20. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

24. januarja 2025

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.");
   }
}
```

20. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

24. januarja 2025

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. Koščki

Pred seboj imamo razmetane koščke sestavljanke, ki je bila s pravokotno mrežo razdeljena na koščke. Prešteli smo število vogalnih, robnih in notranjih koščkov: v, r, n. **Napiši program** (ali podprogram oz. funkcijo, če ti je lažje), ki bo iz teh treh števil izračunal dimenzije sestavljanke: širino in višino (oboje vsaj 2 in kvečjemu 100 000). Tvoj program naj deluje pravilno tudi v primerih, ko eden od koščkov sestavljanke manjka (torej ko je eno od števil v, r in n za ena manjše, kot bi bilo, če bi imeli vse koščke).

Podrobnosti glede tega, kako tvoj (pod)program dobi vhodne podatke in kako vrne ali izpiše rezultate, si izberi sam. Če je možnih več rešitev, je vseeno, katero od njih izpiše. Predpostaviš lahko, da so vhodni podatki taki, da vsaj ena rešitev gotovo obstaja.

Primer: če dobimo $v = 4$ vogalne koščke, $r = 14$ robnih in $n = 11$ notranjih, lahko zaključimo, da imamo opravka z mrežo 6×5 (pri čemer en notranji košček manjka, drugače bi jih bilo namreč 12 in ne le 11).

Lažja različica: za 15 točk od 20 lahko rešiš nalogo v poenostavljeni različici, pri kateri noben košček sestavljanke ne manjka.

2. Tvorjenje besed

Otroci se igrajo igro tvorjenja besed. Za začetek igre določijo besedo, nato pa tvorijo nove besede ob upoštevanju enega od spodnjih pravil:

- novi besedi se odvzame/briše eno črko na katerem koli mestu,
- novi besedi se doda eno črko na katerem koli mestu (na začetku besede ali na koncu besede in v sredini besede),
- v besedi se eno črko spremeni (s tem je mišljeno, da spremenimo eno pojavitev ene črke, ne pa, da bi poiskali in zamenjali vse pojavitve neke črke z neko drugo črko).

Napiši program (ali podprogram oz. funkcijo), ki za dani dve besedi preveri, ali je drugo mogoče dobiti iz prve (v enem koraku) na zgoraj opisani način. Podrobnosti tega, v kakšni obliki tvoj (pod)program dobi vhodne podatke in vrne ali izpiše rezultat, si izberi sam.

3. Simetrična matrika

Na vhodu dobimo karirasto mrežo oz. tabelo $n \times n$ celic, ki ima v nekaterih celicah vpisana naravna števila, druge celice pa so prazne. Zanima nas, ali lahko prazne celice zapolnimo z naravnimi števili tako, da bo mreža simetrična. (Mreža je simetrična natanko tedaj, če za vsaka i in j z območja od 1 do n velja, da je število v celici na preseku i-te vrstice in j-tega stolpca enako kot število v celici na preseku j-te vristce in i-tega stolpca.)

Napiši program, ki kot vhod dobi takšno delno izpolnjeno mrežo in ugotovi, ali jo je mogoče dopolniti tako, da bo simetrična. Če je to mogoče, naj dopolnjeno mrežo tudi izpiše (če je možnih več rešitev, je vseeno, katero od njih izpiše), sicer pa naj izpiše „`NE`“. Tvoj (pod)program lahko vhodne podatke prebere s standardnega vhoda in izpiše rezultate na standardni izhod, lahko pa bere iz datoteke `vhod.txt` in izpiše rezultate v datoteko `izhod.txt` (karkoli ti je lažje). V prvi vrstici vhoda bo število n (vsaj 1 in kvečjemu 100), sledi pa n vrstic, ki vsebujejo vsaka po n števil, ki predstavljajo vsebino vhodne tabele; prazne celice so predstavljene s številom -1.

Primer vhoda:	Eden od možnih pripadajočih izhodov:	Še en primer vhoda:	Pripadajoči izhod:

`4`
`1 2 -1 4`
`-1 -1 -1 7`
`3 -1 2 9`
`4 -1 9 -1`
	`1 2 3 4`
	`2 3 4 7`
	`3 4 2 9`
	`4 7 9 1`
		`4`
		`1 2 -1 4`
		`-1 -1 -1 7`
		`3 -1 2 9`
		`4 -1 8 -1`
			`NE`

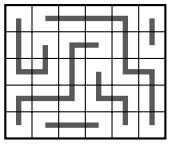
Naslednja slika kaže vhodni mreži iz gornjih dveh primerov; levo je mogoče dopolniti v simetrično, desne pa ne.

1	2		4
			7
3		2	9
4		9	

1	2		4
			7
3		2	9
4		8	

4. Prosti tok

Podano imamo karirasto mrežo $w \times h$ polj (w stolpcev, h vrstic; velja $w \leq 100$ in $h \leq 100$) in k poti v tej mreži. Vsaka pot je sestavljena le iz navpičnih in vodoravnih odsekov in je opisana kot zaporedje koordinat, pri čemer pa so poleg začetka in konca poti navedene samo tiste koordinate, kjer se spremeni smer (iz vodoravne v navpično ali obratno). **Napiši program** (ali podprogram oz. funkcijo), ki kot vhodne podatke dobi velikost mreže ter opise poti in preveri, ali dane poti pokrijejo celotno mrežo in se pri tem tudi nikoli ne sekajo (niti med seboj niti same sebe) — z drugimi besedami: ali vsako celico mreže obišče natanko ena pot (in to natanko enkrat). Podrobnosti tega, v kakšni obliki tvoj (pod)program dobi vhodne podatke, si izberi sam in jih v svoji rešitvi tudi opiši.



Primer: gornja slika kaže mrežo velikosti 6×5, na kateri je $k = 6$ poti (debele sive črte), ki pokrijejo celotno mrežo in se pri tem nikoli ne sekajo.

5. Standardna Youngova tabela

Youngov diagram je diagram z n *škatlicami*, razporejenimi v vrstice in stolpce tako, da ima vsak stolpec manj ali enako število škatlic kot stolpec levo od njega in da ima vsaka vrstica manj ali enako število škatlic kot vrstica nad njo. Če diagram napolnimo z vrednostmi od 1 do n (pri čemer se vsaka od njih pojavlja v natanko eni škatlici), dobimo *Youngovo tabelo*. Če velja, da števila v vsakem stolpcu naraščajo od zgoraj navzdol, v vsaki vrstici pa naraščajo od leve proti desni, govorimo o *standardni Youngovi tabeli*.

Na vhodu je podana Youngova tabela. **Opiši postopek** (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki ugotovi, ali lahko dano tabelo z zamenjavo največ enega para števil spremenimo v standardno Youngovo tabelo.

1	5	7	10
2	6	8	
3	9		
4	12		
11			

1	5	7	10
2	9	8	
3	6		
4	12		
11			

Primer: gornja slika kaže dve Youngovi tabeli. Leva je že standardna, desno pa lahko z eno zamenjavo spremenimo v standardno Youngovo tabelo (zamenjamo 6 in 9).

20. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

24. januarja 2025

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Koščki

Označimo širino in višino mreže z A in B. Vogalni koščki so vedno štirje; robnih je $A-2$ na vsaki od obeh stranic dolžine A (preostala dva koščka sta vogalna) in podobno $B-2$ na vsaki od obeh stranic dolžine B, skupaj torej $2(A-2)+2(B-2)$; notranjih koščkov pa je $(A-2)(B-2)$. Da bo v bodoče manj pisanja, vpeljimo $a = A-2$ in $b = B-2$; robnih koščkov je torej $2a+2b$, notranjih pa $a \cdot b$.

Razmislimo najprej o primeru, ko ne manjka noben košček. Takrat torej dobimo $v = 4$, $r = 2a+2b$ in $n = a \cdot b$. Ena možnost je, da se naloge lotimo bolj matematično: iz enačbe $r = 2a+2b$ izrazimo $b = r/2 - a$, kar lahko nesemo v enačbo $n = ab$ in dobimo $n = a(r/2-a)$, iz tega pa $a^2 - (r/2)a + n = 0$. Po znani formuli za rešitev kvadratne enačbe dobimo rešitvi $a_{1,2} = r/4 \pm \sqrt{r^2/16 - n} = \big(r \pm \sqrt{r^2-16n}\big)/4$. Eno od teh dveh rešitev lahko vzamemo za a in če potem iz tega a izračunamo b po formuli $b = r/2 - a$, bomo opazili, da je to ravno druga rešitev omenjene kvadratne enačbe. Zato je pravzaprav vseeno, katero od obeh rešitev vzamemo za a in katero za b, saj je iz vhodnih podatkov pri tej nalogi tako ali tako nemogoče ugotoviti, katera od obeh stranic je višina, katera pa širina (na primer: mreža 6×5 in mreža 5×6 imata obe enako število koščkov vsake vrste). Če je izraz pod korenom negativen ali pa dobimo ne-celoštevilska a in b, je to znak, da je problem za te v, r in n nerešljiv.

```
#include <cmath>
#include <cstdio>
using namespace std;

// Reši nalogo za primer, ko ne manjka noben košček.
// Vrne true, če rešitev obstaja, sicer false.
bool ResiZEnacbo(int v, int r, int n)
{
  // Preverimo, ali so vogalni koščki res štirje, robnih pa je sodo mnogo.
  if (v != 4 || r % 2 != 0) return false;

  // Vzeli bomo a = (r + sqrt(r^2 - 16n))/4.
  int D = r * r - 16 * n;
  if (D < 0)          // Pri negativnem D kvadratna enačba nima realne rešitve
    return false;     // (imamo preveč notranjih oz. premalo robnih koščkov).
  int koren = (int) round(sqrt(D));
  if (koren * koren != D) return false; // Rešitev ni celoštevilska.
  int a = r + koren;
  if (a % 4 != 0) return false; // Rešitev ni celoštevilska.
  a /= 4;

  // Izračunajmo še b in izpišimo rezultat.
  int b = r / 2 - a;
  printf("A = %d, B = %d\n", a + 2, b + 2); return true;
}
```

Bolj „računalniški“ pristop pa je, da v zanki preizkušamo različne a-je (od 0 naprej) in pri vsakem preverimo, ali lahko z njim pridemo do rešitve, torej ali za tisti a obstaja neki tak b, ki bo skupaj s tem a-jem izpolnil pogoja $r = 2a+2b$ in $n = a \cdot b$. Preverimo na primer, če je $b = n/a$ celo število in če zanj velja $r = 2a+2b$; ali pa preverimo, če je $b = (r-2a)/2$ celo število in če zanj velja $n = a \cdot b$. Slabost te rešitve je, da moramo preizkusiti $O(r)$ ali $O(\sqrt{n})$ različnih a-jev, medtem ko je rešitev s kvadratno enačbo izračunala rezultat v $O(1)$ časa.

```
// Kot ResiZEnacbo, vendar išče rešitve z zanko.
bool ResiZZanko(int v, int r, int n)
{
  // Preverimo, ali so vogalni koščki res štirje, robnih pa je sodo mnogo.
  if (v != 4 || r % 2 != 0) return false;
  for (int a = 0; ; ++a)
  {
    // Če je ena stranica a, kakšna mora biti druga stranica,
    // da bomo dobili natanko r robnih koščkov?
    int b = r / 2 - a;
    if (b < a) break; // Omejimo se lahko na primere, ko je a krajša od obeh stranic.
    // Ali pri teh a in b res dobimo n notranjih koščkov?
    if (a * b != n) continue;
    // Našli smo rešitev.
    printf("A = %d, B = %d\n", a + 2, b + 2); return true;
  }
  return false; // Če pridemo do sem, rešitve nismo našli.
}
```

Doslej smo razmišljali o primeru, ko ne manjka noben košček. Primere, ko en košček manjka, lahko rešimo tako, da še trikrat poženemo enak postopek kot doslej, le da namesto trojice (v, r, n) uporabimo eno od $(v+1, r, n)$, $(v, r+1, n)$ ali $(v, r, n+1)$. Čim najdemo kakšno rešitev, se lahko ustavimo in ostalih možnosti ne pregledujemo; spodnja rešitev v ta namen izkoristi dejstvo, da se pri operatorju || (logični ali) desni operand sploh ne izračuna, če ima levi operand vrednost **true**.

```
bool Resi(int v, int r, int n)
{
  return ResiZEnacbo(v, r, n) || ResiZEnacbo(v + 1, r, n) ||
         ResiZEnacbo(v, r + 1, n) || ResiZEnacbo(v, r, n + 1);
}
```

Zapišimo našo rešitev še v pythonu:

```
import math

# Reši nalogo za primer, ko ne manjka noben košček.
# Vrne True, če rešitev obstaja, sicer False.
def ResiZEnacbo(v: int, r: int, n: int) -> bool:
  # Preverimo, ali so vogalni koščki res štirje, robnih pa je sodo mnogo.
  if v != 4 or r % 2 != 0: return False
  # Vzeli bomo a = (r + sqrt(r^2 - 16n))/4.
  D = r * r - 16 * n
  if D < 0:          # Pri negativnem D kvadratna enačba nima realne rešitve
    return False     # (imamo preveč notranjih oz. premalo robnih koščkov).
  koren = round(math.sqrt(D))
  if koren * koren != D: return False # Rešitev ni celoštevilska.
  a = r + koren
  if a % 4 != 0: return False # Rešitev ni celoštevilska.
  a //= 4
  # Izračunajmo še b in izpišimo rezultat.
  b = r // 2 - a
  print("A = %d, B = %d" % (a + 2, b + 2)); return True

# Kot ResiZEnacbo, vendar išče rešitve z zanko.
def ResiZanko(v: int, r: int, n: int) -> bool:
  # Preverimo, ali so vogalni koščki res štirje, robnih pa je sodo mnogo.
  if v != 4 or r % 2 != 0: return False
  for a in range(r + 1):
    # Če je ena stranica a, kakšna mora biti druga stranica,
    # da bomo dobili natanko r robnih koščkov?
```

```
    b = r // 2 - a
    if b < a: break # Omejimo se lahko na primere, ko je a krajša od obeh stranic.

    # Ali pri teh a in b res dobimo n notranjih koščkov?
    if a * b != n: continue

    # Našli smo rešitev.
    print("A = %d, B = %d" % (a + 2, b + 2)); return True

  return False # Če pridemo do sem, rešitve nismo našli.

def Resi(v: int, r: int, n: int) -> bool:
  return (ResiZEnacbo(v, r, n) or ResiZEnacbo(v + 1, r, n) or
          ResiZEnacbo(v, r + 1, n) or ResiZEnacbo(v, r, n + 1))
```

Manj pazljiva različica rešitve z enačbo. Naša funkcija ResiZEnacbo je pazljivo preverjala, ali po formuli $a = (r + \sqrt{r^2 - 16n})/4$ dobimo celoštevilsko vrednost. Za konec si oglejmo še malo preprostejšo in manj pazljivo različico te rešitve, ki rezultat korenjenja preprosto zaokroži na najbližje celo število. Izkazalo se bo, da tudi ta rešitev vedno daje pravilne rezultate, nekaj več truda pa bo potrebnega, da bomo to tudi dokazali. Naša poenostavljena rešitev je naslednja:

```
bool ResiZEnacbo2(int v, int r, int n)
{
  // Preverimo, ali so vogalni koščki res štirje, robnih pa je sodo mnogo.
  if (v != 4 || r % 2 != 0) return false;

  // Vzeli bomo a = (r + sqrt(r^2 - 16n))/4.
  int D = r * r - 16 * n;
  int koren = (int) round(sqrt(D));
  int a = (r + koren) / 4;

  // Izračunajmo še b in izpišimo rezultat.
  int b = r / 2 - a;
  printf("A = %d, B = %d\n", a + 2, b + 2); return true;
}
```

Recimo, da glavno funkcijo Resi spremenimo tako, da kliče našo novo ResiZEnacbo2 namesto stare ResiZEnacbo. Če jo preizkusimo na več testnih primerih, bomo videli, da vedno vrača pravilne rezultate. Kako to, da zaradi zaokrožanja nikoli ne pride do napak? Razmislimo podrobneje o delovanju naše nove poenostavljene rešitve in o tem, kako in zakaj res vedno pride do pravega rezultata.

Ključno za pravilnost naše nove rešitve je zagotovilo iz besedila naloge, da rešitev vedno obstaja. Z drugimi besedami, predpostaviti smemo, da se funkcijo Resi kliče le za take trojice (v, r, n), ki res predstavljajo število koščkov vsake vrste (vogalnih, robnih, notranjih) v neki pravokotni mreži, pri čemer en košček morda manjka. Za to, da je rešitev pravilna, je dovolj že, če deluje na takih trojicah, na ostalih pa sme vračati napačne rezultate (kar v praksi pomeni, da izpiše neko velikost mreže, namesto da bi ugotovila, da dana vhodna trojica (v, r, n) ne more nastati pri nobeni velikosti mreže).

Recimo torej, da je vhodna trojica (v, r, n), ki jo je dobila funkcija Resi, nastala tako, da smo imeli pravokotno mrežo velikosti $(\alpha+2) \times (\beta+2)$ (za neki celoštevilski $\alpha, \beta \geq 0$) in v njej prešteli koščke vseh treh vrst, pri čemer je en košček morda manjkal. Za (v, r, n) smo torej dobili trojico ene od naslednjih oblik: $(4, 2\alpha + 2\beta, \alpha\beta)$ ali $(3, 2\alpha + 2\beta, \alpha\beta)$ ali $(4, 2\alpha + 2\beta - 1, \alpha\beta)$ ali $(4, 2\alpha + 2\beta, \alpha\beta - 1)$. Spomnimo se, da naredi funkcija Resi štiri poskuse: najprej poskuša (zdaj s podprogramom ResiZEnacbo2 namesto ResiZEnacbo) poiskati rešitev za trojico (v, r, n); če to ne uspe, poskusi nato s trojico $(v + 1, r, n)$; če tudi to ne uspe, poskusi nato z $(v, r + 1, n)$; in če ne uspe niti ta, poskusi nazadnje še s trojico $(v, r, n + 1)$.

Če je Resi na vhodu kot (v, r, n) dobila trojico oblike $(4, 2\alpha+2\beta, \alpha\beta)$, bo našla pravo rešitev že v prvem poskusu: takrat namreč ResiZEnacbo2 izračuna $D = r^2 - 16n = 4(\alpha + \beta)^2 - 16\alpha\beta = 4(\alpha - \beta)^2$, zato dobi koren $= 2(\alpha - \beta)$ in nato $a = (r + \text{koren})/4 = (2(\alpha + \beta) - 2(\alpha - \beta))/4 = \alpha$ ter $b = r/2 - a = 2(\alpha + \beta) - \alpha = \beta$.

Če je Resi na vhodu kot (v, r, n) dobila trojico $(3, 2\alpha + 2\beta, \alpha\beta)$, bo v prvem poskusu — pri trojici (v, r, n) — funkcija ResiZEnacbo2 takoj vrnila **false**, ker bo opazila, da v ni

enak 4. V drugem poskusu, pri trojici $(v+1, r, n)$, pa bo našla pravo rešitev, $a = \alpha$ in $b = \beta$ (izračun je enak kot v prejšnjem odstavku).

Če je Resi na vhodu kot (v, r, n) dobila trojico $(4, 2\alpha+2\beta-1, \alpha\beta)$, bo v prvem poskusu — pri trojici (v, r, n) — funkcija ResiZEnacbo2 takoj vrnila **false**, ker bo opazila, da r ni sod. V drugem poskusu, pri trojici $(v+1, r, n)$, bo tudi takoj vrnila **false**, ker bo opazila, da v ni enak 4. V tretjem poskusu, pri trojici $(v, r+1, n)$, pa bo ResiZEnacbo2 našla pravo rešitev, $a = \alpha$ in $b = \beta$ (izračun je spet čisto tak kot v prejšnjih dveh odstavkih).

Ostane še možnost, da Resi na vhodu kot (v, r, n) dobi trojico $(4, 2\alpha + 2\beta, \alpha\beta - 1)$. V prvem poskusu, pri (v, r, n), izračuna funkcija ResiZEnacbo2 najprej $D = r^2 - 16n = 4(\alpha-\beta)^2+16$. Za nadaljevanje razmisleka je koristno ločiti dva primera. (1) Če je $\alpha = \beta$, smo dobili $D = 16$, zato bomo v naslednji vrstici dobili koren $= 4$ in nato $a = (r+4)/4 = (2\alpha + 2\beta) + 4)/4 = (4\alpha + 4)/4 = \alpha + 1$ ter $b = r/2 - a = (\alpha + \beta) - (\alpha + 1) = \beta - 1$. Dobili smo rešitev, ki je sicer nepričakovana, vendar povsem veljavna. Na primer, če je klicatelj imel mrežo 12×12 z enim manjkajočim notranjim koščkom, je imel $v = 4$ vogalne, $r = 40$ robnih in $n = 99$ notranjih koščkov; toda do enakega števila koščkov vsake vrste bi prišli tudi, če bi imeli mrežo 13×11 brez manjkajočih koščkov. Naloga pravi, da če obstaja več rešitev, je vseeno, katero izpišemo, tako da je rezultat $a = \alpha+1$ in $b = \beta - 1$ v tem primeru tudi pravilen.

(2) Druga možnost je, da sta α in β različni. Recimo brez izgube za splošnost, da je $\alpha > \beta$. Ker je D nekoliko (za 16) večji od $4(\alpha - \beta)^2$, je $\sqrt{D}$ tudi nekoliko večji od $2(\alpha - \beta)$; za spremenljivko koren pa bomo dobili vrednost $\sqrt{D}$, zaokroženo na najbližje celo število. Ali se lahko zgodi, da bi po tem zaokrožanju dobili vrednost, ki bi bila od $2(\alpha - \beta)$ večja vsaj za 4? To se zgodi, če je $\sqrt{D} \geq 2(\alpha - \beta) + 3{,}5$, kar naprej pomeni $D \geq 4(\alpha - \beta)^2 + 14(\alpha - \beta) + 49/4$, torej $16 \geq 14(\alpha - \beta) + 49/4$, torej $\alpha - \beta \leq 15/56$. Toda to je nemogoče; leva stran je ≥ 1 (ker sta α in β celi števili in je $\alpha > \beta$), desna stran pa < 1, torej leva stran gotovo ni manjša ali enaka desni. Predpostavka, da bi za koren dobili vrednost vsaj $2(\alpha - \beta) + 4$, nas je pripeljala v protislovje; torej velja koren $= 2(\alpha - \beta) + t$ za neki $t \in \{0, 1, 2, 3\}$. Funkcija ResiZEnacbo2 zato v naslednjem koraku izračuna $a = (r + \text{koren})/4 = (2(\alpha + \beta) + 2(\alpha - \beta) + t)/4 = (4\alpha + t)/4$. Točna vrednost tega izraza bi bila torej $\alpha + t/4$, toda ker imamo v izvorni kodi naše funkcije tam celoštevilsko deljenje, se rezultat zaokroži navzdol na najbližje celo število, to pa je α (ker je $0 \leq t < 4$ in zato $0 \leq t/4 < 1$). Dobimo torej $a = \alpha$, v naslednji vrstici pa potem $b = r/2 - a = \beta$, torej smo našli pravo rešitev.

Vidimo torej, da Resi res vedno vrne pravo rešitev in da četrti klic funkcije ResiZEnacbo2, torej tisti s parametri $(v, r, n+1)$, celo sploh nikoli ne pride na vrsto. □

Razmislek, ki smo ga pravkar opravili, nam omogoča tudi, da rešitev še malo poenostavimo. Namesto da v funkciji Resi delamo štiri poskuse (štirikrat kličemo funkcijo ResiZEnacbo2), lahko le pogledamo, katerega od prvih treh poskusov bi bilo treba izvesti: če je $v = 3$, je to znak, da manjka eden od vogalnih koščkov in bomo pravi rezultat dobili pri $(v+1, r, n)$; če je r lih, je to znak, da manjka eden od robnih koščkov in bomo pravi rezultat dobili pri $(v, r+1, n)$; sicer pa bodisi ne manjka noben košček bodisi manjka eden od notranjih koščkov, rezultat pa bomo dobili pri (v, r, n).

```
void Resi2(int v, int r, int n)
{
  if (v == 3) ++v;            // Morda manjka eden od vogalnih koščkov.
  else if (r % 2 == 1) ++r;   // Morda manjka eden od robnih koščkov.
  ResiZEnacbo2(v, r, n);
}
```

2. Tvorjenje besed

Recimo, da nas zanima, ali je mogoče (po pravilih iz besedila naloge) dobiti besedo t iz besede s. Opazimo lahko, da se lahko pri pravilih iz besedila naloge dolžina besede spremeni največ za 1; če se torej s in t razlikujeta po dolžini za več kot 1, potem gotovo ni mogoče dobiti ene iz druge. Če sta s in t enako dolgi, potem druga gotovo ne more nastati iz prve z brisanjem ali vrivanjem črke, zato moramo preveriti le, ali je nastala t iz s s spremembo ene črke. V zanki torej primerjajmo istoležne znake obeh besed — na primer $s[i]$ in $t[i]$ — in štejmo, na koliko mestih se istoležna znaka razlikujeta. Če

opazimo natanko eno neujemanje, potem je mogoče dobiti t iz s s spremembo ene črke, sicer pa ne (in sta besedi bodisi enaki bodisi je prišlo do spremembe več kot ene črke).

Podobno, če je t za en znak daljša od s, potem gotovo ni nastala iz s z brisanjem ali spremembo črke, zato moramo preveriti le, ali je nastala z vrivanjem ene črke. Tudi tu lahko primerjamo istoležne znake obeh besed; ko opazimo prvo neujemanje, vemo, da je trenutna črka t-ja tista, ki bi jo bilo treba vriniti v s, od tod naprej pa se morajo vse črke ujemati (sicer se besedi preveč razlikujeta). Pri tem preverjanju, ali se od vrinjene črke naprej besedi ujemata, pa moramo seveda paziti na to, da so črke t-ja zamaknjene za eno mesto v desno glede na črke s-ja, ker je bila v t-ju prednje vrinjena nova črka; primerjati moramo torej $s[i-1]$ in $t[i]$.

Ostane še primer, ko je t za en znak krajša od s; takrat lahko nastane iz s le z brisanjem ene črke. To bi lahko preverjali s podobnim razmislekom kot v prejšnjem odstavku, še lažje pa je, če takrat s in t v mislih zamenjamo in preverjamo, ali je t mogoče dobiti iz s z vrivanjem ene črke (torej lahko brez sprememb uporabimo postopek iz prejšnjega odstavka).

Da z implementacijo rešitve ne bo preveč dela, si lahko pomagamo še z naslednjim opažanjem: naša postopka za primer, ko sta s in t enako dolga, in za primer, ko je t za eno črko daljši od s, sta si v resnici zelo podobna; razlika je le v tem, da v prvem primeru od trenutka, ko opazimo prvo neujemanje, še naprej primerjamo istoležne znake obeh nizov ($s[i]$ in $t[i]$), v drugem primeru pa po prvem neujemanju znake drugega niza zamaknemo (primerjamo $s[i-1]$ in $t[i]$).

```
#include <cstring>
#include <cmath>
#include <utility>
using namespace std;

bool StaPodobni(const char *s, const char *t)
{
  // Preverimo, če se dolžini razlikujeta največ za 1.
  int sd = strlen(s), td = strlen(t);
  if (abs(sd - td) > 1) return false;

  // Če sta različno dolga, naj bo t daljši od s.
  if (sd > td) { swap(s, t); swap(sd, td); }

  // Primerjajmo znak po znak.
  bool neujemanje = false;
  for (int i = 0; i < td; ++i)

    // Če smo že opazili eno neujemanje in je t daljši od s, to pomeni, da
    // je bil v t vrinjen en znak, ki ga v s ni, in moramo zato odslej
    // primerjati t[i] s s[i - 1] namesto s s[i].
    if (t[i] == s[i - (neujemanje && td > sd ? 1 : 0)]) continue;

    // Če opazimo že drugo neujemanje, sta si preveč različna.
    else if (neujemanje) return false;
    else neujemanje = true;

  // Če pridemo do konca z natanko enim neujemanjem, se besedi razlikujeta ravno prav.
  return neujemanje;
}
```

Še ena možna rešitev je, da primerjamo znake od začetka obeh nizov do prvega neujemanja in nato še od konca obeh nizov v levo spet do prvega neujemanja; če tako odkrita ujemanja pokrijejo v daljšem od obeh nizov (ali v obeh nizih, če sta enako dolga) vse znake razen enega, potem se razlikujeta ravno prav, da je mogoče dobiti enega iz drugega po pravilih iz besedila naloge. Oglejmo si implementacijo te rešitve v pythonu:

```
def StaPodobni(s: str, t: str) -> bool:
  # Preverimo, če se dolžini razlikujeta največ za 1.
  sd = len(s); td = len(t)
  if abs(sd - td) > 1: return False

  # Če sta različno dolga, naj bo t daljši od s.
```

```
  if sd > td: s, sd, t, td = t, td, s, sd

  # Primerjajmo znak po znak z leve.
  i = 0
  while i < sd and s[i] == t[i]: i += 1

  # Primerjajmo znak po znak z desne.
  j = 0
  while j < sd - i and s[sd - 1 - j] == t[td - 1 - j]: j += 1

  # Zdaj vemo, da se niza ujemata v prvih i znakih ter v zadnjih j znakih.
  # Ali pokrije to v daljšem nizu (torej t) vse znake razen enega?
  return i + j == td - 1
```

3. Simetrična matrika

Označimo z $M[i, j]$ vrednost v celici na preseku i-te vrstice in j-tega stolpca. Pogoj za simetričnost je, da pri vsakih i in j velja, da sta $M[i, j]$ in $M[j, i]$ enaki. Pojdimo torej z dvema gnezdenima zankama po vseh kombinacijah i in j in pri vsaki poglejmo, kakšni vrednosti sta v celicah $M[i, j]$ in $M[j, i]$. Če sta obe prazni, lahko vanju vpišemo katerokoli vrednost, vendar seveda v obe enako. Če je ena prazna, druga pa vsebuje neko vrednost, moramo to vrednost vpisati še v prazno celico. Če pa sta obe celici že neprazni, moramo preveriti, ali sta njuni vrednosti enaki; če sta, je tu vse v redu, sicer pa lahko takoj zaključimo, da se dane mreže ne bo dalo dopolniti v simetrično (ker pogoj $M[i, j] = M[j, i]$ za trenutni celici ne bo veljal ne glede na to, kaj počnemo z morebitnimi praznimi celicami drugod v mreži). Za predstavitev mreže v našem programu uporabimo na primer dvodimenzionalno tabelo (*array*). Oglejmo si implementacijo takšne rešitve v C++:

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

int main()
{
  // Preberimo vhodne podatke.
  int n, M[100][100];
  cin >> n;
  for (int i = 0; i < n; ++i) for (int j = 0; j < n; ++j) cin >> M[i][j];

  // Dopolnimo mrežo v simetrično (ali ugotovimo, da je to nemogoče).
  bool ok = true;
  for (int i = 0; i < n && ok; ++i) for (int j = 0; j <= i; ++j)

    // Obdelajmo celici M[i][j] in M[j][i].
    // Če sta obe še prazni, vpišimo v obe kar število 1.
    if (M[i][j] < 0 && M[j][i] < 0) M[i][j] = 1, M[j][i] = 1;

    // Če je ena prazna, druga pa ne, skopirajmo vrednost druge v prvo.
    else if (M[i][j] < 0) M[i][j] = M[j][i];
    else if (M[j][i] < 0) M[j][i] = M[i][j];

    // Sicer sta obe neprazni; preverimo, če imata enako vrednost.
    else if (M[i][j] != M[j][i]) { ok = false; break; }

  // Izpišimo rezultate.
  if (! ok) cout << "NE" << endl;
  else for (int i = 0; i < n; ++i) for (int j = 0; j < n; ++j) {
    cout << M[i][j];
    if (j < n - 1) cout << "  "; else cout << endl; }
  return 0;
}
```

Zapišimo to rešitev še v pythonu:

```
# Preberimo vhodne podatke.
n = int(input())
M = [[int(s) for s in input().split()] for i in range(n)]

# Dopolnimo mrežo v simetrično (ali ugotovimo, da je to nemogoče).
```

```
ok = True
for i in range(n):
  for j in range(i + 1):
    # Obdelajmo celici M[i][j] in M[j][i].
    # Če sta obe še prazni, vpišimo v obe kar število 1.
    if M[i][j] < 0 and M[j][i] < 0: M[i][j] = 1; M[j][i] = 1
    # Če je ena prazna, druga pa ne, skopirajmo vrednost druge v prvo.
    elif M[i][j] < 0: M[i][j] = M[j][i]
    elif M[j][i] < 0: M[j][i] = M[i][j]
    # Sicer sta obe neprazni; preverimo, če imata enako vrednost.
    elif M[i][j] != M[j][i]: ok = False; break
  if not ok: break
# Izpišimo rezultate.
if not ok: print("NE")
else:
  for i in range(n): print(" ".join(str(x) for x in M[i]))
```

4. Prosti tok

Ker mreža ni velika (največ 100×100 celic), si lahko pripravimo tabelo $w \cdot h$ logičnih vrednosti, v katerih bomo označevali, ali je določena celica že pokrita ali ne. Z nekaj gnezdenimi zankami pojdimo po poteh, pri vsaki poti po odsekih, iz katerih je sestavljena, pri vsakem odseku pa po vseh celicah, po katerih ta odsek poteka. Za vsako takó obiskano celico preverimo, ali je pokrita že od prej; če še ni, jo zdaj v tabeli označimo kot pokrito; če pa je bila pokrita že od prej, je to znak, da se tu dve poti sekata (ali pa se trenutna pot tu seka sama s sabo) in lahko postopek takoj končamo. Če uspemo na ta način obdelati vse poti, ne da bi kdaj opazili, da se kje sekajo, moramo na koncu preveriti le še, ali so pokrite vse celice mreže. To lahko naredimo z zanko po celicah, lahko pa tudi sproti med obravnavo poti štejemo pokrite celice in na koncu le preverimo, če je število pokritih celic enako $w \cdot h$.

Oglejmo si implementacijo takšne rešitve v C++. Par koordinat predstavlja struktura tipa Koord; opis poti je vektor takšnih struktur; naš podprogram Preveri dobi vektor takšnih opisov poti kot parameter, poleg njega pa še širino in višino mreže.

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

struct Koord { int x, y; };
typedef vector<Koord> Pot;

bool Preveri(int w, int h, const vector<Pot> &poti)
{
  // M[y * w + x] pove, ali je kakšna pot že pokrila celico (x, y).
  vector<bool> M(w * h, false);
  int stPokritih = 0;
  // Pojdimo po poteh in poglejmo, katere celice pokrijejo.
  for (const Pot &P : poti)
  {
    // Pokrijmo začetno celico.
    int x = P[0].x, y = P[0].y;
    if (M[y * w + x]) return false; else M[y * w + x] = true, ++stPokritih;
    // Sledimo preostanku poti.
    for (int i = 1; i < P.size(); ++i)
    {
      int X = P[i].x, Y = P[i].y;
      // S trenutnega položaja (x, y) se počasi premaknimo do naslednje točke na poti, (X, Y).
      while (x != X || y != Y) {
        // Premaknimo se za eno celico proti (X, Y).
        if (x < X) ++x; else if (x > X) --x; else if (y < Y) ++y; else if (y > Y) --y;
        // Pokrijmo novo celico (x, y).
```

```
            if (M[y * w + x]) return false; else M[y * w + x] = true, ++stPokritih; }
        }
    }
    // Če pridemo do sem, vemo, da se poti ne sekajo; preverimo le še,
    return w * h == stPokritih;        // ali pokrijejo celotno mrežo.
}
```

Zapišimo to rešitev še v pythonu. Za predstavitev para koordinat bomo uporabili kar pythonov tip tuple:

```
def Preveri(w: int, h: int, poti: list[list[tuple[int, int]]]) -> bool:
    # M[y][x] pove, ali je kakšna pot že pokrila celico (x, y).
    M = [[False] * w for y in range(h)]
    stPokritih = 0

    # Pojdimo po poteh in poglejmo, katere celice pokrijejo.
    for P in poti:

        # Pokrijmo začetno celico.
        (x, y) = P[0]
        if M[y][x]: return False
        M[y][x] = True; stPokritih += 1
        # Sledimo preostanku poti.
        for i in range(1, len(P)):
            (X, Y) = P[i]
            # S trenutnega položaja (x, y) se počasi premaknimo do naslednje točke na poti, (X, Y).
            while x != X or y != Y:

                # Premaknimo se za eno celico proti (X, Y).
                if x < X: x += 1
                elif x > X: x -= 1
                elif y < Y: y += 1
                elif y > Y: y -= 1

                # Pokrijmo novo celico (x, y).
                if M[y][x]: return False
                M[y][x] = True; stPokritih += 1

    # Če pridemo do sem, vemo, da se poti ne sekajo; preverimo le še,
    return w * h == stPokritih      # ali pokrijejo celotno mrežo.
```

Časovna zahtevnost doslej opisane rešitve je $O(wh)$, kar je za namene našega tekmovanja čisto dovolj dobro, saj je mreža velika največ 100×100 celic; kot zanimivost pa vseeno omenimo še učinkovitejšo rešitev. Vsako pot si predstavljajmo kot zaporedje vodoravnih in navpičnih odsekov, pri čemer celico, v kateri se dva zaporedna odseka stikata, štejmo le k enemu od njiju (vseeno, kateremu), ne k obema. Vsak odsek si lahko tudi predstavljamo kot (navpično ali vodoravno) daljico v ravnini. Naloga zahteva, da preverimo, ali se kakšna dva odseka sekata oz. prekrivata in ali vsi odseki skupaj pokrijejo ravno celotno mrežo. Prvi del tega vprašanja je le poseben primer znanega problema iz računske geometrije, kjer dobimo množico d daljic in nas zanima, ali se kakšni dve od njih sekata; to je mogoče rešiti s preletom ravnine v $O(d \log d)$ časa (v našem primeru torej d pomeni skupno število odsekov na vseh poteh).[1] Če se izkaže, da se odseki nikjer ne sekajo oz. prekrivajo, je potem število pokritih celic ravno enako skupni dolžini vseh odsekov; torej moramo le še sešteti dolžine odsekov in preveriti, če je vsota enaka wh; če je, so pokrite vse celice mreže, sicer pa nekatere niso. Tako smo torej nalogo rešili v $O(d \log d)$ časa in časovna zahtevnost ni več odvisna od površine mreže.

5. Standardna Youngova tabela

Preglejmo vhodno tabelo po vrsticah in po stolpcih ter poiščimo vse pare sosednjih celic (oz. škatlic), kjer leva ni manjša od desne (če sta v isti vrstici) oz. zgornja ni manjša od spodnje (če sta v istem stolpcu); tem parom recimo *neugodni*. Če ni nobenega neugodnega para, lahko takoj zaključimo, da je tabela že standardna, in končamo.

[1] Gl. npr. Wikipedijo *s. v.* "Multiple line segment intersection" in tam navedeno literaturo.

Sicer naj bo (a, b) eden od neugodnih parov. Če naj bi se dalo tabelo predelati v standardno z eno zamenjavo, bo morala v tej zamenjavi sodelovati vsaj ena od vrednosti a in b, sicer bo napaka na tistem mestu ostala. Toda enak razmislek lahko opravimo za vsak neugoden par; za zamenjavo bomo morali torej izbrati dve taki vrednosti, da bo v vsakem neugodnem paru udeležena vsaj ena od njiju. Toda posamezna vrednost je lahko prisotna v največ štirih parih, saj ima posamezna celica v tabeli največ štiri sosede. Če je torej neugodnih parov več kot osem, lahko takoj zaključimo, da se tabele ne bo dalo z eno zamenjavo predelati v standardno.

Sicer imamo torej največ osem neugodnih parov, v katerih nastopa skupno največ 16 vrednosti. Za vsaki dve od teh vrednosti, recimo u in v, lahko zdaj preverimo, ali je v vsakem neugodnem paru prisotna vsaj ena od njiju, in če je, lahko nato še preverimo, ali bi z zamenjavo vrednosti u in v tabela postala standardna. Poseben primer pa nastopi, če je kakšna vrednost u že sama po sebi prisotna v vseh neugodnih parih (to se lahko zgodi, če so neugodni pari kvečjemu štirje); tedaj ni nujno, da si za v izberemo eno od ostalih vrednosti iz neugodnih parov, pač pa moramo pri tem u preizkusiti zamenjave z vsemi drugimi vrednostmi tabele (torej $n - 1$ zamenjav).

V najslabšem primeru bomo morali torej preizkusiti $O(n)$ zamenjav. Za vsako možno zamenjavo pa lahko v $O(1)$ časa preverimo, ali bi tabela po tej zamenjavi postala standardna: za vsakega od (starih) neugodnih parov preverimo, če bi po zamenjavi prenehal biti neugoden, in nato za obe zamenjani vrednosti pregledamo njune sosede in preverimo, ali bi po zamenjavi kakšna od njiju tvorila s kakšnim od teh sosedov nov neugoden par. Če se pri tem preizkusu izkaže, da so vsi stari neugodni pari prenehali biti neugodni in da ni noben par na novo postal neugoden, bo tabela po tej zamenjavi standardna.

20. srednješolsko tekmovanje ACM v znanju računalništva

Šolsko tekmovanje

24. januarja 2025

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. Koščki

- Vse točke lahko dobijo tako rešitve, ki rešujejo kvadratno enačbo in porabijo $O(1)$ časa, kot rešitve, ki pregledujejo možne dolžine stranic v zanki in porabijo $O(r)$ ali $O(\sqrt{n})$ časa. Če pa bi bila kakšna rešitev še bolj neučinkovita in bi porabila $O(n)$ časa, naj dobi največ 15 točk, če je sicer pravilna.

- Primer, ko eden od koščkov manjka, smo v naši rešitvi obravnavali tako, da smo štirikrat klicali isti podprogram. Manj elegantna možnost bi bila, da bi napisali štiri kopije bolj ali manj enake (oz. zelo podobne) izvorne kode; tudi take rešitve naj dobijo vse točke, če so sicer pravilne.

- Funkcija ResiZEnacbo v naši rešitvi pazljivo preverja, ali bi po formuli $a = (r + \sqrt{r^2 - 16n})/4$ res dobila celoštevilski a; za enako dobro naj šteje tudi rešitev, ki bi a izračunala manj pazljivo, zaokrožila koren v omenjeni formuli na najbližje celo število in nato tako dobljeni a uporabila skupaj z $b = r/2 - a$. Čisto lahko se namreč zgodi (odvisno od podrobnosti rešitve), da tudi taka rešitev daje pravilne rezultate (primer je funkcija ResiZEnacbo2 iz naše rešitve); glavno vprašanje pri taki rešitvi je potem, ali znamo tudi dokazati, da res vedno daje pravilne rezultate, vendar takšnega dokaza od tekmovalcev na šolskem tekmovanju tako ali tako ne moremo pričakovati.

2. Tvorjenje besed

- Za vse točke pričakujemo rešitev z linearno časovno zahtevnostjo v odvisnosti od dolžine obeh nizov, $O(|s| + |t|)$. Morebitne rešitve z zahtevnostjo $O(|s| \cdot |t|)$ naj dobijo največ 12 točk, če so sicer pravilne; morebitne rešitve s še večjo časovno zahtevnostjo pa naj dobijo največ 8 točk, če so sicer pravilne.

- Če sta dani dve besedi popolnoma enaki, je pravilni odgovor načeloma ta, da ni mogoče dobiti ene iz druge, ker pravila iz besedila naloge zahtevajo, da eno črko spremenimo (ali vrinemo ali pobrišemo). Rešitvam, ki v takem primeru pomotoma odgovorijo, da je mogoče dobiti eno besedo iz druge, naj se zaradi tega odšteje dve točki.

3. Simetrična matrika

- Točkovanje pri tej nalogi naj bo približno takšno: 5 točk za branje vhodnih podatkov in izpis rezultatov; 5 točk za pravilno obravnavo primerov, ko sta $M[i, j]$ in $M[j, i]$ obe prazni; 5 točk za pravilno obravnavo primerov, ko je natanko ena od $M[i, j]$ in $M[j, i]$ prazna; in 5 točk za pravilno obravnavo primerov, ko sta $M[i, j]$ in $M[j, i]$ obe neprazni.

- V naši rešitvi gremo z eno zanko po i od 0 do $n - 1$ in pri vsakem i z notranjo zanko po j od 0 do i. Morebitnim rešitvam, ki bi šle v notranji zanki vsakič od 0 do $n - 1$ in torej po nepotrebnem pregledale tabelo dvakrat, naj se zaradi te drobne neučinkovitosti ne odšteva točk.

- Če bi šla morda notranja zanka od 0 do $i-1$ namesto do i in bi rešitev zato pozabila pregledati celice na glavni diagonali (in si izmisliti vrednosti za morebitne prazne celice na glavni diagonali), naj se ji zaradi tega odšteje dve točki.

4. Prosti tok

- Naloga pravi, da si lahko reševalec sam izbere podrobnosti glede predstavitev vhodnih podatkov, vendar mora seveda ostati v okviru dejstva, da so podane samo tiste koordinate, v katerih pot spremeni svojo smer. Če bi kakšna rešitev predpostavila, da so podane koordinate vseh celic, ki jih pot obišče, postane problem zaradi tega lažji; takšna rešitev naj dobi največ 10 točk, če je sicer pravilna.
- Rešitev sme predpostaviti, da je opis poti sestavljen iz vsaj dveh točk in da nobeni dve zaporedni točki v opisu poti nista enaki.
- V celicah, kjer pot preide iz vodoravnega v navpični odsek ali obratno, je potrebne nekaj pazljivosti, da jih ne obravnavamo dvakrat (enkrat pri vodoravnem in enkrat pri navpičnem odseku) in zato pomotoma dobimo vtis, da se tam dve poti sekata. Rešitvam, ki bi naredile to napako, naj se zaradi tega odšteje tri točke.
- Rešitev, ki bi imela kvadratno časovno zahtevnost v odvisnosti od števila poti ali od skupnega števila vseh odsekov na vseh poteh, naj dobi največ 15 točk, če je sicer pravilna. Če bi imela kvadratno zahtevnost v odvisnosti od skupnega števila celic na vseh poteh, naj dobi največ 10 točk. Do teh reči bi lahko prišlo, če bi rešitev na kakšen zelo neučinkovit način preverjala, ali se poti kdaj sekajo.

5. Standardna Youngova tabela

- Za vse točke pričakujemo rešitev s časovno zahtevnostjo $O(n)$. Rešitve, ki preizkusijo le $O(n)$ zamenjav (tako kot naša rešitev), vendar za obravnavo vsake od njih porabijo po $O(n)$ časa (npr. zato, ker vsakič pregledajo celo tabelo, da vidijo, ali bi bila po zamenjavi standardna) in imajo zato časovno zahtevnost $O(n^2)$, naj dobijo največ 15 točk. Rešitve, ki preizkusijo $O(n^2)$ zamenjav (na primer kar vse možne zamenjave dveh števil) in vsako od njih obravnavajo v $O(1)$ časa, naj dobijo največ 12 točk. Rešitve pa, ki imajo časovno zahtevnost $O(n^3)$ ali več (npr. ker preizkusijo $O(n^2)$ zamenjav in porabijo za vsako od njih po $O(n)$ časa), naj dobijo največ 7 točk.
- Naloga zahteva le opis postopka, ne pa implementacije. Zato tudi ni treba, da se opis spušča v vse podrobnosti (kot se tudi tisti v naši rešitvi ne). Radi bi predvsem, da se iz opisa vidi, da se je reševalec zavedal, da se morajo zamenjave dogajati v okolici tistih celic, kjer so bile v prvotnem stanju tabele napake (zaradi katerih tabela ni bila standardna), in da lahko stanje tabele po zamenjavi (ali je standardna ali ni) preverimo tako, da pregledamo celice v okolici zamenjanih, ni pa treba še enkrat pregledovati cele tabele.

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. Koščki	lažja do srednja naloga v prvi skupini
2. Tvorjenje besed	srednje težka naloga v prvi ali lahka v drugi skupini
3. Simetrična matrika	lažja naloga v prvi skupini
4. Prosti tok	srednje težka naloga v drugi ali lažja v tretji skupini
5. Youngova tabela	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.