicer jo dodajmo v vrsto, da bomo kasneje nadaljevali pot iz nje.
      zeVidena[v] = true; vrsta.emplace(v);
    }
  }
  return naseIlzhod;
}
```

Oglejmo si še implementacijo takšne rešitve v pythonu. Koordinate kockic lahko predstavimo s pythonovimi n-tericami (tuple); v taki obliki pričakuje naša spodnja funkcija začetni položaj (zacetek), kot rezultat pa vrne funkcija par takšnih n-teric, eno za najbližji izhod in eno za najbolj oddaljenega; če iz začetnega položaja sploh ni dosegljiv noben izhod, funkcija vrne (None, None). Za označevanje tega, katere kockice smo že videli (in dodali v vrsto), smo za spremembo namesto tabele uporabili množico (set v pythonu).[2]

[2]Podobno bi lahko naredili tudi v gornji C++ovski rešitvi, npr. z razredom unordered_set iz standardne knjižnice. Ali je boljša tabela ali množica, je v splošnem težko reči, saj je to odvisno od tega, kolikšen delež kvadra bomo morali preiskati (več kockic ko obiščemo, manj je koristi od tega, da smo namesto tabele uporabili množico).

```
import collections

# jePrazna[x * B * C + y * C + z] pove, ali je kockica (x, y, z) prazna.
def Labirint(A: int, B: int, C: int, jePrazna: list[int], zacetek: tuple[int, int, int]):
    # Iskanje v širino bomo začeli pri kockici „zacetek“.
    u0 = (zacetek[0] * B + zacetek[1]) * C + zacetek[2]
    zeVidena = set([u0]); vrsta = collections.deque([u0])
    najblizjiIzhod = None; najboljOddaljenIzhod = None

    # Preiščimo vse prazne kockice, dosegljive iz začetnega položaja.
    while vrsta:
        u = vrsta.popleft(); x = u // (B * C); y = (u // C) % B; z = u % C

        # Kockica (x, y, z) je prazna in dosegljiva. Ali je tudi izhod?
        if x == 0 or y == 0 or z == 0 or x == A - 1 or y == A - 1 or z == A - 1:

            # Zadnji najdeni izhod bo najbolj oddaljen.
            najboljOddaljenIzhod = (x, y, z)

            # Prvi najdeni izhod je tudi najbližji.
            if najblizjiIzhod is None: najblizjiIzhod = (x, y, z)

        # Preglejmo sosede trenutne kockice. Po eni od koordinat se lahko premaknemo za ± 1.
        for soseda in range(6):
            v = [x, y, z]; v[soseda // 2] += 2 * (soseda % 2) - 1; (xx, yy, zz) = v

            # Kockica (xx, yy, zz) je soseda kockice (x, y, z).
            # Ali smo morda padli ven iz labirinta?
            if not (0 <= xx < A and 0 <= yy < B and 0 <= zz < C): continue;

            # Če smo to sosednjo kockico že videli ali pa je zazidana, jo preskočimo.
            v = (xx * B + yy) * C + zz
            if v in zeVidena or not jePrazna[v]: continue

            # Sicer jo dodajmo v vrsto, da bomo kasneje nadaljevali pot iz nje.
            zeVidena.add(v); vrsta.append(v)
    return (najblizjiIzhod, najboljOddaljenIzhod)
```

5. Skladišče

Razmislimo, kaj se dogaja s srednjim zabojem po teži, ko se množica izbranih zabojev spreminja. Recimo, da je bilo prej izbranih $2k$ zabojev in srednji je bil k-ti najlažji med njimi; in recimo, da se jim pridruži še en nov zaboj. Zdaj imamo $2k + 1$ izbranih zabojev, torej bo srednji tisti, ki je zdaj $(k + 1)$-vi najlažji med njimi. To je morda isti zaboj kot prej, če se je namreč novi zaboj vrinil pred njega v vrstnem redu po teži (torej če je bil novi zaboj lažji od dosedanjega srednjega); če pa je novi zaboj težji od starega srednjega, se srednji zaboj spremeni — novi srednji je tisti, ki stoji tik za starim srednjim v vrstnem redu po teži.

Podobno bi lahko razmišljali tudi v primerih, ko je bilo prej izbranih liho mnogo zabojev in se jim pridruži še eden; in spet podobno tudi v primerih, ko neki zaboj izpade iz množice izbranih zabojev. Če si predstavljamo izbrane zaboje urejene po teži, se to, kateri zaboj je srednji, vedno spreminja le po malem, za en zaboj naprej ali nazaj po tem vrstnem redu.

V gornjem razmisleku smo tudi videli, da je koristno vedeti, ali je novi zaboj (ki ga dodajamo med izbrane) lažji ali težji od (dosedanjega) srednjega po teži. Ker nam podprogrami, ki nam jih daje naloga na razpolago za delo z zaboji, omogočajo primerjati zaboje po teži le tako, da primerjamo neki zaboj iz dvorane A z najtežjim zabojem dvorane B ali z najlažjim zabojem dvorane C, bo torej koristno, če poskrbimo, da bo eden od slednjih dveh ravno srednji zaboj po teži (med izbranimi).

Na misel nam torej lahko pride, da bi hranili neizbrane zaboje v dvorani A, izbrane zaboje pa v dvoranah B in C, in sicer naj B vsebuje srednji zaboj po teži in vse tiste, ki so lažji od njega, dvorana C pa naj vsebuje vse (izbrane) zaboje, ki so težji od srednjega. Ko se neki zaboj spremeni iz neizbranega v izbranega, ga lahko primerjamo s srednjim zabojem po teži (s funkcijo JeLazjiB) in se na podlagi tega odločimo, ali ga bomo poslali v dvorano B ali C. Po tej spremembi se lahko izkaže, da je v eni ali drugi dvorani preveč zabojev: če imamo izbranih m zabojev, jih mora biti $\lceil m/2 \rceil$ v dvorani B ter $\lfloor m/2 \rfloor$

v dvorani C.[3] Če jih je v dvorani B preveč, moramo premakniti najtežji zaboj iz B v C (kjer postane najlažji zaboj), če pa je preveč zabojev v C, moramo najlažjega iz C premakniti v B (kjer to postane najtežji zaboj).[4]

Z dosedanjim razmislekom smo že rešili lažjo različico naloge, pri kateri se zabojem stanje spreminja le iz neizbranega v izbrano. Malo več dela pa je s spremembami v obratni smeri. Ko se neki izbrani zaboj spremeni v neizbranega, bi naša dosedanja rešitev načeloma zahtevala, da ga premaknemo iz dvorane B oz. C (v katerikoli od njiju se je pač tedaj nahajal) v dvorano A (potem pa lahko spet primerno „uravnotežimo“ dvorani B in C, torej po potrebi premaknemo kak zaboj iz tiste, v kateri jih je zdaj preveč, v tisto, kjer jih je zdaj premalo). Toda spomnimo se, da iz dvoran B ali C ne moremo premakniti poljubnega zaboja, pač pa iz B le najtežjega, iz C pa le najlažjega. Če naš zaboj, ki se je pravkar spremenil iz izbranega v neizbranega, ni tak, ga bomo morali torej do nadaljnjega pustiti v dvorani, kjer se nahaja; za vsako od dvoran B ali C bomo morali ločeno šteti, koliko je v njej izbranih in koliko neizbranih zabojev; za vsak zaboj bomo morali v neki tabeli vzdrževati podatek o tem, ali je izbran ali ne; in ko se potem želimo ukvarjati z najtežjim zabojem v B (in podobno pri najlažjem zaboju v C), moramo najprej preveriti, če je ta zaboj sploh še izbran; če ni, je to eden od tistih, ki bi jih morali že prej premakniti iz B v A, pa tega takrat nismo mogli narediti in ga moramo premakniti zdaj (v spodnji rešitvi za to skrbi podprogram PospraviNeizbrane, ki premika zaboje iz B v A, dokler ni najtežji v B eden od izbranih zabojev, ter iz C v A, dokler ni najlažji v C eden od izbranih zabojev).

Čeprav zahteva naloga le opis postopka, si za primer vseeno oglejmo tudi implementacijo te rešitve v C++:

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

enum Dvorana { A = 0, B = 1, C = 2 };
// Naslednja vektorja za vsak zaboj povesta, v kateri dvorani se nahaja in ali je izbran.
vector<Dvorana> kje; vector<bool> izbran;
int iB, iC; // število izbranih zabojev v B oz. C
int nB, nC; // število neizbranih zabojev v B oz. C

void Inicializacija(int n)
{
  kje.clear(); kje.resize(n, A);
  izbran.clear(); izbran.resize(n, false);
  nB = 0; nC = 0; iB = 0; iC = 0;
}

void PospraviNeizbrane()
{
  // Premikajmo zaboje iz B, dokler je najtežji zaboj B-ja neizbran.
  while (nB > 0)
    if (int z = PrekmakniIzB(); ! izbran[z]) --nB, kje[z] = A;
    else { PrekmakniIzAvB(z); break; }
  // Premikajmo zaboje iz C, dokler je najlažji zaboj C-ja neizbran.
  while (nC > 0)
    if (int z = PrekmakniIzC(); ! izbran[z]) --nC, kje[z] = A;
    else { PrekmakniIzAvC(z); break; }
}

int SpremeniStanje(int z)
{
  izbran[z] = ! izbran[z];
  if (izbran[z]) {
    // Če je z že v eni od dvoran B in C, je treba le popraviti števce.
    if (kje[z] == B) { ++iB; --nB; }
```

[3] Zapis $\lceil \cdot \rceil$ pomeni, da rezultat deljenja zaokrožimo navzgor, $\lfloor \cdot \rfloor$ pa, da ga zaokrožimo navzdol.

[4] Več kot enega zaboja na ta način ne bo treba premakniti; o tem se lahko prepričamo, če ločeno obravnavamo štiri primere glede na parnost števila izbranih zabojev in glede na to, ali je novi izbrani zaboj prišel v B ali v C. Podrobnosti prepuščamo bralcu za vajo.

```
      else if (kje[z] == C) { ++iC; --nC; }
      // Sicer ga moramo premakniti iz A v eno od dvoran B in C.
      else if (iB + nB > 0 && JeLazjiOdB(z)) { PremakniIzAvB(z); ++iB; kje[z] = B; }
      else { PremakniIzAvC(z); ++iC; kje[z] = C; } }
    else {
      // Če je z postal neizbran, ostane zaenkrat v svoji dvorani (B ali C),
      // mi pa le popravimo števca izbranih in neizbranih zabojev te dvorane.
      if (kje[z] == B) { --iB; ++nB; }
      else { --iC; ++nC; } }

    // Uravnotežimo dvorani B in C.
    int stIzbranih = iB + iC;
    int ciljB = (stIzbranih + 1) / 2; // Toliko izbranih bi moralo biti v B.
    if (iB > ciljB) {
      PospraviNeizbrane();
      int z = PremakniIzB(); --iB;
      PremakniIzAvC(z); kje[z] = C; ++iC;}
    else if (iB < ciljB) {
      PospraviNeizbrane();
      int z = PremakniIzC(); --iC;
      PremakniIzAvB(z); kje[z] = B; ++iB; }
    // Najtežji zaboj v B je zdaj srednji izbrani po teži.
    if (iB == 0) return -1;
    PospraviNeizbrane();
    int srednji = PremakniIzB(); PremakniIzAvB(srednji); return srednji;
}
```

21. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

23. januarja 2026

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. Napredek

- Za vse točke pričakujemo rešitev s časovno zahtevnostjo $O(n)$. Za drobne neučinkovitosti znotraj te časovne zahtevnosti naj se ne odšteva točk (npr. v naši pythonovski rešitvi bi lahko mesto1 bil slovar namesto seznama).

- Rešitve s časovno zahtevnostjo $O(n \log n)$ naj dobijo največ 15 točk, tiste s časovno zahtevnostjo $O(n^2)$ pa največ 10 točk, če so sicer pravilne.
- Glede rezultatov je dovolj, če program nekako vrne ali izpiše, kateri tekmovalec je najbolj napredoval; ni treba izpisati, za koliko mest je napredoval.
- Če bi kakšna rešitev pomotoma računala, kateri je najbolj nazadoval namesto najbolj napredoval, naj se ji zaradi tega odšteje dve točki.
- Ni se težko prepričati, da gotovo obstaja vsaj en tekmovalec, ki je na drugi tekmi uvrščen vsaj tako visoko kot na prvi. Rešitev se sme opirati na to dejstvo (npr. pri inicializaciji kakšne spremenljivke, preden začne iskati tekmovalca z največjim napredkom), ne da bi ga posebej utemeljila.

2. Tabela medalj

- Pri tej nalogi poudarek ni na učinkovitosti rešitve, saj besedilo posebej pravi, da je držav in medalj razmeroma malo. Če je v podatkih omenjenih d različnih držav, sme rešitev (za vse točke) porabiti $O(d)$ časa za vsako prebrano medaljo (npr. ker hrani zapise o državah v seznamu, ki ga mora preiskati po vrsti, da najde zapis za pravo državo; primer tega je naša rešitev v C++) in $O(d^2)$ časa za urejanje držav v pravi vrstni red (npr. morda si bo kak tekmovalec napisal svojo improvizirano različico urejanja z izbiranjem ali kaj podobnega).
- Naši dve rešitvi za določanje širine stolpcev z medaljami najprej poiščeta največjo vrednost v vsakem stolpcu, nato pa izračunata, koliko znakov dolg niz nastane iz te vrednosti. Za enako dobro naj šteje tudi rešitev, ki pretvori vsako vrednost v niz in si zapomni dolžino najdaljšega izmed tako dobljenih nizov.
- Če se več držav ujema v številu medalj vseh treh barv, je vseeno, v kakšnem vrstnem redu so urejene (ni treba, da so npr. po imenu ali kaj podobnega). Če pa bi kakšna rešitev uredila države le padajoče po številu zlatih medalj, pozabila pa bi, da je treba tiste z enakim številom zlatih urediti padajoče po številu srebrnih (in če je treba, tudi po bronastih), naj se ji zaradi tega odšteje tri točke.
- Besedilo naloge pravi, da stolpci ne smejo biti širši, kot je treba. Če se kakšna rešitev ne bi trudila izračunati širine stolpcev, ampak bi zanje postavila neko vnaprej določeno dovolj veliko širino (npr. iz besedila naloge vidimo, da je za imena držav dovolj 20 znakov, za število medalj pa največ 4 znaki), naj se ji zaradi tega odšteje 5 točk.
- Rešitvi, ki med stolpci pozabi izpisati presledek, naj se zaradi tega odšteje dve točki.
- Naši rešitvi sta si za poravnavanje stolpcev pri izpisu pomagali z I/O-manipulatorji v C++ in s formatted string literali v pythonu. Za enako dobro naj šteje tudi, če bi rešitev sama poskrbela za izpis primernega števila presledkov na primernem mestu (in jih npr. izpisovala v zanki).

3. Okrožnica

- Besedilo naloge pravi, da je učiteljev, razredov itd. malo, zato pri tej nalogi poudarek ni na učinkovitosti rešitve. Naši dve rešitvi na primer predelata seznam učiteljev, ki se jim izogibamo, v tabelo logičnih vrednosti ali pa v množico, kar nam omogoča, da v $O(1)$ časa preverimo, ali se nekemu učitelju izogibamo ali ne. Če bi kakšna rešitev namesto tega šla vsakič znova (pri vsakem zapisu na urniku) v zanki po seznamu in preverjala, ali je učitelj s trenutnega zapisa prisoten v seznamu, naj se ji zaradi te neučinkovitosti odšteje 1 točko. Ravno tako naj se odšteje 1 točko rešitvi, ki bi predpostavila, da kot vhodni podatek že dobi množico učiteljev, ki se jim je treba izogibati, in ne seznama oz. zaporedja (kot pravi naloga).

- Če bi kakšna rešitev za preverjanje tega, ali je neki razred že obvestila ali ne, porabila po več kot $O(1)$ časa, naj se ji zaradi tega odšteje 1 točko. (Primer bi bila npr. rešitev, ki si pripravlja seznam že obiskanih razredov namesto tabele logičnih vrednosti ali množice; ali pa rešitev, ki se vsakič z zanko zapelje po doslej sestavljenem razporedu obiskov).
- Naloga ne zahteva, naj bo razpored obiskov, ki ga vrne tekmovalčeva rešitev, v kakšnem posebnem vrstnem redu (npr. urejen po času).
- Rešitev sme predpostaviti, da so zapisi v urniku, ki ga dobi kot vhodni podatek, na neki način urejeni (npr. po času), čeprav od tega sicer ni nobene posebne koristi.
- Naši dve rešitvi kot razpored obiskov pripravita seznam parov oblike $\langle$*učilnica*, *ura*$\rangle$, za enako dobro pa naj velja tudi rešitev, ki vrne seznam parov oblike $\langle$*razred*, *ura*$\rangle$ ali pa kar seznam primernih zapisov iz urnika, ki ga je dobila kot vhodni podatek.
- Naloga posebej poudarja, da smemo obiskati posamezni razred največ enkrat. Rešitvam, ki obiščejo kak razred po večkrat, naj se zaradi tega odšteje štiri točke.
- Rešitvam, ki obiščejo kakšnega od učiteljev, ki bi se jim morale izogibati, naj se zaradi tega odšteje štiri točke.
- Rešitvam, ki ne preverijo, ali so uspele obiskati vse razrede (in zato morda vrnejo neki neveljaven razpored, ki obišče nekaj razredov, ne pa vseh), naj se zaradi tega odšteje tri točke.

4. Tridimenzionalni labirint

- V našem primeru rešitve smo labirint predstavili z enodimenzionalno tabelo logičnih vrednosti; enako dobro je seveda lahko tudi kaj drugega, npr. tridimenzionalna tabela ali pa celo množica praznih kockic ipd. Lahko bi tudi uporabili tabelo velikosti $100 \times 100 \times 100$, kar je pri tej nalogi največji možni labirint, in bi bila pač pri manjših labirintih delno neizkoriščena.
- Podobno je tudi vseeno, ali rešitev za označevanje že obiskanih kockic uporablja tabelo (kot naša C++ovska rešitev) ali množico celih števil (kot naša pythonovska rešitev) ali celó množico trojic (x, y, z). Če pa bi rešitev uporabila tu kakšno posebej neučinkovito podatkovno strukturo, pri kateri bi preverjanje, ali je bila kockica že obiskana, vzelo več kot $O(1)$ časa, naj se ji zaradi tega odšteje dve točki.
- Namesto iskanja v širino si lahko si predstavljamo rešitev, ki z rekurzijo pregleduje vse možne poti iz začetne kockice. Težava je, da je (če so v labirintu cikli) takšnih poti lahko eksponentno mnogo, zato je lahko ta rešitev zelo neučinkovita. Takšne rešitve naj dobijo največ 15 točk, če so drugače pravilne.
- Če pa bi takšna rešitev morda pozabila preveriti, ali je v neki kockici že bila na nekem zgodnejšem koraku poti (torej v nekem nadrejenem rekurzivnem klicu), in bi se zaradi tega na nekaterih labirintih lahko zaciklala, naj dobi največ 13 točk. (To je namreč potem pravzaprav rešitev lažje različice naloge, omenjene na koncu besedila naloge.)

5. Skladišče

- Ta naloga je za šolsko tekmovanje razmeroma težka in ne moremo pričakovati, da bodo tekmovalci pravilno poskrbeli za prav vse podrobnosti, sploh ker nekatere od njih verjetno opazimo šele, če rešitev implementiramo in preizkusimo na računalniku. Zanimata nas predvsem (1) ideja, da v dvorani B hranimo zaboje, lažje od srednjega, v dvorani C pa zaboje, težje od srednjega (srednji zaboj lahko

potem načeloma hranimo v katerikoli od njiju; naša rešitev ga hrani v B, lahko pa bi ga tudi v C, če bi temu primerno prilagodili rešitev), in da zaboje po potrebi primerno prerazporejamo med dvoranama; in (2) ideja, da zaboje, ki so postali neizbrani, „leno“ premikamo iz dvoran B in C nazaj v A, torej šele takrat, ko tak zaboj postane najtežji v B ali najlažji v C. Lažja različica naloge (za 13 točk od 20), ki jo omenja zadnji odstavek besedila, zahteva le idejo (1).

- Naloga pravi, naj operaciji JeLazjiOdB in JeLazjiOdC izvedemo čim manjkrat. Za vse točke pričakujemo, da rešitev izvede v povprečju $O(1)$ teh operacij na vsako spremembo stanja zaboja; z drugimi besedami, povprečno število teh operacij na vsako spremembo stanja zaboja mora biti navzgor omejeno z neko konstanto, neodvisno od n. (Naša uradna rešitev izvede največ eno tako operacijo pri spremembi neizbranega zaboja v izbranega, pri obratni spremembi pa sploh nobene.) Rešitve, ki izvedejo več kot $O(1)$ takšnih operacij, naj dobijo največ 8 točk, če so drugače pravilne. Primer takšne rešitve dobimo, če opazimo, da če imamo vse zaboje ves čas v dvorani A, lahko dva zaboja, recimo u in v, primerjamo po teži takole: `PremakniIzAvB(v); bool b = JeLazjiOdB(u); PremakniIzB();`. Po teh treh stavkih nam `b` pove, ali je zaboj u lažji od zaboja v. Ko znamo tako primerjati poljubna dva zaboja po teži, lahko vzdržujemo npr. urejen seznam izbranih zabojev, vanj na primerno mesto vrivamo zaboje, ko postanejo izbrani, ali jih iz njega brišemo, ko prenehajo biti izbrani, in zlahka vidimo, kateri je srednji po teži. Toda takšna rešitev uporabi po vsaki spremembi stanja $O(n)$ primerjav (klicev JeLazjiOdB); to lahko zmanjšamo na $O(\log n)$, če pri ugotavljanju, kam v seznamu je treba vriniti nov zaboj, uporabimo bisekcijo, ali pa če izbrane zaboje namesto v urejenem seznamu hranimo v kakšni drevesasti podatkovni strukturi, v vsakem primeru pa je to več od $O(1)$.

- Primer podrobnosti, ki jih človek pri tej nalogi zlahka spregleda in ki morda v rešitvi niti ne pridejo do izraza, če pišemo le opis postopka in ne implementacije:
  - Morda pustimo pri uravnoteževanju dvoran B in C (da je v vsaki od njiju pravo število izbranih zabojev) v kakšni od njiju en izbran zaboj preveč ali premalo.
  - Ko pride nov izbran zaboj, ga morda primerjamo z najtežjim iz B, ne da bi se prej prepričali, da B ni prazna.
  - Morda vrnemo najtežji zaboj iz B v prepričanju, da je to srednji po teži med izbranimi zaboji, pa se prej nismo prepričali, ali je ta zaboj sploh izbran (lahko je to eden od zabojev, ki niso več izbrani, pa jih še nismo utegnili premakniti iz B nazaj v A).
  - Ko neki zaboj postane izbran, ga morda vedno poskušamo premakniti iz A v B ali C, pri tem pa pozabimo na možnost, da je ta zaboj lahko že v B ali C (ker je bil nekoč prej izbran, potem pa je postal neizbran in ga odtlej še nismo uspeli premakniti v A).

  Za takšne drobne napake se morda lahko odšteva po eno točko, ni pa mišljeno, da bi se zaradi tega bistveno spremenilo število točk, ki jih je rešitev sicer dobila (če jih je) zaradi pravilne ideje.

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. Napredek	lažja do srednja naloga v prvi skupini
2. Tabela medalj	srednja do težja naloga v prvi ali lažja v drugi skupini
3. Okrožnica	težja naloga v prvi ali lažja v drugi skupini
4. Labirint	srednje težka naloga v drugi ali lažja v tretji skupini
5. Skladišče	težka naloga v drugi ali srednja 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.