11. tekmovanje ACM v znanju računalništva

Institut Jožef Stefan, Ljubljana, 19. marca 2016

Bilten

Bilten 11. tekmovanja ACM v znanju računalništva
Institut Jožef Stefan, 2016

Uredil Janez Brank

Avtorji nalog: Nino Bašić, Boris Gašperin, Matija Grabnar, Tomaž Hočevar, Boris Horvat, Vid Kocijan, Jurij Kodre, Matjaž Leonardis, Mark Martinec, Mitja Lasič, Matija Lokar, Jure Slak, Marjan Šterk, Patrik Zajec, Janez Brank.

Naklada: 200 izvodov

Vprašanja, pripombe, komentarji, popravki ipd. v zvezi z biltenom so dobrodošli. Pišite nam na naslov `rtk-info@ijs.si`.

CIP — Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

37.091.27:004(497.4)(0.034.2)

TEKMOVANJE ACM v znanju računalništva (11 ; 2016 ; Ljubljana)
Bilten [Elektronski vir] / 11. tekmovanje ACM v znanju računalništva, Ljubljana, 19. marca 2016 ; [avtorji nalog Nino Bašić ... [et al.] ; uredil Janez Brank]. — El. knjiga. — Ljubljana : Institut Jožef Stefan, 2016

ISBN 978-961-264-105-4 (PDF)
1. Bašić, Nino 2. Brank, Janez, 1979–
287731456

KAZALO

STRUKTURA TEKMOVANJA

Tekmovanje poteka v treh težavnostnih skupinah. Tekmovalec se lahko prijavi v katerokoli od teh treh skupin ne glede na to, kateri letnik srednje šole obiskuje. Prva skupina je najlažja in je namenjena predvsem tekmovalcem, ki se ukvarjajo s programiranjem šele nekaj mesecev ali mogoče kakšno leto. Druga skupina je malo težja in predpostavlja, da tekmovalci osnove programiranja že poznajo; primerna je za tiste, ki se učijo programirati kakšno leto ali dve. Tretja skupina je najtežja, saj od tekmovalcev pričakuje, da jim ni prevelik problem priti do dejansko pravilno delujočega programa; koristno je tudi, če vedo kaj malega o algoritmih in njihovem snovanju.

V vsaki skupini dobijo tekmovalci po pet nalog; pri ocenjevanju štejejo posamezne naloge kot enakovredne (v prvi in drugi skupini lahko dobi tekmovalec pri vsaki nalogi do 20 točk, v tretji pa pri vsaki nalogi do 100 točk).

V lažjih dveh skupinah traja tekmovanje tri ure; tekmovalci lahko svoje rešitve napišejo na papir ali pa jih natipkajo na računalniku, nato pa njihove odgovore oceni temovalna komisija. Naloge v teh dveh skupinah večinoma zahtevajo, da tekmovalec opiše postopek ali pa napiše program ali podprogram, ki reši določen problem. Pri pisanju izvorne kode programov ali podprogramov načeloma ni posebnih omejitev glede tega, katere programske jezike smejo tekmovalci uporabljati. Podobno kot v zadnjih nekaj letih smo tudi letos ponudili možnost, da tekmovalci v prvi in drugi skupini svoje odgovore natipkajo na računalniku; tudi tokrat se je za to odločila večina tekmovalcev, je pa bilo letos nekaj več kot prejšnja leta tudi primerov pisanja odgovorov na papir. Da bi bilo tekmovanje pošteno tudi do morebitnih reševalcev na papir, so bili na računalnikih za prvo in drugo skupino le urejevalniki besedil, ne pa tudi razvojna orodja, prevajalniki in dokumentacija o programskih jezikih in knjižnicah.

V tretji skupini rešujejo vsi tekmovalci naloge na računalnikih, za kar imajo pet ur časa. Pri vsaki nalogi je treba napisati program, ki prebere podatke iz vhodne datoteke, izračuna nek rezultat in ga izpiše v izhodno datoteko. Programe se potem ocenjuje tako, da se jih na ocenjevalnem računalniku izvede na več testnih primerih, število točk pa je sorazmerno s tem, pri koliko testnih primerih je program izpisal pravilni rezultat. (Podrobnosti točkovanja v 3. skupini so opisane na strani 20.) Letos so bili v 3. skupini dovoljeni programski jeziki pascal, C, C++, C#, java in VB.NET.

Nekaj težavnosti tretje skupine izvira tudi od tega, da je pri njej mogoče dobiti točke le za delujoč program, ki vsaj nekaj testnih primerov reši pravilno; če imamo le pravo idejo, v delujoč program pa nam je ni uspelo preliti (npr. ker nismo znali razdelati vseh podrobnosti, odpraviti vseh napak, ali pa ker smo ga napisali le do polovice), ne bomo dobili pri tisti nalogi nič točk.

Tekmovalci vseh treh skupin si lahko pri reševanju pomagajo z zapiski in literaturo, v tretji skupini pa tudi z dokumentacijo raznih prevajalnikov in razvojnih orodij, ki so nameščena na tekmovalnih računalnikih.

Na začetku smo tekmovalcem razdelili tudi list z nekaj nasveti in navodili (str. 7–9 za 1. in 2. skupino, str. 19–22 za 3. skupino).

Omenimo še, da so rešitve, objavljene v tem biltenu, večinoma obsežnejše od tega, kar na tekmovanju pričakujemo od tekmovalcev, saj je namen tukajšnjih rešitev

pogosto tudi pokazati več poti do rešitve naloge in bralcu omogočiti, da bi se lahko iz razlag ob rešitvah še česa novega naučil.

Poleg tekmovanja v znanju računalništva smo organizirali tudi tekmovanje v off-line nalogi, ki je podrobneje predstavljeno na straneh 126–129.

Podobno kot v zadnjih nekaj letih smo izvedli tudi šolsko tekmovanje, ki je potekalo 22. januarja 2016. To je imelo eno samo težavnostno skupino, naloge (ki jih je bilo pet) pa so pokrivale precej širok razpon težavnosti. Tekmovalci so pisali odgovore na papir in dobili enak list z nasveti in navodili kot na državnem tekmovanju v 1. in 2. skupini (str. 7–9). Odgovore tekmovalcev na posamezni šoli so ocenjevali mentorji z iste šole, za pomoč pa smo jim pripravili nekaj strani z nasveti in kriteriji za ocenjevanje (str. 112–116). Namen šolskega tekmovanja je bil tako predvsem v tem, da pomaga šolam pri odločanju o tem, katere tekmovalce poslati na državno tekmovanje in v katero težavnostno skupino jih prijaviti. Šolskega tekmovanja se je letos udeležilo 310 tekmovalcev s 27 šol (vse so bile srednje).

NASVETI ZA 1. IN 2. SKUPINO

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 (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). Za manjše sintaktične napake se 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 uporabiti fgets; vendar pa za rešitev naših tekmovalnih nalog v prvi in drugi skupini 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.");
    }
}
```

NALOGE ZA PRVO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del prek računalnika. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku (npr. kopiraj in prilepi v Notepad in shrani v datoteko). **Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave in bi rad, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici.**

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 (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

1. Tipkanje

Znašel si se v vlogi zapisnikarja, ki mora na računalniku vnesti seznam n besed. To počneš tako, da s pritiski na tipkovnico vnašaš črko po črko v vnosno polje, ki je na začetku prazno. Ko je v polju izpisana zahtevana beseda, zaključiš vnos s pritiskom na tipko Enter. Beseda pri tem ostane v vnosnem polju. Nato s pritiski na tipko Backspace pobrišeš nekaj (morda vse, ali pa nobene) zadnjih črk in nadaljuješ z vnosom naslednje besede. **Napiši program**, ki bo izpisal, najmanj koliko pritiskov tipk boš potreboval, da vneseš podane besede. Tvoj program lahko vhodne podatke bere s standardnega vhoda ali pa iz datoteke `besede.txt` (kar ti je lažje); v prvi vrstici je število besed, nato pa sledi ustrezno število vrstic, v vsaki od njih je po ena beseda. Besede vsebujejo le male črke angleške abecede. Posamezna beseda je dolga največ 100 znakov.

Primer vhoda:

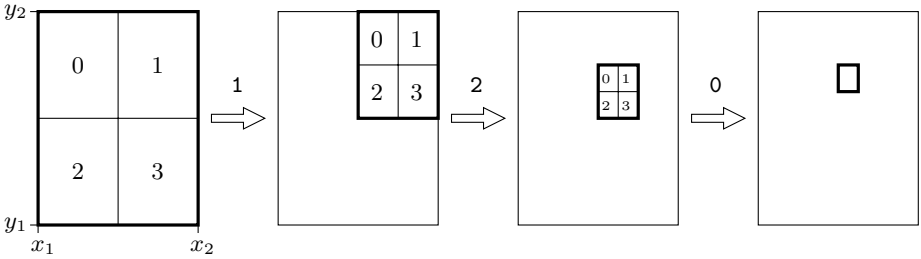
```
3
abc
aaaa
ba
```

Pripadajoči izhod:

```
17
```

2. Zoom

Na računalniku gledamo zemljevid, ki pokriva v koordinatnem sistemu območje $[x_1, x_2] \times [y_1, y_2]$, kot kaže spodnja slika. Razdelimo ga na štiri četrtine in jih označimo: 0 je zgornja leva, 1 je zgornja desna, 2 je spodnja leva in 3 je spodnja desna. Izberemo si eno od četrtin in pozumiramo prikaz zemljevida tako, da vidimo le še to četrtino. Tudi njo razdelimo na četrtine in pozumiramo v eno od njih. Ta korak še nekajkrat ponovimo. Z nizom, ki ga sestavljajo znaki `0`, `1`, `2` in `3`, lahko opišemo, v katero četrtino smo se premaknili na vsakem koraku. Naslednja slika kaže primer za niz `"120"`:



Napiši podprogram Zoom(s, x1, y1, x2, y2), ki za dani niz s (zaporedje znakov `0`, `1`, `2`, `3`, ki opisujejo potek zumiranja) in koordinate x_1, x_2, y_1, y_2 celotnega zemljevida (pri tem zagotovo velja $x_1 < x_2$ in $y_1 < y_2$) izpiše koordinate tistega območja, ki ga gledamo po zadnjem koraku zumiranja.

Tvoj podprogram naj bo takšne oblike:

```
procedure Zoom(s: string; x1, y1, x2, y2: double);                    { v pascalu }
void Zoom(char *s, double x1, double y1, double x2, double y2);       /* v C/C++ */
void Zoom(string s, double x1, double y1, double x2, double y2);      // v C++
public static void Zoom(String s, double x1, double y1, double x2, double y2);    // v javi
public static void Zoom(string s, double x1, double y1, double x2, double y2);    // v C#
def Zoom(s, x1, y1, x2, y2): ...      # v pythonu; s je tipa str, ostali pa tipa float
```

3. Zaklepajski izrazi

Oklepajski izrazi so nizi, ki jih sestavljajo sami oklepaji in zaklepaji, morajo pa biti pravilno gnezdeni. Oklepaji in zaklepaji so lahko različnih oblik: okrogli `( )`, oglati `[ ]`, zaviti `{ }` in kotni `< >`. „Pravilno gnezdeni“ pomeni, da mora imeti vsak oklepaj tudi pripadajoč zaklepaj enake oblike (in obratno), podniz med njima pa mora biti tudi sam zase oklepajski izraz.

Nekaj primerov oklepajskih izrazov: `<<>>`, `[()(<>)]()`, `{[{}()]<>}`.

Nekaj primerov nizov, ki *niso* oklepajski izrazi: `(()(`, `([)]`, `<>><<>`.

Dan je nek niz `s`, v katerem se pojavljajo le zaklepaji različnih oblik — `) ] } >` — in zvezdice `*`. **Napiši podprogram** Dopolni(s), ki vsako zvezdico v nizu `s` spremeni v enega od oklepajev — torej znakov `( [ { <` — tako, da bo iz tega na koncu nastal pravilno gnezden oklepajski izraz. Podprogram naj tako popravljeni niz izpiše, če pa se izkaže, da ga ni mogoče ustrezno popraviti (da bi nastal oklepajski izraz), naj izpiše, da je problem nerešljiv.

Primer: iz `"**]*))"` lahko naredimo `"([]())"`. Iz `"***>*)"` pa ne moremo narediti veljavnega oklepajskega izraza ne glede na to, kako spreminjamo zvezdice v oklepaje.

Tvoj podprogram naj bo takšne oblike:

```
procedure Dopolni(s: string);          { v pascalu }
void Dopolni(char *s);                 /* v C/C++ */
void Dopolni(string s);                // v C++
public static void Dopolni(String s);  // v javi
public static void Dopolni(string s);  // v C#
def Dopolni(s): ...                    # v pythonu; s je tipa str
```

4. Izštevanka

Otroci so se že malo naveličali vsakokrat uporabljati preprosto izštevanko „An ban pet podgan“ za določitev tistega, ki lovi, zato so se odločili za manjšo spremembo: vsak naj ima dve življenji, kdor ostane brez, je „izbrani“.

V krog se postavi n otrok, vsak stoji z obema nogama na tleh (t.j. ima dve življenji). Naj bodo oštevilčeni z zaporednimi številkami od 1 do n. Začnemo pri otroku številka 1 in od njega naredimo k korakov po krogu (pri tem torej njega ne štejemo, ampak štejemo šele otroke od 2 naprej) — k je pozitivno celo število, lahko je tudi večje od n. Otrok na tako določenem mestu mora dvigniti nogo (izgubi eno življenje). Če mora dvigniti še drugo in zato pasti, je izštevanke konec in ta otrok postane „izbrani“. Če pa še vedno stoji na eni nogi (je pravkar izgubil šele prvo življenje), se izštevanka nadaljuje (spet začne šteti) pri otroku neposredno za njim, hkrati pa izštevanko podaljšamo za eno besedo, torej povečamo k za 1.

Napiši program, ki bo prebral dve pozitivni celi števili — n in začetno vrednost k, opravil korake izštevanke in izpisal številko „izbranega“ otroka, to je tistega, ki je padel po tleh, ker je izgubil obe življenji. Število otrok n je vsaj 1 in kvečjemu 100.

Primer: če imamo $n = 5$ in $k = 3$, morajo po vrsti dvigovati noge otroci 4, 3, 3 in igre je konec — izbran je otrok številka 3.

5. H-indeks

Hirschev indeks (krajše tudi h-indeks) je ena izmed številnih ocen, ki poskušajo meriti uspešnost raziskovalcev. Cilj ocene h-indeks je uravnotežiti produktivnost (število objavljenih člankov) in vplivnost (citiranost njegovih člankov) posameznega raziskovalca. H-indeks je definiran kot največje celo število h, za katerega velja, da je raziskovalec objavil h člankov, kjer je bil vsak izmed teh člankov citiran vsaj h-krat. **Opiši postopek** (ali napiši program ali podprogram, kar ti je lažje), ki bo iz podanega seznama citiranosti člankov izračunal h-indeks. Posamezni elementi tega seznama so števila, ki za posamezne članke povedo, kolikokrat so bili citirani.

Primer: če imamo seznam $[6, 5, 3, 2, 5, 10, 5, 7]$, je njegov h-indeks enak 5, kajti v seznamu obstaja vsaj pet elementov, večjih ali enakih 5, ne obstaja pa v njem vsaj šest elementov, večjih ali enakih 6.

NALOGE ZA DRUGO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del prek računalnika. Vse te možnosti so enakovredne. Odgovore, oddane prek računalnika, bomo natisnili na papir in ocenjevali na enak način kot tiste, ki so bili že oddani na papirju.

Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko bi rad začasno prekinil pisanje rešitve naloge ter se lotil druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na lokalnem računalniku (npr. kopiraj in prilepi v Notepad in shrani v datoteko). **Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave in bi rad, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici.**

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 (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

1. Sorodstvo

Navdušencu za rodoslovje se je sesula baza podatkov o njegovih prednikih in sorodstvenih povezavah med njimi. Vse, kar je uspel rešiti, so letnice rojstev in smrti posameznih ljudi. Ima torej seznam parov celih števil (r_i, s_i) za $i = 1, \ldots, n$, ki povedo, da se je oseba i rodila v letu r_i in umrla v letu s_i. Vrstni red ljudi v seznamu je lahko poljubno premešan, torej ni nujno, da so npr. urejeni po letu rojstva ali kaj podobnega.

Predpostavimo, da je razlika v starosti staršev in otrok vsaj d let (d je neko celo število, večje od 0). Zato bomo rekli, da je oseba j morebitni otrok osebe i natanko tedaj, ko velja $r_i + d \leq r_j \leq s_i$.

Opiši postopek, ki za dani d in podatke o letih rojstev in smrti $(r_1, s_1), \ldots, (r_n, s_n)$ sestavi najdaljše zaporedje oseb, v katerem velja, da je vsaka naslednja oseba v zaporedju morebitni otrok prejšnje osebe v zaporedju.

Tvoja rešitev naj se ne opira na kakšne realistične predpostavke o tem, kako dolgo ljudje živijo in koliko otrok imajo, saj imajo nekateri uporabniki v svojem rodbinskem drevesu tudi vesoljce, sumerske polbogove in podobna dolgoživa bitja.

2. Suhi dnevi

Avtomatska vremenska postaja meri količino padavin. Večkrat na dan se tako odčita, koliko padavin je padlo od prejšnjega odčitka, ta podatek se vsakokrat zapiše v datoteko kot dve pozitivni celi števili: prvo število v vrstici je datum (kako je to število sestavljeno, ni določeno, zagotovljeno je le, da ima isti dan vedno enako številko, različno od drugih dni); drugo število v vrstici je količina novozapadlih padavin od prejšnjega odčitka. Zagotovimo lahko, da je v vsakem dnevu vsaj en odčitek in da je zadnji odčitek vsakega dneva opolnoči (torej ni neizmerjenih ostankov, ki bi se prenašali v naslednji dan). Podatki se zapisujejo kronološko, torej niso časovno pomešani med seboj.

Tako se je nabralo za točno eno leto podatkov. **Napiši program**, ki bo prebral te podatke in izpisal, koliko je bilo suhih dni — to so dnevi, v katerih ni zapadlo nič padavin. Poleg tega naj izpiše tudi, koliko je bilo največje število zaporednih suhih dni v tem letu. Tvoj program naj bere iz datoteke `meritve.txt` ali pa iz standardnega vhoda, kar ti je lažje; meritve naj bere vse do konca podatkov (EOF).

Primer podatkov:

```
20160101 0
20160101 0
20160101 12
20160101 30
02012016 10
02012016 0
02012016 120
12345 0
12345 0
12345 0
20160104 0
20160105 0
20160105 23
...
20161231 0
```

3. Virus

V računalniškem podjetju so naredili 1000 zgoščenk z nekim programom, namenjenim za prodajo. Žal pa je na eno izmed zgoščenk zašel tudi virus, ki ga želimo odkriti in tisto zgoščenko uničiti. Okuženo zgoščenko bi radi našli čim prej tako, da zgoščenke testiramo na enem ali več računalnikih. Na enem računalniku lahko poženemo poljubno mnogo zgoščenk. Če je na nekem računalniku med zagnanimi zgoščenkami bila tudi taka z virusom, se bo računalnik do naslednjega dne sesul. Takrat (torej naslednji dan) ga lahko ponovno usposobimo in ga uporabimo za nadaljnje poskuse.

Čas za zagon zgoščenk je zanemarljivo majhen (lahko ga odmisliš). Virus iz okuženega računalnika se ne bo razširil na druge zgoščenke.

Opiši postopek, kako na računalnikih poganjati zgoščenke, da ugotovimo, katera zgoščenka je okužena z virusom. Pravzaprav opiši dva postopka (vsak je vreden polovico točk pri tej nalogi) z naslednjimi omejitvami:

(*a*) Za eksperimentiranje imamo na voljo le en računalnik, zgoščenko pa bi radi našli v minimalnem številu dni.

(*b*) Zgoščenko moramo najti v enem dnevu, za eksperimentiranje pa želimo uporabiti čim manj računalnikov (po možnosti precej manj kot 1000).

4. Analiza enot

Pogosto se zgodi, da dijak pri pouku fizike na tablo napiše napačno formulo, npr. za hitrost: $v = s \cdot t$. Nato profesor prav tako pogosto vzklikne: „Pa, saj to se že od daleč vidi, da je narobe. Na levi strani so enote metri na sekundo, na desni pa imaš metre krat sekunde, seveda je narobe, saj se enote ne ujemajo!“ **Napiši program** ali podprogram (kar ti je lažje), ki prebere fizikalno formulo in preveri, ali se enote na obeh straneh enačaja ujemajo.

Formula je dana kot enakost dveh ulomkov, na primer

```
a b c d / x y z = h / i j k
```

kar ustreza formuli $\frac{abcd}{xyz} = \frac{h}{ijk}$. Pri tem male črke angleške abecede (od `a` do `z`) predstavljajo fizikalne količine. Imenovalci ulomkov niso nujno prisotni, če pa so, je na vsaki strani znaka `/` gotovo vsaj ena količina.

Poleg tega imaš na začetku podane tudi enote za vsako fizikalno količino, ki v formuli nastopa. Enote so podane v obliki ulomka, za katerega veljajo enaka pravila kot za ulomke v formuli. Na primer:

```
a : m / s s
```

Vse fizikalne količine bodo označene z malimi črkami angleške abecede, prav tako pa tudi njihove enote. Enote in količine imajo lahko enake črke. Tvoj program lahko bere podatke s standardnega vhoda ali pa iz datoteke `enote.txt` (kar ti je lažje). V prvi vrstici je število količin n, ki bodo uporabljene v enačbi. V naslednjih n vrsticah sledijo izražave teh količin v osnovnih enotah. Nato sledi enačba v obliki, kot je opisana zgoraj. Posamezni znaki v formulah in opisih količin so ločeni s po enim presledkom. Tvoj program naj izpiše samo „`Formula je pravilna.`“ ali „`Formula ni pravilna.`“, odvisno od tega, ali se enote ujemajo ali ne.

Primer vhoda:

```
3
h : m
g : m / s s
v : m / s
h = v v / g
```

Pripadajoči izhod:

```
Formula je pravilna.
```

Še en primer:

```
4
f : g m / s s
m : g
t : s
s : m
f / m = s / t
```

Pripadajoči izhod:

```
Formula ni pravilna.
```

5. Za žužke gre

Mirko je navdušen žužkofil in podpornik pravic otrok. Doma ima terarij z n žužki, oštevilčenimi s števili od 1 do n, toda ne pozna njihovega spola. Kljub temu pa trdi, da so gotovo vsi žužki heteroseksualni. To želi dokazati tako, da en mesec strmi v terarij in si beleži interakcije med žužki.

Natančno **opiši postopek**, ki sprejme seznam interakcij med žužki in pove, ali obstaja taka razporeditev spolov, da so bile vse interakcije heteroseksualne (torej med žužkoma različnih spolov). Tvoj postopek kot vhodne podatke dobi število žužkov n, število interakcij med njimi m in seznam teh m interakcij — vsaka interakcija je opisana z dvema številoma s_i in t_i, ki povesta, da sta v i-ti interakciji sodelovala žužka s številkama s_i in t_i.

V posamezni interakciji vedno sodelujeta natanko dva žužka in noben žužek ni nikoli v interakciji sam s sabo. Žužki so lahko samo dveh spolov in spola ne spreminjajo med mesecem opazovanja.

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*.`in` in izpisujejo svoje rezultate v *imenaloge*.`out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil mapo s svojim uporabniškim imenom ter navodili, ki jih pravkar prebiraš. Ko boš sedel pred računalnik, boš dobil nadaljnja navodila za prijavo v sistem.

Na vsakem računalniku imaš na voljo disk `D:\`, v katerem lahko kreiraš svoje datoteke in imenike. Programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java ali VB.NET, mi pa jih bomo preverili s 64-bitnimi prevajalniki FreePascal, GNUjevima `gcc` in `g++`, prevajalnikom za javo iz OpenJDK 1.8 in s prevajalnikom Mono 4.2 za C# in VB.NET. Za delo lahko uporabiš Lazarus (IDE za pascal), `gcc`/`g++` (GNU C/C++ — command line compiler), `javac` (za javo 1.8), Visual Studio, Eclipse in druga orodja.

Na spletni strani `http://rtk2016.fri1.uni-lj.si/` boš dobil nekaj testnih primerov.

Prek iste strani lahko oddaš tudi rešitve svojih nalog, tako da tja povlečeš datoteko z izvorno kodo svojega programa. Ime datoteke naj bo takšne oblike:

imenaloge.`pas`
imenaloge.`c`
imenaloge.`cpp`
ImeNaloge.`java`
ImeNaloge.`cs`
ImeNaloge.`vb`

Datoteka z izvorno kodo, ki jo oddajaš, ne sme biti daljša od 30 KB.

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih (praviloma desetih). Za vsak testni primer se bo izpisalo, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund ali pa porabil več kot 200 MB pomnilnika, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku, razen s predpisano vhodno in izhodno datoteko. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku), prenosnih računalnikov, prenosnih telefonov itd.

Preden oddaš kak program, ga najprej prevedi in testiraj na svojem računalniku, oddaj pa ga šele potem, ko se ti bo zdelo, da utegne pravilno rešiti vsaj kakšen testni primer.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih; pri prvi in tretji nalogi je po deset testnih primerov in pri vsakem od njih dobi program 10 točk, če je izpisal pravilen odgovor, sicer pa 0 točk; pri drugi, četrti in peti nalogi je po 20 testnih primerov in pri vsakem od njih dobi program 5 točk, če je izpisal pravilen odgovor, sicer pa 0 točk.

Nato se točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N-1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

Skupno število točk tekmovalca je vsota po vseh nalogah. Tekmovalce razvrstimo po skupnem številu točk.

Vsak tekmovalec se mora sam zase odločiti o tem, katerim nalogam bo posvetil svoj čas, v kakšnem vrstnem redu jih bo reševal in podobno. Verjetno je priporočljivo najprej reševati lažje naloge. Liste z nalogami lahko po tekmovanju obdržiš.

Poskusna naloga (ne šteje k tekmovanju) (`poskus.in`, `poskus.out`)

Napiši program, ki iz vhodne datoteke prebere dve celi števili (obe sta v prvi vrstici, ločeni z enim presledkom) in izpiše desetkratnik njune vsote v izhodno datoteko.

Primer vhodne datoteke:

```
123 456
```

Ustrezna izhodna datoteka:

```
5790
```

Primeri rešitev (dobiš jih tudi kot datoteke na `http://rtk2016.fri1.uni-lj.si/`):

- V pascalu:

```
program PoskusnaNaloga;
var T: text; i, j: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i, j); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * (i + j)); Close(T);
end. {PoskusnaNaloga}
```

- V C-ju:

```
#include <stdio.h>
int main()
{
  FILE *f = fopen("poskus.in", "rt");
  int i, j; fscanf(f, "%d %d", &i, &j); fclose(f);
  f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * (i + j));
  fclose(f); return 0;
}
```

- V C++:

```
#include <fstream>
using namespace std;
int main()
{
  ifstream ifs("poskus.in"); int i, j; ifs >> i >> j;
  ofstream ofs("poskus.out"); ofs << 10 * (i + j);
  return 0;
}
```

- V javi:

```
import java.io.*;
import java.util.Scanner;
public class Poskus
{
  public static void main(String[] args) throws IOException
  {
    Scanner fi = new Scanner(new File("poskus.in"));
    int i = fi.nextInt(); int j = fi.nextInt();
    PrintWriter fo = new PrintWriter("poskus.out");
    fo.println(10 * (i + j)); fo.close();
  }
}
```

- V C#:

```
using System.IO;
class Program
{
  static void Main(string[] args)
  {
    StreamReader fi = new StreamReader("poskus.in");
    string[] t = fi.ReadLine().Split(' '); fi.Close();
    int i = int.Parse(t[0]), j = int.Parse(t[1]);
    StreamWriter fo = new StreamWriter("poskus.out");
    fo.WriteLine("{0}", 10 * (i + j)); fo.Close();
  }
}
```

- V Visual Basic.NETu:

```
Imports System.IO

Module Poskus

  Sub Main()
    Dim fi As StreamReader = New StreamReader("poskus.in")
    Dim t As String() = fi.ReadLine().Split() : fi.Close()
    Dim i As Integer = Integer.Parse(t(0)), j As Integer = Integer.Parse(t(1))
    Dim fo As StreamWriter = New StreamWriter("poskus.out")
    fo.WriteLine("{0}", 10 * (i + j)) : fo.Close()
  End Sub

End Module
```

NALOGE ZA TRETJO SKUPINO

1. Letala (`letala.in`, `letala.out`)

Z letali bi radi prepeljali več zabojnikov iz kraja A v kraj B. Posamezno letalo lahko vsak dan odpelje največ en zabojnik iz A v B in se nato vrne nazaj v A. Imamo n zabojnikov in k letal. Za vsak zabojnik i poznamo njegovo maso m_i, za vsako letalo j pa poznamo njegovo nosilnost c_j (to pomeni, da lahko to letalo pelje le tiste zabojnike, katerih masa je manjša ali enaka c_j). V istem dnevu lahko pošljemo na pot več letal. **Napiši program**, ki izračuna najmanjše število dni, v katerih je mogoče v okviru teh omejitev prepeljati vse zabojnike iz A v B. Če pa jih sploh ni mogoče prepeljati, naj tvoj program izpiše -1.

Vhodna datoteka: v prvi vrstici sta dve celi števili, n in k, ločeni s presledkom. Zanju bo veljalo $1 \leq n \leq 100\,000$ in $1 \leq k \leq 100\,000$. V drugi vrstici je n celih števil, ločenih s po enim presledkom, ki podajajo mase zabojnikov. Za vsako maso velja $1 \leq m_i \leq 10^9$. V tretji vrstici je k celih števil, ločenih s po enim presledkom, ki podajajo nosilnosti letal. Za vsako nosilnost velja $1 \leq c_j \leq 10^9$.

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število dni, v katerih je mogoče prepeljati vse zabojnike iz kraja A v kraj B. Če sploh ni mogoče prepeljati vseh zabojnikov, izpiši -1.

Primer vhodne datoteke:

```
10 3
20 100 60 30 40 90 80 50 10 70
45 120 30
```

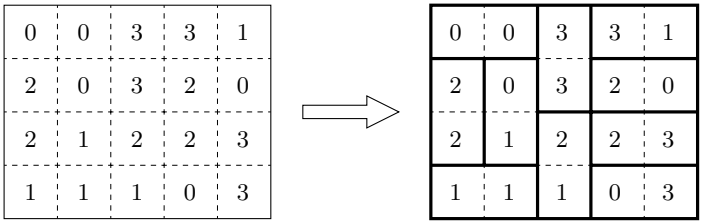
Pripadajoča izhodna datoteka:

```
6
```

2. Dominosa (`dominosa.in`, `dominosa.out`)

Pri igri Dominosa imamo pravokotno mrežo s sodim številom kvadratnih polj, v katerih so vpisana števila. Po dve sosednji polji lahko povežemo tako, da tvorita domino. Vsako domino lahko tvorimo vodoravno ali navpično. Naša naloga je, da iz vseh števil v mreži tvorimo domine tako, da uporabimo vsa polja (vsako natanko enkrat) in da nimamo dveh enakih domin (domini, ki vsebujeta isti dve številki).

Naslednja slika kaže primer mreže 5×4 polj, ki smo jo uspešno razdelili na same različne domine:



Če se omejimo na domine, ki imajo na posameznem polju od 0 do n pik, je največji možni pravokotnik, pri katerem je problem sploh še lahko rešljiv, velikosti $(n+2) \times (n+1)$. Pri naši nalogi bodo testni primeri vedno takšne maksimalne velikosti; pri

vsakem testnem primeru torej dobiš nek n in nato pravokotnik števil (od 0 do n) z $n+1$ vrsticami in $n+2$ stolpci.

Na primer, pri $n = 3$ je možnih 10 domin — od $[0\,|\,0]$ do $[3\,|\,3]$ — in igrali bi se na mreži velikosti 5×4; pri $n = 4$ je možnih 15 domin — od $[0\,|\,0]$ do $[4\,|\,4]$ — in igrali bi se na mreži velikosti 6×5; itd.

Vhodna datoteka: v prvi vrstici je celo število n (zanj velja $3 \le n \le 9$), ki pove maksimalno število pik na posameznem polju pri tem testnem primeru. Sledi $n+1$ vrstic, v vsaki od teh pa je $n+2$ števil (cela števila od 0 do n), ločenih s po enim presledkom. Te vrstice podajajo vsebino pravokotne mreže, ki jo moraš razdeliti na domine.

Izhodna datoteka: izpiši mrežo tako, da med vsaki dve sosednji vrstici oz. stolpca mreže vrineš še po eno vrstico oz. stolpec znakov „.“ (pika); kjer pa dve sosednji polji tvorita eno domino, izpiši med njiju znak „-“ (če sta v isti vrstici) ali „|“ (če sta v istem stolpcu).

Vsi testni primeri pri tej nalogi bodo izbrani tako, da je mrežo gotovo mogoče razdeliti na domine v skladu z zahtevami naloge. Če je možnih več rešitev (več različnih razdelitev mreže na domine), je vseeno, katero od njih izpišeš.

Primer vhodne datoteke:

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

Pripadajoča izhodna datoteka:

```
0-0.3.3-1
....|....
2.0.3.2-0
|.|......
2.1.2.2-3
....|....
1-1.1.0-3
```

3. Galaktična zavezništva (`xor.in`, `xor.out`)

V galaksiji je več stoletij vladal red. Galaktična republika je svojo vojsko že davno razpustila ter ga skozi čas uspešno vzdrževala na miren in diplomatski način. A razmere so se začele spreminjati, saj število pristašev temne strani strmo narašča. Senat republike je zato na izrednem zasedanju enoglasno odločil, naj se ponovno ustanovi enotna republikanska vojska, dovolj mogočna, da bo zatrla vsak morebiten poskus vzpona temne strani sile. Vsakemu izmed planetov, včlanjenih v republiko, je bilo določeno, naj zbere svoje najboljše vojaške stratege, ki se bodo na izboru na planetu Croissant potegovali za zasedbo elitnih položajev v novonastali vojaški hierarhiji. Položaji so oštevilčeni po pomembnosti od 1 do m, kjer je mesto 1 najpomembnejše.

Vsak izmed n planetov na izbor pošlje vojaške strateg, kjer je vsak izurjen za zasedbo natanko določenega položaja. Kandidate planeta a lahko tako opišemo z nizom m bitov, $a = a_1 a_2 \ldots a_m$, kjer je $a_i = 1$, če ima planet a za i-ti položaj izurjenega stratega, sicer pa je $a_i = 0$. Ker si planeti želijo imeti v vojski čim večji vpliv, so pripravljeni sklepati zavezništva.

V zavezništvo se planeti vselej povezujejo v trojicah. Predstavljajmo si tri planete, ki nameravajo skleniti zavezništvo, s pripadajočimi nizi kandidatov $a = a_1 \ldots a_m$, $b = b_1 \ldots b_m$ in $c = c_1 \ldots c_m$. Če imata na i-tem položaju natanko dva izmed planetov svoja kandidata, se ta dva spreta in odideta, tako da potem celotno zavezništvo na tem položaju nima nikogar. Če ima na i-tem položaju svojega

kandidata tudi tretji planet, le-ta zasede ravnokar izpraznjeno mesto. Položaje, na katerih ima zavezništvo svoje kandidate, lahko torej opišemo z binarnim nizom $d = d_1 \ldots d_m$, definiranim takole: bit d_i je prižgan ($d_i = 1$), če je izmed bitov a_i, b_i, c_i prižgan natanko eden ali pa vsi trije; če pa sta od treh bitov prižgana natanko dva ali nobeden, potem je bit d_i ugasnjen ($d_i = 0$). (Z drugimi besedami, niz d dobimo tako, da XORamo med sabo nize a, b in c).

Dve zavezništvi, recimo $d = d_1 d_2 \ldots d_m$ in $d' = d'_1 d'_2 \ldots d'_m$, lahko po moči primerjamo takole: poiščimo najpomembnejši položaj, pri katerem se ti dve zavezništvi razlikujeta (najmanjši i, pri katerem je $d_i \neq d'_i$). Tisto zavezništvo, ki ima na tem mestu prižgan bit, je močnejše od tistega, ki ima na tem mestu ugasnjen bit.

Napiši program, ki prebere podatke o več planetih in sestavi iz njih najmočnejše možno zavezništvo treh planetov.

Vhodna datoteka: v prvi vrstici sta dve celi števili, najprej n in nato m, ločeni s presledkom. Pri tem je n število planetov (zanj velja $3 \leq n \leq 7500$), m pa število položajev (zanj velja $1 \leq m \leq 50$). Sledi n vrstic, za vsak planet po ena. V vsaki od teh vrstic je m znakov `0` ali `1`; če je i-ti znak v neki vrstici enak `1`, to pomeni, da ima tisti planet na položaju i svojega človeka, sicer (torej i-ti znak v tej vrstici enak `0`) pa ga nima.

Izhodna datoteka: vanjo izpiši niz m ničel in enic, ki opisuje najmočnejše možno zavezništvo treh planetov, kar jih je mogoče sestaviti iz n planetov v vhodni datoteki.

Primer vhodne datoteke:

```
5 8
10101000
01001100
10101000
00101110
11000010
```

Pripadajoča izhodna datoteka:

```
11001010
```

Komentar: najmočnejše zavezništvo je pri tem primeru tisto, ki ga sestavljajo prvi, drugi in četrti planet.

4. Asteroidi (`asteroidi.in`, `asteroidi.out`)

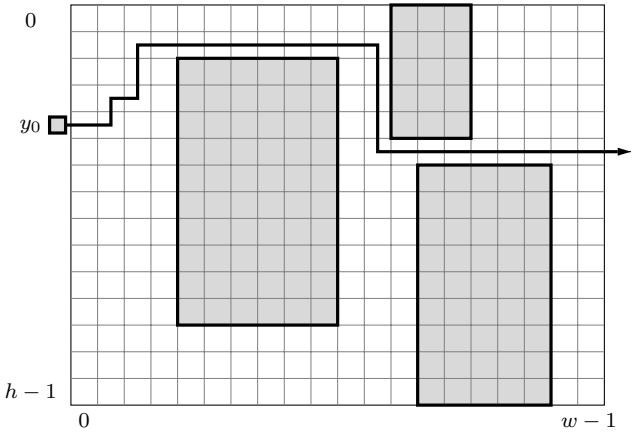
V dvodimenzionalni vesoljski simulaciji krmilimo majhno vesoljsko plovilo skozi polje n asteroidov. Medtem ko počasi plujemo skozi polje naprej vzdolž x-osi (v smeri nazaj se ni dovoljeno premikati), se s hitrimi premiki vzdolž y-osi izogibamo asteroidom. Omejeni smo tudi z zgornjim in spodnjim robom polja asteroidov, ki v simulaciji sovpada z zgornjim in spodnjim robom zaslona, v realnem svetu pa bi nas onkraj roba čakala meglica, v kateri bi se izgubili, ali pa bi nas pojedla vesoljska koza. Eden od posebnih izzivov je prečkati polje z minimalnim premikanjem vzdolž y-osi, to je z minimalno vsoto absolutnih vrednosti premikov vzdolž y-osi.

Napiši program, ki bo prebral dimenzije polja asteroidov, položaje in velikosti asteroidov ter izpisal najmanjšo vsoto premikov, s katero lahko prečkamo polje.

V našem poenostavljenem prikazu simulacije je polje asteroidov predstavljeno kot velika pravokotna karirasta mreža dimenzij $w \times h$ (širina w, višina h). V mrežo vpeljimo koordinatni sistem: vrstice oštevilčimo od 0 (najbolj zgornja vrstica) do $h-1$ (najbolj spodnja vrstica), stolpce pa od 0 (najbolj levi stolpec) do $w-1$ (najbolj

desni stolpec). Asteroide predstavljajo manjši pravokotniki, podani s koordinatami zgornjega levega in spodnjega desnega oglišča. Raketo si predstavljamo kot kvadrat velikosti 1×1. Začetni položaj rakete je $(-1, y_0)$ za neko podano koordinato (številko vrstice) y_0; od tam se raketa lahko premika po en kvadratek v desno, med temi premiki pa naredi še premike za poljubno število kvadratkov gor ali dol. Prvi premik mora vedno biti v desno, z $(-1, y_0)$ na $(0, y_0)$. Če se raketa po nekem premiku gor ali dol ne more premakniti naprej v smeri desno, ne da bi se zaletela v asteroid, je obtičala in je igre konec. Vse koordinate in dimenzije so nenegativna cela števila.

Naslednja slika kaže primer polja asteroidov (z $w = 20$, $h = 15$, $y_0 = 4$ in $n = 3$) in ene od možnih optimalnih poti rakete skozenj:



Vhodna datoteka: v prvi vrstici so podane višina h, širina w, začetna y-koordinata rakete y_0 (zanjo velja $0 \le y_0 \le h-1$) ter število asteroidov n. Veljalo bo $1 \le w \le 10^9$, $0 \le y_0 < h \le 10^9$ ter $1 \le n \le 300$. V 80 % primerov bo veljalo tudi $w \le 10^6$ in $h \le 10^5$. V 60 % primerov bo veljalo tudi $w \le 10^6$ in $h \le 10^3$. V 40 % primerov bo veljalo tudi $w \le 10^3$, $h \le 10^3$ ter $n \le 100$.

V naslednjih n vrsticah pa so podane koordinate zgornjega levega in spodnjega desnega kota i-tega asteroida $x_{1i}, y_{1i}, x_{2i}, y_{2i}$. Pri vsakem asteroidu i bo veljalo $0 \le x_{1i} \le x_{2i} < w$ in $0 \le y_{1i} \le y_{2i} < h$.

Asteroidi se ne prekrivajo, lahko pa se dotikajo.

Izhodna datoteka: poišči pot z najmanjšo skupno vsoto absolutnih vrednosti premikov v y-smeri in izpiši to vsoto v izhodno datoteko. Če nikakor ni možno priti skozi polje asteroidov, izpiši -1.

Primer vhodne datoteke:
(to je primer z gornje slike)

```
15 20 4 3
4 2 9 11
12 0 14 4
13 6 17 14
```

Pripadajoča izhodna datoteka:

```
7
```

5. Brisanje niza (`brisanje.in`, `brisanje.out`)

Dan je nek niz, ki ga sestavljajo same male črke angleške abecede. Iz njega smemo pobrisati enega ali več zaporednih znakov, vendar le, če so vsi enaki. Tako dobimo nek krajši niz, iz katerega lahko spet kaj pobrišemo in tako naprej. Prej ali slej lahko na ta način pridemo do praznega niza (iz katerega ne moremo pobrisati ničesar več).

Napiši program, ki prebere vhodni niz in ugotovi, kolikšno je najmanjše potrebno število brisanj, s katerimi lahko iz njega naredimo prazen niz.

Primer: če začnemo z nizom `aabbbacaa`, lahko na primer brišemo takole (na vsakem koraku je podčrtan tisti del niza, ki ga bomo naslednjega pobrisali):

$$\texttt{aabbb}\underline{\texttt{a}}\texttt{caa} \to \texttt{aa}\underline{\texttt{bbb}}\texttt{caa} \to \underline{\texttt{aa}}\texttt{caa} \to \underline{\texttt{c}}\texttt{aa} \to \underline{\texttt{aa}} \to \texttt{""}$$

Tu smo torej porabili 5 brisanj. Gre pa tudi s samo 3 brisanji:

$$\texttt{aabbba}\underline{\texttt{c}}\texttt{aa} \to \texttt{aa}\underline{\texttt{bbb}}\texttt{aaa} \to \underline{\texttt{aaaaa}} \to \texttt{""}$$

Vhodna datoteka: v prvi vrstici je celo število n, ki pove dolžino vhodnega niza pri tem testnem primeru. Veljalo bo $1 \leq n \leq 1000$. Pri 40 % testnih primerov bo veljalo tudi $n \leq 10$. V drugi vrstici je vhodni niz, ki bi ga radi pobrisali; dolg je n znakov, vsi ti znaki pa so male črke angleške abecede (od `a` do `z`).

Izhodna datoteka: vanjo izpiši eno samo celo število, in sicer najmanjše število brisanj, s katerimi je mogoče niz iz vhodne datoteke popolnoma pobrisati (tako, da iz njega nastane prazen niz).

Primer vhodne datoteke:	Pripadajoča izhodna datoteka:
`9` `aabbbacaa`	`3`

NALOGE ZA ŠOLSKO TEKMOVANJE

22. januarja 2016

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.

1. Nadležne besede

Neko besedilo hočemo natipkati s telefonsko tipkovnico, kjer se več črk tipka z isto tipko (črke *abc* so na tipki 2, *def* na tipki 3, *ghi* na tipki 4, *jkl* na tipki 5, *mno* na tipki 6, *pqrs* na tipki 7, *tuv* na tipki 8 in *wxyz* na tipki 9). Zato je nadležno, če se v besedi pojavita dve zaporedni črki, ki se tipkata z isto tipko. V nekaterih besedah se to zgodi celo po večkrat — na primer, v besedi *praprababica* kar šestkrat (*pr*, še enkrat *pr*, *ab*, *ba*, še enkrat *ab* in na koncu še *ca*).

Napiši program, ki prebere seznam besed in izpiše tisto, v kateri se največkrat zgodi, da se dve zaporedni črki tipkata z isto tipko. (Če je več takih besed, je vseeno, katero od njih izpišeš.) Vsaka beseda je v svoji vrstici in je dolga največ 100 znakov; vsi znaki so male črke angleške abecede. Tvoj program lahko bere s standardnega vhoda ali pa iz datoteke `besede.txt` (kar ti je lažje).

2. Prepisovanje

Na šoli za moderno umetnost je izvirnost najbolj cenjena vrlina. Učenci redno pišejo teste iz izvirnosti, ki potekajo tako, da n učencev posedejo v vrsto drug zraven drugega, jih oštevilčijo od 1 do n in jim naročijo, naj na list napišejo število. Toda jojmene, učitelj je opazil, da učenci prepisujejo, saj njihova števila niso izvirna, ampak so si števila učencev, ki sedijo skupaj, pogosto zelo podobna.

Učenec lahko prepisuje le od sošolca, ki je neposredno levo ali desno od njega (če ga ima). Učitelj ima določeno toleranco izvirnosti t: če se števili dveh sosednjih učencev razlikujeta za t ali manj, sta premalo izvirni in se za oba tadva učenca sumi, da sta prepisovala.

Napiši program, ki prebere n, t in seznam števil, ki so jih na testu napisali učenci, in izpiše število učencev, ki so osumljeni prepisovanja. Vhodne podatke lahko bereš s standardnega vhoda ali pa iz datoteke `prepisovanje.txt` (kar ti je lažje). V prvi vrstici vhoda sta število učencev n in učiteljeva toleranca t, ločeni s presledkom (veljalo bo $n \leq 1\,000\,000$ in $0 < t \leq 1\,000\,000\,000$). V drugi vrstici je n celih števil, ločenih s presledki; vsako od teh števil je med $-1\,000\,000\,000$ in $+1\,000\,000\,000$).

Primer vhoda:

```
7 2
1 4 2 6 -3 -5 -4
```

Pripadajoči izhod:

```
5
```

Komentar: tu imamo $n = 7$ učencev in toleranco $t = 2$. Števili 1 in 4 sta za 3 narazen, tako da tu ni suma prepisovanja. Števili 4 in 2 sta za 2 narazen, tako da sta drugi in tretji učenec osumljena. Števila 2 in 6 ter 6 in -3 so dovolj narazen, ni suma. Števila -3 in -5 ter -5 in -4 so preveč skupaj, tako da so peti, šesti in sedmi učenec osumljeni. Tako je vsega skupaj osumljenih prepisovanja pet učencev.

3. Riziko

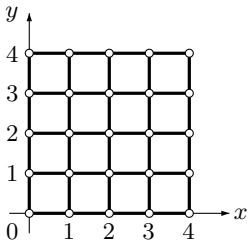
Dva igralca se igrata naslednjo igro. Najprej prvi igralec vrže tri kocke in jih uredi padajoče po številu pik; nato enako naredi še drugi igralec, le da ima ta samo dve kocki namesto treh. Nato primerjata prvo kocko prvega igralca in prvo kocko drugega igralca; tisti igralec, čigar kocka ima več pik, dobi dve točki (če imata oba enako število pik, dobi vsak po eno točko). Nato na enak način primerjata še drugo kocko prvega igralca in drugo kocko drugega igralca; spet dobi dve točki tisti, čigar kocka ima več pik (če pa imata oba enako število pik, dobi vsak po eno točko).

Napiši program, ki prebere pet celih števil od 1 od 6 — najprej tri kocke prvega igralca, nato dve kocki drugega igralca, vendar ne ene ne druge še niso nujno urejene padajoče — in izpiše, koliko točk dobi prvi in koliko drugi igralec.

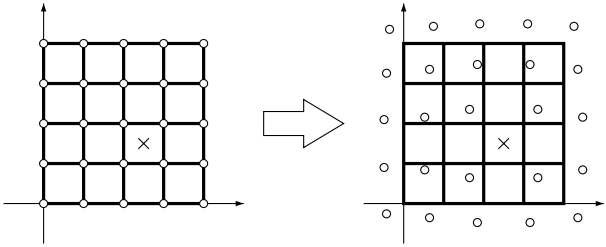
Primer: recimo, da prvi igralec vrže 1, 6, 2, drugi pa 4, 5. Po urejanju padajoče ima prvi igralec 6, 2, 1, drugi pa 5, 4. Najprej torej primerjata 6 in 5, pri čemer dobi dve točki prvi igralec; nato pa primerjata še 2 in 4, pri čemer dobi dve točki drugi igralec. Izid te igre je torej ta, da je vsak igralec dobil po dve točki.

4. Eksplozija

V koordinatni ravnini imamo kvadrat velikosti 4×4; njegov spodnji levi kot je v koordinatnem izhodišču. Ta kvadrat lahko v mislih razdelimo na 16 enotskih kvadratov velikosti 1×1. V vsako točko, ki je oglišče kakšnega od teh enotskih kvadratov, postavimo nek točkast predmet (teh predmetov je torej skupaj 25), kot kaže naslednja slika:



V središču enega od 4×4 enotskih kvadratov pride do eksplozije. Ta povzroči, da se vsak od naših 25 točkastih predmetov premakne za $1/2$ enote (z drugimi besedami, za polovico dolžine stranice enotskega kvadrata) stran od središča eksplozije (premakne se torej po poltraku, ki povezuje središče eksplozije s prvotnim položajem tistega predmeta). Primer kaže naslednja slika; križec $\times$ označuje središče eksplozije:



Opiši postopek, ki kot vhodne podatke dobi koordinate vseh 25 točkastih predmetov po eksploziji (vendar ne v kakšnem posebnem vrstnem redu — točke so lahko poljubno premešane) in izračuna, v središču katerega kvadrata je prišlo do eksplozije. **Dobro utemelji**, zakaj je tvoja rešitev pravilna.[1]

5. Barvanje plošče

Imamo okroglo ploščo, razdeljeno na n enako širokih izsekov. Izseki so oštevilčeni v smeri urinega kazalca od 0 do $n-1$. Na začetku so vsi izseki bele barve, mi pa bi radi nekatere pobarvali črno; dana je tabela pobarvaj, ki nam pove, katere izseke hočemo pobarvati črno (vrednost pobarvaj[k] je **true**, če je treba izsek k pobarvati črno, sicer pa je **false**). Plošča je vpeta v napravo, ki zna ploščo vrteti in ima na eni strani barvno glavo, s katero lahko pobarva po en izsek naenkrat. Plošča podpira tri ukaze:

- Levo: če je bil prej pod barvno glavo izsek številka k, je po tem premiku pod glavo izsek številka $k+1$ (razen če je bil $k = n-1$, takrat pa je po premiku pod glavo izsek številka 0);
- Desno: če je bil prej pod barvno glavo izsek številka k, je po tem premiku pod glavo izsek številka $k-1$ (razen če je bil $k = 0$, takrat pa je po premiku pod glavo izsek številka $n-1$);
- Pobarvaj: pobarva s črno barvo izsek, ki je trenutno pod barvno glavo. Nič ni narobe, če isti izsek pobarvaš večkrat, vendar od tega tudi ni nobene koristi. Izseka, ki je bil nekoč že pobarvan črno, kasneje ne moremo pobarvati nazaj na belo.

Ploščo bi radi pobarvali v skladu s zahtevami iz tabele pobarvaj in pri tem izvedli čim manj ukazov. Znan je tudi začetni položaj plošče (torej številka izseka, ki je na začetku pod barvno glavo — recimo ji z). Vseeno nam je, kako bo plošča zasukana na koncu našega postopka. **Opiši postopek** ali pa napiši program ali podprogram (kar ti je lažje), ki iz teh podatkov izračuna najmanjše število ukazov, ki jih potrebujemo.

[1] Iz te naloge lahko dobimo še več podobnih, malo težjih, če sprostimo nekatere omejitve pri vhodnih podatkih. Poskusi na primer rešiti nalogo še v naslednjih primerih: (a) Do eksplozije lahko pride na poljubni točki, ne nujno v središču enega od enotskih kvadratov. (b) Namesto 16 enotskih kvadratov imamo kvadrate velikosti $u \times u$ za neko konstanto u, ki je enaka pri vseh 16 kvadratih, vendar je mi ne dobimo kot vhodni podatek. (Pri eksploziji se vsak predmet premakne za razdaljo $u/2$.) (c) Doslej smo predpostavljali, da ima naša mreža 16 kvadratov svoj spodnji levi kot v koordinatnem izhodišču in da je lepo poravnana s koordinatnima osema (stranice naših kvadratov so bile vodoravne in navpične); reši nalogo tudi za primer, da je mreža zasukana za nek neznani kot α in premaknjena za nek neznani premik (x_0, y_0) stran od koordinatnega izhodišča.

NEUPORABLJENE NALOGE IZ LETA 2014

V tem razdelku je zbranih nekaj nalog, o katerih smo razpravljali na sestankih komisije pred 9. tekmovanjem ACM v znanju računalništva (leta 2014), pa jih potem na tistem tekmovanju nismo uporabili (ker se nam je nabralo več predlogov nalog, kot smo jih potrebovali za tekmovanje). Ker tudi te neuporabljene naloge niso nujno slabe, jih zdaj objavljamo v letošnjem biltenu, če bodo komu mogoče prišle prav za vajo. Dodali smo tudi težje različice nekaj nalog, ki smo jih v lažji obliki uporabili na tekmovanju 2015. Poudariti pa velja, da niti besedilo teh nalog niti njihove rešitve (ki so na str. 76–111) niso tako dodelane kot pri nalogah, ki jih zares uporabimo na tekmovanju. Razvrščene so približno od lažjih k težjim.

1. Ganttov diagram

Janez je imel program, ki je sprožil kup drugih pod-procesov, in končal, ko so se končali. Program je včasih končal hitro, včasih pa je trajal predolgo. Janez je ugotovil da je to zato, ker so pod-procesi včasih trajali različno dolgo.

Zato se je odločil, da bo napisal program, ki mu bo omogočil, da bo hitro videl, kateri pod-proces je povzročil težave, kadar je celota trajala predolgo.

Imel je vhodno datoteko, ki je v vsaki vrstici vsebovala tri podatke o pod-procesu:

- številko sekunde, ko je bil proces sprožen
- številko sekunde, ko je bil proces končan
- ime procesa (dolgo je največ 100 znakov).

Primer:

```
8 13 uprava
1 14 priprava
8 35 naprava
30 37 zelenkasto
```

Pomagaj mu in **napiši program**, ki prebere takšno vhodno datoteko in izpiše diagram naslednje oblike:

```
0--------10--------20--------30--------40--------50
|############### priprava      |         |         |
|       ###### uprava          |         |         |
|       ############################## naprava     |
|         |         |         ######## zelenkasto |
```

Program naj dolžino celotne vrstice prilagodi tako, da se bo pravilno prikazal pod-proces, ki se bo končal zadnji. Bolje bodo ocenjene tiste rešitve, ki bodo v spominu držale samo eno izhodno vrstico.

2. Cezar

Julij Cezar je v svojih sporočilih vojaškega pomena uporabljal zelo preprost način šifriranja besedila, pri katerem je vsako črko prvotnega besedila nadomestil z drugo

črko, ki leži v abecedi za n mest naprej (in na koncu naokrog), nečrkovni znaki pa se ne spremenijo. Če je zamik n denimo 3 v desno in bi uporabili le male črke angleške abecede, bi tako zamenjali:

$$\mathtt{a} \to \mathtt{d}, \mathtt{b} \to \mathtt{e}, \mathtt{c} \to \mathtt{f}, \ldots, \mathtt{w} \to \mathtt{z}, \mathtt{x} \to \mathtt{a}, \mathtt{y} \to \mathtt{b}, \mathtt{z} \to \mathtt{c}$$

in iz besede `krokodil` bi dobili `nurnrglo`.

Takega šifriranja ni težko razbiti, je pa uporabno, kadar ne bi želeli, da je neko besedilo čitljivo že na prvi pogled, hkrati pa ga ni težko dešifrirati, če bralec to želi. Tako na primer neki uganki lahko priložimo rešitev ali povzetku neke zgodbe njen zaključek (spojler).

Popularna različica Cezarjeve šifre je ROT13, pri katerem je $n = 13$, kar pri 26 črkah angleške abecede predstavlja ravno zamik za pol abecede, zato je preslikava simetrična. Zanimivo je, da se nekatere besede, zašifrirane po postopku ROT13, prevedejo v druge veljavne besede, tako na primer v slovenščini lahko najdemo naslednje pare besed, ki so ena drugi ROT13 preslikava:

```
bo   ob
ceni prav
cev  pri
ena  ran
gre  ter
iver vire
```

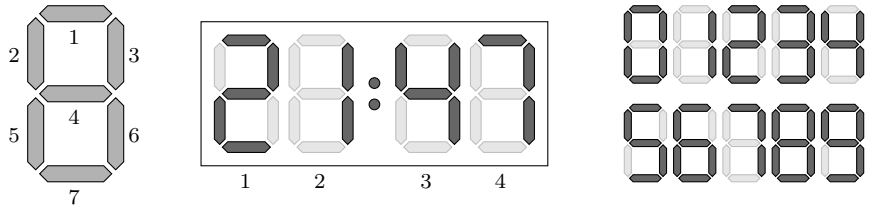
Na vhodni datoteki imamo seznam besed, v vsaki vrstici eno. Besede sestojijo le iz malih črk angleške abecede. **Napiši program**, ki bo z vhodne datoteke bral besede, za vsako od njih poiskal njeno ROT13 zašifrirano preslikavo in zanjo preveril, ali predstavlja neko drugo besedo s tega seznama besed. Program naj izpiše vse pare tako najdenih besed (v poljubnem vrstnem redu), tako kot v gornjem zgledu. Predpostavimo lahko, da v vhodni datoteki ni več kot 1000 besed, da se nobena ne pojavi po večkrat in da nobena ni daljša od 20 znakov.

3. Digitalna ura

Imamo digitalno uro, ki na svojem zaslonu prikazuje štiri števke, dve za uro in dve za minuto. Vsaka števka je sestavljena iz sedmih segmentov, vsak segment pa je lahko prižgan ali ugasnjen. Števe so od leve proti desni oštevilčene od 1 do 4, segmenti pa pri vsaki števki od 1 do 7, kot kaže slika na str. 33.

Za upravljanje prikaza na zaslonu je na voljo podprogram **void** Spremeni(**int** stevka, **int** segment), ki kot parametra dobi številko števke (od 1 do 4) in segmenta (od 1 do 7) ter spremeni stanje tega segmenta te števke (če je bil prižgan, ga ugasne, če pa je bil ugasnjen, ga prižge).

Napiši podprogram **void** Osvezi(**int** h, **int** m), ki ga bo sistem poklical na začetku vsake minute in mu kot parametra podal trenutni čas (v urah in minutah od polnoči). Tvoj podprogram naj predpostavi, da zaslon ure trenutno pravilno kaže čas prejšnje minute (torej zadnje minute pred tisto, ki se je ravnokar začela), in naj z ustreznimi klici funkcije Spremeni poskrbi, da bo zaslon pravilno prikazoval novi čas.



Ilustracija k nalogi „Digitalna ura“. Leva slika kaže, kako so segmenti oštevilčeni od 1 do 7; srednja slika kaže, kako so števke na zaslonu ure oštevilčene od 1 do 4; desna slika kaže, kateri segmenti so pri posamezni števki (od 0 do 9) prižgani, kateri pa ugasnjeni.

4. Funkciji

Dani sta naslednji dve funkciji:

```
int Prva(int a, int b)
{
  int c = 1;
  while (b > 0)
  {
    if (b & 1) c *= a;
    a *= a; b >>= 1;
  }
  return c;
}

int Druga(int a, int b)
{
  int c = 1, d = 1;
  while (d <= b) d <<= 1;
  while (d > 0)
  {
    c *= c;
    if (d & b) c *= a;
    d >>= 1;
  }
  return c;
}
```

Opiši, kaj in kako računata tidve funkciji. Pri tem predpostavi, da sta parametra a in b nenegativna in dovolj majhna, da pri računanju ne pride do prekoračitve obsega tipa **int**.[2]

5. 3-d tiskalnik

Predmet, ki ga natisnemo s 3-d tiskalnikom, je pravzaprav sestavljen iz velikega števila majhnih enako velikih kockic. Dana je trodimenzionalna tabela T, v kateri nam element T[x][y][z] pove, ali mora biti kockica s koordinatami (x, y, z) v našem predmetu prisotna ali ne.

Težava je, da tiskalnik tiska po plasteh od spodaj navzgor, torej od manjših z proti večjim, in ne bi bilo dobro, če bi nekatere kockice med tiskanjem „visele

[2] Malo drugačno nalogo na podobno temo smo na tekmovanju leta 2014 uporabili kot 4. nalogo v 2. skupini (glej bilten 2014, str. 19–20 in rešitev na str. 59–61).

v zraku“, ker jih s preostankom predmeta povezujejo le višje ležeče plasti, ki jih tiskalnik še ni natisnil.

Definirajmo: kockica (x, y, z) je *podprta*, če leži v najnižji plasti (torej če je $z = 0$) ali pa če je podprta njena spodnja soseda $(x, y, z - 1)$ ali pa ena od njenih štirih sosed v isti plasti $(x \pm 1, y, z)$ ali $(x, y \pm 1, z)$.

Napiši podprogram, ki za dano tabelo T preveri, ali so vse kockice v njej podprte.[3]

6. Kontrolne naloge

Bliža se zaključek ocenjevalnega obdobja in z njim kontrolne naloge. Dijaki so se tokrat organizirali in zbrali stare kotrolke pri vseh predmetih. Predmetov je p in pri i-tem od njih (za $i = 1, \ldots, p$) so zbrali k_i nalog. Ker so nekateri profesorji nekoliko leni, bi se lahko naloge na prihajajočih kontrolkah tudi ponovile. Dobri dijaki so sposobni rešiti n nalog na dan, pri čemer morajo biti vse naloge iz istega predmeta, slabši dijaki pa rešijo eno nalogo na dan. V razredu je d dobrih in s slabih dijakov.

Opiši postopek (ali napiši podprogram, če ti je lažje), ki izračuna najmanjše število dni, ki ga potrebujejo, da rešijo vse naloge, če si delo primerno razdelijo med seboj. Pri tem predpostavi, da so vse količine, omenjene v tej nalogi (p, d, s, t in $k_1, \ldots, k_p$), znane.[4]

7. Hišna številka

Tablica s hišno številko je pritrjena na steno z dvema žebljičkoma. Zgornji odpade in tablica se okrog spodnjega zavrti za 180 stopinj. Števke, ki so imele v prvotnem številu vrednost 6, so v tem novem položaju videti kot 9; bivše 9 so zdaj videti kot 6; števke 0, 1 in 8 so še zdaj videti kot 0, 1 ali 8; števke 2, 3, 4, 5 in 7 pa po tem obratu za 180 stopinj niso videti kot veljavne števke. Tako na primer iz števila 601 nastane 109; iz 123 ne nastane veljavno število; iz 96 pa nastane spet isto število, 96.

(a) **Opiši postopek**, ki za dano število n poišče najmanjšo tako hišno številko, ki je večja od n in ki se pri obratu za 180 stopinj ne spremeni. Predpostavi, da velja $1 \leq n \leq 10^{10^6}$.[5]

(b) Koliko je takih k-mestnih hišnih številk, ki se pri obratu za 180 stopinj ne spremenijo?

8. Lov na zaklad

V kleti si odkril zaprašen zemljevid, na katerem je označena pot do skrivnega zaklada. Kot pravi avanturist si se takoj lotil iskanja. Na zemljevidu piše, da se zaklad nahaja na tropskem otoku, poraščenem z džunglo in gostim rastlinjem. Kot pa da to ne bi bilo dovolj, se je ravno v času, ko si prišel na otok, začelo hudo monsunsko

[3] To je malo težja različica tretje naloge s šolskega tekmovanja 2015 (glej bilten 2015, str. 32 in rešitev na str. 78–9), kjer smo imeli namesto treh dimenzij le dve.

[4] To je težja različica četrte naloge s šolskega tekmovanja 2015; pri prvotni različici naloge je bilo podatno neko konkretno število dni t, naš postopek pa je moral preveriti, če je mogoče rešiti vse naloge v t dneh ali manj (glej bilten 2015, str. 33 in rešitev na str. 79–80).

[5] To je težja različica druge naloge s šolskega tekmovanja 2015 (glej bilten 2015, str. 31–2 in rešitev na str. 77–8); tam je bila omejitev $1 \leq n \leq 10^6$.

deževje z močnim vetrom z juga, zato je premikanje proti jugu veliko bolj naporno od, na primer, premikanja proti severu. Zemljevid pokriva le nek pravokoten del otoka in iz izkušenj veš, da je bolje, da ne hodiš v nepoznana območja, saj obstaja velika verjetnost, da se boš izgubil ali postal hrana domorodcem. **Napiši program**, ki ti poišče najhitrejšo pot od tvoje trenutnega položaja do zaklada.

Vhodna datoteka: v prvi vrstici se nahajata naravni števili h in w (veljalo bo $1 \leq h \leq 100$ in $1 \leq w \leq 100$), ki označujeta višino in širino otoka. Sledi h vrstic s po w znaki, ki podajajo zemljevid otoka; pri tem pika `.` označuje prehodno območje, znak `#` označuje neprehodno goščavo, znak `$` označuje zaklad in znak `*` označuje tvoj trenutni položaj. Premikaš se lahko samo na sosednje kvadratke, vzdolž štirih smeri neba, karkoli drugega bi lahko zmedlo svojo orientacijo in tvojo usodo prevedlo na prejšnji primer. Za premik na sosednji kvadratek v smeri vzhod–zahod potrebuješ 30 minut, za premik proti jugu 60 minut, za premik proti severu pa 20 minut.[6]

Izhodna datoteka: vanjo izpiši skupno število minut, ki jih potrebuješ, da najdeš zaklad.

Primer vhodne datoteke:

```
5 6
......
.#....
.#*...
.###..
$.....
```

Pripadajoča izhodna datoteka:

```
300
```

Komentar: da pridemo od začetnega položaja do zaklada, moramo iti okrog ene od stranic L-ja, ki ga tvorijo neprehodna polja. Če gremo okrog navpične stranice, pridemo do zaklada v 330 minutah, če pa gremo okrog vodoravne stranice, porabimo le 300 minut.

9. Minsko polje

Poveljnik te je zadolžil, da zavaruješ dostop do vojaške baze z minskim poljem. Baza se nahaja južno od minskega polja, edini dostop do nje pa je s severne strani preko travnika, ki je razdeljen na karirasto mrežo $w \times h$ celic. Poveljnik ti je že nekoliko pomagal s tem, da je predlagal seznam celic, ki naj vsebujejo mine. Na srečo si dovolj zgodaj opazil, da bi bila po predlaganem načrtu baza preveč dobro zavarovana — nihče ne bi mogel v bazo ali iz nje. Odločil si se, da boš sledil načrtu in postavljal mine eno za drugo, če pa bi trenutna mina onemogočila dostop do baze s severne strani minskega polja, jo boš enostavno ignoriral. V vsakem trenutku mora torej obstajati zaporedje premikov (levo, desno, gor, dol), ki te varno pripelje s severne na južno stran preko minskega polja. **Opiši postopek**, ki bo izpisal končno stanje min na polju.

[6] Zanimiva je tudi naslednja, malo težja različica naloge: kaj če moč vetra sčasoma narašča in so naši premiki zato vse počasnejši? Recimo bolj konkretno: če se ob času t minut od začetka sprehoda začnemo premikati iz nekega kvadratka v neki smeri, bo ta premik trajal $s \cdot (1 + c \cdot t)$ minut, pri čemer je c neka konstanta (ki je podana kot vhodni podatek), s pa je trajanje takega premika v prvotni različici naloge.

10. Speči agenti

Tajna služba je v neki tuji državi spletla omrežje spečih agentov. Agentov je n, vsak od njih pa pozna enega ali več drugih agentov (poznanstva so vedno obojestranska), kar lahko predstavimo z neusmerjenim grafom, v množica točk V pomeni agente, množica povezav E pa poznanstva med njimi. Vsak agent ima enega ali več ključev iz množice K. Skupina agentov $U \subseteq V$ je zmožna izvesti neko tajno operacijo, če je podgraf, ki ga tvorijo njihove točke, povezan (z drugimi besedami: če se da v grafu priti po povezavah od vsakega agenta iz U do vsakega drugega agenta iz U in se pri tem ves čas premikati le po agentih iz U) in če je unija njihovih ključev ravno enaka K (z drugimi besedami, če za vsak ključ iz K obstaja vsaj en agent iz U, ki ima ta ključ).

Lahko se zgodi, da bodo nekatere agente sovražniki razkrinkali in aretirali. Množico aretiranih agentov označimo z A. Oni seveda ne bodo mogli sodelovati v tajni operaciji, pa tudi tistih agentov, ki se neposredno poznajo s kakšnim od aretiranih, iz previdnosti raje ne bomo vključili v operacijo. **Opiši postopek**, ki za dane podatke o grafu, o ključih vsakega agenta in množico A določi, ali je množica vseh agentov, ki niso bili niti aretirani niti v neposrednem poznanstvu s kakšnim od aretiranih, zmožna izvesti tajno operacijo (po definiciji iz prejšnjega odstavka).

11. Načrtovanje tipkovnice

Obstaja več standardov, kako lahko tipke na številčni tipkovnici (na primer na telefonu) uporabimo za vnos črk. Ponavadi se na primer črke `abc` vnaša s tipko `2` (kar pomeni, da vnesemo črko `a` z enim pritiskom na tipko `2`, črko `b` z dvema in črko `c` s tremi), črke `def` s tipko `3` in tako naprej. Za vnos ene črke lahko torej potrebujemo tudi po več pritiskov na tipko. Dano je neko dolgo besedilo (vsi znaki v njem so male črke angleške abecede), ki bi ga radi natipkali s številčno tipkovnico. **Opiši postopek**, ki poišče takšno razporeditev črk na tipke (od `2` do `9`), pri kateri bomo za vnos tega besedila porabili najmanjše možno število pritiskov na tipke. V tem navodilu se pravzaprav skriva več podnalog, odvisno od tega, kakšne omejitve postavimo:

(a) Prvih nekaj črk (po abecednem vrstnem redu) naj bo razporejenih na tipko `2`, naslednjih nekaj na tipko `3` in tako naprej. Tudi znotraj vsake tipke morajo biti črke urejene po abecedi. Na posamezno tipko morata priti vsaj `2` črki in kvečjemu 4 črke.

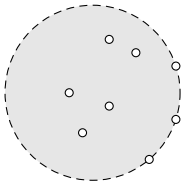
(b) Posamezna tipka lahko dobi poljubne črke v poljubnem vrstnem redu (na primer: lahko se odločimo, da bomo s tipko `2` tipkali črke `xae`, s tipko `b` črke `vhfk` itd.). Na posamezno tipko morata priti vsaj 2 črki in kvečjemu 4 črke.

(c) Kot (b), le da lahko na posamezno tipko pride poljubno število črk (lahko tudi nobena).

12. Oddajnik

Janez in njegova dva prijatelja so se odločili, da bodo začeli oddajati svoj lasten radijski signal. Kupili so že tri oddajnike, ki s pomočjo interference in drugih zelo zapletenih fizikalnih pojavov oddajajo signal tako, da je signal za poslušanje radia dovolj močan natanko znotraj kroga, ki ga ti trije oddajniki določajo (torej kroga, ki

gre skozi vse tri oddajnike). Signal se sliši dovolj dobro tudi v hišah, ki ležijo točno na robu kroga. V vasi, kjer živijo, bi sedaj radi postavili te tri oddajnike tako, da bi bila površina, ki jo pokrivajo, čim večja, pri tem pa nobena hiša v vasi ne sme ležati zunaj kroga. **Opiši postopek**, ki kot vhodne podatke dobi koordinate vseh hiš (predpostavimo, da so hiše točkaste), in izračuna, v katere tri hiše morajo Janez in njegova prijatelja postaviti oddajnike.



Primer: gornja slika kaže skupino hiš in največji primerni krog (torej največji tak krog, za katerega velja, da vsaj tri hiše ležijo na robu kroga in nobena hiša ne leži zunaj kroga).

13. Kompleksnost vezij

Pri tej nalogi se bomo ukvarjali s funkcijami, ki kot parametre dobijo n logičnih spremenljivk in tudi za rezultat vrnejo neko logično vrednost. Funkcije so torej oblike $\{0,1\}^n \to \{0,1\}$. Takšno funkcijo lahko računamo z *logičnim vezjem*; to je usmerjen acikličen graf, v katerem imamo n *vhodnih točk* (z oznakami $x_1, \ldots, x_n$), ostale točke pa so *logična vrata*; pri tem morajo imeti vhodne točke vhodno stopnjo 0. Logična vrata so treh vrst: IN ($\wedge$), ALI ($\vee$) in NE ($\neg$), pri čemer morajo imeti vrata NE vhodno stopnjo 1, vrata vrst IN in ALI pa vhodno stopnjo vsaj 2.

Vsaka točka u našega grafa predstavlja zdaj neko logično funkcijo $f_u(x_1, \ldots, x_n)$. Če je u vhodna točka z oznako x_i, definirajmo $f_u(x_1, \ldots, x_n) = x_i$; če je u točka tipa NE, v katero kaže povezava iz točke v, definirajmo $f_u(x_1, \ldots, x_n) = \neg f_v(x_1, \ldots, x_n)$; če je u točka tipa ALI, v katero kažejo povezave iz točk $v_1, \ldots, v_k$, definirajmo $f_u(x_1, \ldots, x_n) = \vee_{i=1}^{k} f_{v_i}(x_1, \ldots, x_n)$; če pa je u točka tipa IN, definiramo f_u enako kot pri ALI, le da uporabimo $\wedge$ namesto $\vee$. V vezju eno od točk razglasimo za *izhodno točko*; če je to recimo točka u, potem rečemo, da vezje *računa* funkcijo f_u.

Kompleksnost vezja definirajmo kot število povezav v njem. Ni se težko prepričati, da lahko za vsako funkcijo $f : \{0,1\}^n \to \{0,1\}$ najdemo veliko vezij, ki računajo prav to funkcijo. Definirajmo *kompleksnost funkcije* f kot minimum kompleksnosti vseh vezij, ki računajo to funkcijo.

(*a*) Koliko različnih funkcij $\{0,1\}^n \to \{0,1\}$ obstaja?

(*b*) **Opiši postopek**, ki kot parameter dobi funkcijo f (opisano na primer s tabelo) in sestavi poljubno (ne nujno najmanj kompleksno!) vezje, ki računa to funkcijo. Kakšna je kompleksnost dobljenega vezja?

V nadaljevanju te naloge se omejimo na $n = 3$, torej delamo s funkcijami treh spremenljivk.

(*c*) **Opiši postopek**, ki določi kompleksnost vseh funkcij treh spremenljivk. Katera funkcija (oz. funkcije) ima najvišjo kompleksnost? Nariši primer najpreprostejšega (najmanj kompleksnega) vezja za eno od teh funkcij.

(*d*) Ali se rezultati podnaloge (*c*) kaj spremenijo, če uvedemo dodatno omejitev, da smejo imeti vrata tipa ALI in IN po največ tri vhodne povezave?

(*e*) Kaj pa, če zahtevamo pri teh vratih natanko dve vhodni povezavi? Katerim funkcijam se kompleksnost najbolj poveča?

(*f*) Recimo, da kompleksnost vezja namesto s številom povezav definiramo s številom logičnih vrat v njem. Reši podnaloge (*c*), (*d*), (*e*) še za to definicijo kompleksnosti.

14. Osebni rekord

Športna ura z GPS sprejemnikom med tekom periodično zapisuje koordinate in čas. Naloga je iz takšnega dnevnika teka najti najkrajši čas, v katerem je tekač pretekel dano razdaljo. Na primer:

```
0 m, 0 s
0 m, 10 s
100 m, 40 s
300 m, 70 s
```

Torej je najprej 10 sekund stal na mestu, nato je prvih 100 m pretekel v 30 s, potem pa v nadaljnjih 30 s še 200 m. Recimo, da nas zanima čas za najhitrejših 100 m. Izračunamo lahko, da je za najhitrejših 100 m porabil 15 s ali manj; več kot to pa iz razpoložljivih podatkov ne moremo ugotoviti.

(Posebej poudarimo, da lahko določimo le *zgornjo mejo* za čas najhitreje pretečenega 100-metrskega intervala. O spodnji meji ne moremo reči ničesar pametnega. To je posledica dejstva, da za zadnjih 200 metrov teka vemo le, da je tekač zanje porabil 30 sekund; mogoče je od tega pretekel 100 metrov zelo hitro, drugih 100 pa zelo počasi in se je skupaj nabralo 30 sekund.)

V splošnem recimo, da imamo $n+1$ meritev, (D_i, T_i) za $i = 0, 1, \ldots, n$. Vsak tak par nam pove, da bil tekač ob času T_i na razdalji D_i od štarta. Za te podatke velja $0 = T_0 < T_1 < \ldots < T_{n-1} < T_n = T$ in $0 = D_0 < D_1 < \ldots < D_{n-1} \leq D_n = D$, pri čemer je D dolžina celotnega teka, T pa njegovo trajanje. **Opiši postopek**, ki za dano zaporedje meritev in za dano število d ugotovi najnižjo zgornjo mejo za čas, v katerem je bil pretečen najhitrejši d-metrski podinterval našega teka.[7]

Ker je naloga precej težka, jo lahko razdelimo na nekaj podvprašanj, od lažjih proti težjim:

(*a*) Reši nalogo za primere, ko je $n = 1$ in je D večkratnik d-ja (D in d smeta biti sicer poljubni realni števili, vendar pa je količnik D/d celo število).

(*b*) Reši nalogo za poljuben n, vendar z omejitvijo, da so vsi D_i-ji večkratniki d-ja.

(*c*) Reši nalogo za $n = 1$, $D = 250$, $t = 50$, $d = 100$.

[7] Da se izognemo dvoumnostim, napišimo nalogo še malo bolj formalno. Rekli bomo, da funkcija $P : [0, T] \to [0, D]$ predstavlja možen *potek* teka, če je naraščajoča (ne nujno strogo naraščajoča — tekač lahko stoji pri miru) in zvezna (tekač se ne more premikati z neskončno hitrostjo). Ta funkcija nam torej za vsak čas pove tekačev položaj (razdaljo od štarta). Potek P je *skladen* z vhodnimi podatki, če za vsak i od 0 do n velja $P(T_i) = D_i$. Čas, v katerem je tekač pretekel območje od x do x', označimo s $P[x, x'] := \min\{t : P(t) = x'\} - \max\{t : P(t) = x\}$. Čas najhitreje pretečenega d-metrskega intervala je potem $f(P) := \inf\{P[x, x + d] : 0 \leq x \leq D - d\}$. Naloga sprašuje po $f^\star := \sup\{f(P) : P \text{ je skladen}\}$.

(*d*) Reši nalogo za $n = 1$ in brez omejitve, da je D večkratnik d-ja.

(*e*) Reši nalogo za $n = 2$, $D_1 = 19$, $D_2 = 38$, $T_1 = 6$, $T_2 = 12$, $d = 10$.

(*f*) Reši nalogo v splošnem, torej za poljuben n in brez omejitve, da so D_i-ji večkratniki d-ja.

REŠITVE NALOG ZA PRVO SKUPINO

1. Tipkanje

Ko beremo zaporedje besed, je poleg trenutne besede koristno hraniti še prejšnjo besedo in njeno dolžino. Ko preberemo novo besedo, primerjajmo istoležne znake prejšnje in trenutne besede, dokler ne opazimo prvega neujemanja (ali pa pridemo do konca kakšne od besed). Recimo, da je bila prejšnja beseda dolga p znakov, trenutna je dolga d znakov, ujemata pa se v prvih u znakih. To pomeni, da bi moral naš zapisnikar po izpisu prejšnje besede pobrisati zadnjih $p - u$ znakov iz vnosnega polja (s tipko Backspace), nato natipkati zadnjih $d - u$ znakov nove besede in nato pritisniti Enter. Število pritiskov na tipke se torej poveča za $(p - u) + (d - u) + 1$. Ta postopek ponavljamo v zanki, dokler ne obdelamo vseh n besed.

Spodnji program v C-ju hrani obe besedi (prejšnjo in trenutno) v spremenljivki tabela; vsak od kazalcev beseda in prejBeseda kaže na eno vrstico tabele, pred branjem nove besede pa kazalca zamenjamo, tako da prejBeseda kaže na tisto besedo, na katero je prej kazala beseda (ker tista beseda ni več trenutna, ampak je zdaj prejšnja), ta pa kaže na vrstico, na katero je prej kazala prejBeseda; tam je zdaj že predprejšnja beseda, ki jo lahko povozimo z novo besedo (ki jo bomo vsak hip prebrali). Tako si prihranimo nekaj časa, ker besed ni treba kopirati iz ene tabele v drugo.

```
#include <stdio.h>
#define MaxDolz 100

int main()
{
  char tabela[2][MaxDolz + 1], *beseda = tabela[0], *prejBeseda = tabela[1], *t;
  int n, dolz, prejDolz, ujemanje, rezultat = 0;
  scanf("%d\n", &n); /* Preberimo število besed. */
  *beseda = 0; dolz = 0; /* Na začetku je vnosno polje prazno. */
  while (n-- > 0)
  {
    /* Prejšnjo besedo si zapomnimo v prejBeseda, njeno dolžino pa v prejDolz. */
    t = beseda; beseda = prejBeseda; prejBeseda = t; prejDolz = dolz;
    /* Preberimo novo besedo. */
    scanf("%s\n", beseda);
    /* Preštejmo, v koliko znakih se ujema s prejšnjo. */
    ujemanje = 0;
    while (beseda[ujemanje] && beseda[ujemanje] == prejBeseda[ujemanje]) ujemanje++;
    /* Poglejmo, kako dolga je ta beseda. */
    dolz = ujemanje; while (beseda[dolz]) dolz++;
    /* Po izpisu prejšnje besede moramo torej (prejDolz − ujemanje)-krat
       pritisniti Backspace, natipkati zadnjih (dolz − ujemanje) znakov */
       nove besede in nato še pritisniti Enter. */
    rezultat += (prejDolz − ujemanje) + (dolz − ujemanje) + 1;
  }
  /* Izpišimo rezultat. */
  printf("%d\n", rezultat); return 0;
}
```

2. Zoom

Niz s, ki opisuje potek zoomiranja, pregledujmo v zanki po znakih in vsakič ustrezno popravimo koordinate opazovanega območja. Izračunajmo mejo med levo in desno polovico, $x_m = (x_1 + x_2)/2$; in mejo med zgornjo in spodnjo polovico, $y_m = (y_1 + y_2)/2$. Če se moramo premakniti v eno od levih dveh četrtin (0 in 2), se desni rob našega območja premakne z x_2 na x_m, sicer (če se premikamo v eno od desnih dveh četrtin, to sta 1 in 3)) pa se levi rob premakne z x_1 na x_m. Podobno je tudi pri y-koordinatah. Odvisno od trenutnega znaka niza s moramo torej popraviti eno od koordinat x_1, x_2 ter eno od koordinat y_1, y_2. Ko pridemo do konca niza s, moramo dobljene koordinate le še izpisati.

```
#include <stdio.h>
void Zoom(char *s, double x1, double y1, double x2, double y2)
{
  double xm, ym;
  for ( ; *s; s++)
  {
    xm = (x1 + x2) / 2; ym = (y1 + y2) / 2;
    if (*s == '0' || *s == '1') y1 = ym; else y2 = ym;
    if (*s == '0' || *s == '2') x2 = xm; else x1 = xm;
  }
  printf("[%g, %g] x [%g, %g]\n", x1, x2, y1, y2);
}
```

3. Zaklepajski izrazi

Vhodni niz je koristno brati od konca proti začetku, pri tem pa vzdrževati seznam oz. sklad s podatki o tem, kateri oklepaji (kakšnih oblik) so trenutno odprti (torej da smo prebrali tisti zaklepaj, nismo pa še ustvarili pripadajočega oklepaja zanj). Ko pridemo do zvezdice, poglejmo na vrh sklada; oklepaj, ki ga bomo naredili iz te zvezdice, se mora ujemati z zaklepajem na vrhu sklada, sicer naš niz ne bo pravilno gnezden. Tako torej vemo, kakšne vrste oklepaj narediti iz trenutne zvezdice, zaklepaj z vrha sklada pa lahko pobrišemo, saj smo ga zdaj zaprli z oklepajem. Če se kdaj zgodi, da pridemo do zvezdice, sklad pa je prazen, lahko takoj odnehamo in vemo, da se niza ne da predelati v veljaven oklepajski izraz. Ko smo pregledali že cel niz od konca proti začetku, moramo le še preveriti, če je sklad takrat prazen; če ni, to pomeni, da je v vhodnem nizu preveč zaklepajev in ga tudi ne moremo predelati v veljaven oklepajski izraz.

Spodnji podprogram hrani sklad v tabeli sklad, število znakov na njem pa v spremenljivki sp.

```
void Dopolni(char *s)
{
  int n = strlen(s), i, sp = 0;
  char *sklad = (char *) malloc(n);
  for (i = n - 1; i >= 0; i--)
  {
    /* Če je trenutni znak zaklepaj, ga dodamo na sklad. */
    if (s[i] != '*') { sklad[sp++] = s[i]; continue; }
    /* Sicer imamo zvezdico. Če je sklad prazen, je niz neveljaven. */
```

```
    if (sp == 0) break;
    /* Pobrišimo zaklepaj z vrha sklada in spremenimo trenutno zvezdico
       v pripadajoči oklepaj. */
    sp--;
    if (sklad[sp] == ')') s[i] = '(';
    else if (sklad[sp] == ']') s[i] = '[';
    else if (sklad[sp] == '}') s[i] = '{';
    else if (sklad[sp] == '>') s[i] = '<';
  }
  if (i < 0 && sp == 0) printf("%s\n", s);
  else printf("Problem je nerešljiv.\n");
  free(sklad);
}
```

Gre pa tudi brez ločene tabele s skladom. Vsi znaki našega sklada so v resnici zaklepaji, ki smo jih dobili iz že pregledanega dela vhodnega niza `s`. Dovolj je, če si zapomnimo indeks, na katerem se v nizu `s` nahaja zaklepaj, ki je trenutno na vrhu sklada; spodnji podprogram ga hrani v spremenljivki `z`. Poleg tega pa si v spremenljivki `d` zapomni, koliko elementov je na skladu oz. z drugimi besedami: kako globoko je v nizu `s` vgnezden zaklepaj na indeksu `z`. Ko pobrišemo element s sklada, se moramo le zapeljati z `z`-jem naprej po nizu `s`, dokler ne naletimo na prvi tak zaklepaj, ki je vgnezden za en nivo plitveje.

```
void Dopolni2(char *s)
{
  int i, z, n = 0, d, dd; while (s[n]) n++;
  z = 0; d = 0; /* z = trenutni zaklepaj na vrhu sklada; d = globina sklada. */
  for (i = n - 1; i >= 0; i--)
  {
    /* Če je trenutni znak zaklepaj, ga dodamo na sklad. */
    if (s[i] != '*') { z = i; d++; continue; }
    /* Sicer imamo zvezdico. Če je sklad prazen, je niz neveljaven. */
    if (d == 0) break;
    /* Popravimo trenutni znak na oklepaj, ki se ujema z zaklepajem z vrha sklada. */
    if (s[z] == ')') s[i] = '(';
    else if (s[z] == ']') s[i] = '[';
    else if (s[z] == '}') s[i] = '{';
    else if (s[z] == '>') s[i] = '<';
    /* Pobrišimo zaklepaj z vrha sklada. */
    z++; d--; dd = d;
    for ( ; z < n; z++)
      if (s[z] == '(' || s[z] == '[' || s[z] == '{' || s[z] == '<') d++;
      else if (d == dd) break;
      else d--;
  }
  if (i < 0 && d == 0) printf("%s\n", s);
  else printf("Problem je nerešljiv.\n");
}
```

Ta rešitev porabi le $O(1)$ dodatnega pomnilnika (poleg niza `s`), ima pa to slabost, da v najslabšem primeru porabi $O(n^2)$ časa, če se mora v notranji zanki pogosto premikati zelo daleč naprej po nizu, preden doseže prvi naslednji zaklepaj, ki je

vgnezden za en nivo plitveje. Do te težave pride na primer pri vhodnih nizih oblike `**)*)*)*)*)*)*)*)*))`.

4. Izštevanka

Med simulacijo izštevanja bomo v tabeli ziv hranili za vsakega otroka podatek o tem, koliko življenj še ima. Na začetku postavimo vse elemente tabele na 2, ko pa pri izštevanju pridemo do nekega otroka, mu število življenj zmanjšamo za 1. Če pri tem pade na 0, se ustavimo in tega otroka izpišemo.

Spodnji program ima otroke oštevilčene od 0 do $n-1$ namesto od 1 do n. Tako lahko njihove številke neposredno uporabimo kot indekse v tabelo ziv. Z njimi je tudi zelo lahko računati: če smo bili pri otroku i in se pomaknemo za k mest naprej, pridemo do otroka $i+k$. Ker otroci stojijo v krogu, številka n predstavlja spet otroka 0, številka $n+1$ otroka 1 in tako naprej; z drugimi besedami, po vsakem koraku izštevanja moramo od številke otroka obdržati le ostanek po deljenju z n. Na koncu moramo pri izpisu paziti še na to, da številki prištejemo 1, tako da bomo imeli izpis v razponu od 1 do n (kot zahteva naloga) namesto od 0 do $n-1$.

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

int main()
{
  int n, k, i, *ziv;
  scanf("%d %d", &n, &k);    /* Preberimo vhodne podatke. */
  /* Pripravimo si tabelo, v kateri bomo za vsakega otroka
     hranili število življenj. Na začetku ima vsak 2 življenji. */
  ziv = (int *) malloc(n * sizeof(int));
  for (i = 0; i < n; i++) ziv[i] = 2;
  /* Odsimulirajmo potek izštevanja. Namesto od 1 do n imamo
     otroke oštevilčene od 0 do n - 1. */
  i = 0;
  while (ziv[i] > 0)
  {
    i = (i + k) % n;          /* Naredimo n korakov naprej od trenutnega otroka. */
    --ziv[i];                 /* Zmanjšajmo mu število življenj za 1. */
    k++;                      /* Povečajmo dolžino izštevanke. */
  }
  printf("%d\n", i + 1);      /* Izpišimo, pri katerem otroku smo končali. */
  free(ziv); return 0;        /* Pospravimo za sabo. */
}
```

5. H-indeks

Preprosta rešitev je, da v zanki gledamo vse večje h-je in za vsakega preverimo, ali obstaja vsaj h člankov, od katerih ima vsak po vsaj h citatov. Za to preverjanje potrebujemo še eno vgnezdeno zanko, ki gre po vseh člankih in šteje, koliko jih ima vsaj h citatov:

```
int HIndeks1(int seznam[], int n)
{
  int i, k, h;
```

```
  for (h = 1; ; h++)
  {
    /* Na tem mestu vemo, da obstaja vsaj h − 1 člankov s po vsaj h − 1 citati.
       Preverimo, ali obstaja vsaj h člankov s po vsaj h citati. */
    for (i = 0, k = 0; i < n; i++)
      if (seznam[i] >= h) k++;
    /* Zdaj vemo, da obstaja k člankov s po vsaj h citati. Če jih je manj kot h,
       potem h že ni več primeren h-indeks, torej moramo vrniti h − 1. */
    if (k < h) return h − 1;
  }
}
```

Zanka se gotovo prej ali slej konča, saj k ne more biti večji od dolžine seznama, torej n, tako da bo pogoj **if** (k < h) najkasneje pri $h = n + 1$ gotovo izpolnjen. Časovna zahtevnost tega postopka je v najslabšem primeru $O(n^2)$ — zunanja zanka izvede največ $n + 1$ iteracij, notranja pa pri vsaki od njih po n iteracij. Približno tako rešitev smo na tekmovanju tudi pričakovali od tekmovalcev prve skupine.

Bolj učinkovito pa lahko nalogo rešimo takole. Če ima nekdo n člankov, njegov h-indeks ne more biti več kot n, zato lahko članke, ki imajo več kot n citatov, obravnavamo tako, kot da bi imeli natanko n citatov (saj „odvečni“ citati ne morejo povečati h-indeksa, ker ima avtor premalo člankov).

Zdaj za vsako možno število citatov h od 0 do n preštejmo, koliko člankov ima natanko h citatov. To lahko za vse h naredimo z enim samim prehodom čez vhodni seznam. Števce člankov hranimo v tabeli f (z indeksi od 0 do n); na začetku postavimo vse elemente na 0, nato pa se sprehodimo po seznamu člankov in ko vidimo članek s h citati, povečamo $f[h]$ za 1.

Ko imamo tabelo f pripravljeno, gremo lahko po njej z indeksom h od konca proti začetku in računamo delne vsote $s[h] = f[h] + f[h+1] + \ldots + f[n]$. Taka delna vsota nam pove, koliko je člankov z vsaj h citati; če je takih člankov vsaj h (torej: če je $s[h] \geq h$), potem je ta h primeren kandidat za h-indeks. Ker potrebujemo največji tak h in ker pregledujemo h-je od višjih proti nižjim, se ustavimo takoj, ko najdemo prvi primerni h, saj je ta gotovo tudi najvišji primerni h sploh.

Ta postopek ima časovno zahtevnost le $O(n)$, saj potrebuje samo en sprehod po vhodnem seznamu in dva po tabeli f (prvič, ko jo inicializiramo na 0, in drugič, ko računamo delne vsote). Zapišimo ga še v C-ju:

```
#include <stdlib.h>

int HIndeks2(int seznam[], int n)
{
  int i, h, s, *f = (int *) malloc((n + 1) * sizeof(int));

  /* Postavimo vse elemente tabele f na 0. */
  for (h = 0; h <= n; h++) f[h] = 0;

  /* Za vsako število citatov preštejmo, koliko člankov ima toliko citatov.
     Članke z več kot n citati štejmo, kot da imajo le n citatov. */
  for (i = 0; i < n; i++) {
    h = seznam[i]; if (h > n) h = n;
    f[h]++; }

  /* Računajmo delne vsote od višjih h proti nižjim,
     dokler delna vsota ne doseže (ali preseže) h. */
  s = 0; h = n + 1;
```

```
  while (h > s)
    /* Premaknimo se s h za 1 navzdol in dodajmo f[h] k delni vsoti s. */
    s += f[--h];
  free(f);
  return h;
}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Sorodstvo

Za začetek je koristno zapise urediti naraščajoče po letu rojstva, saj iz omejitev v nalogi sledi, da je lahko oseba potencialni otrok le tistih oseb, ki so bile rojene pred njo. Recimo, da v tem vrstnem redu zapise oštevilčimo od 1 do n. Naj bo a_i dolžina najdaljše take verige, ki se začne z osebo i. Ena možnost je, da se veriga s to osebo tudi konča, tedaj je njena dolžina 1. Lahko pa se veriga nadaljuje pri neki osebi j (kar je sicer mogoče le, če velja $r_i + d \leq r_j < s_i$); ker je najdaljša veriga z začetkom pri osebi j dolga a_j, bo zdaj naša veriga z začetkom pri i dolga $a_j + 1$. Med vsemi temi možnostmi vzemimo za a_i tisto, ki je največja (torej ki dá najdaljšo verigo). Lahko si tudi zapomnimo (recimo kot n_i), pri katerem j smo ta maksimum dosegli; s pomočjo teh podatkov bomo kasneje lahko najdaljšo verigo tudi izpisali.

Vidimo lahko, da so nam pri izračunu a_i prišle prav vrednosti a_j za tiste osebe j, ki so rojene kasneje kot i (vsaj d let kasneje). Zato bomo te dolžine verig računali od konca zaporedja proti začetku, torej od kasneje rojenih oseb proti zgodnje rojenim. Tako bomo vedno imeli pri roki vse a_j, ki jih potrebujemo pri izračunu nekega a_i.

Zapišimo dobljeni postopek s psevdokodo:

```
uredi osebe po letu rojstva in jih v tem vrstnem redu oštevilči od 1 do n;
for i := n downto 1:
  a_i := 1; n_i := −1;
  for j := i + 1 to n:
    if r_j < r_i + d then continue;
    if r_j > s_i then break;
    if a_j + 1 > a_i then a_i := a_j + 1,   n_i := j;
```

Nato moramo poiskati tisti i, pri katerem smo dosegli največjo vrednost a_i. Njegovo verigo zdaj lahko izpišemo takole:

```
while i ≠ −1:
  izpiši i;
  i := n_i;
```

Ta rešitev ima časovno zahtevnost $O(n^2)$ zaradi vgnezdenih zank po i in j. Dalo bi se jo še izboljšati, če bi nad tabelo $a = (a_1, \ldots, a_n)$ gradili drevesasto strukturo z maksimumi po dveh, štirih, osmih itd. zaporednih elementov. Interval j-jev, ki nas pri nekem i zanima (torej tistih, ki ustrezajo pogoju $r_i + d \leq r_j \leq s_i$), lahko poiščemo z bisekcijo v času $O(\log n)$, nato pa z omenjenim drevesom maksimumov v $O(\log n)$ časa tudi poiščemo maksimum vrednosti a_j po vseh teh j. Skupaj z urejanjem na začetku postopka bomo tako imeli časovno zahtevnost le še $O(n \log n)$.

2. Suhi dnevi

Vhodne podatke bomo brali vrstico po vrstico. V spremenljivki nVseh bomo šteli vse suhe dni, v spremenljivki maxSkupina hranimo dolžino najdaljše skupine suhih dni doslej, v spremenljivki trenSkupina pa dolžino strnjene skupine suhih dni, v kateri se trenutno nahajamo (če trenutni dan ni suh, je trenSkupina = 0).

Podatek o tem, ali je trenutni dan suh ali ne, hranimo v spremenljivki suh. Pri prvi meritvi tistega dneva jo postavimo na **true** ali **false** odvisno od tega, ali je meritev enaka 0 ali ne, odtlej pa pri vsaki nadaljnji meritvi za ta dan preverimo, če je različna od 0, in če je, postavimo suh na **false**.

To, ali je dan res suh, dokončno vemo šele takrat, ko preberemo vse njegove meritve. Ko torej pridemo do konca meritev tistega dne, lahko (če je bil dan res suh) povečamo števec nVseh in dolžino trenutne skupine suhih dni trenSkupina; in če je ta skupina zdaj najdaljša doslej, si jo zapomnimo v maxSkupina. Če pa trenutni dan ni bil suh, pustimo nVseh pri miru in postavimo dolžino trenutne skupine suhih dni na 0.

Kako vemo, kdaj smo prebrali vse meritve dosedanjega dneva? To v resnici vemo šele takrat, ko naletimo na prvo meritev naslednjega dneva ali pa na konec vhodnih podatkov (EOF). To pomeni, da moramo primerjati številko dneva pri trenutni meritvi s tisto pri prejšnji meritvi; če sta različni, pomeni, da je konec meritev tistega prejšnjega dneva. Pri tem moramo paziti še na dva robna primera: na začetku vhodne datoteke sploh še nimamo prejšnje meritve (da zaznamo ta primer, si pomagamo s spremenljivko prvi), na koncu vhodne datoteke pa nimamo trenutne meritve (ta primer pokrijemo s pomočjo spremenljivke ok, ki pove, ali je branje trenutne meritve uspelo ali ne).

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

int main()
{
  int maxSkupina = 0, trenSkupina = 0, nVseh = 0, dan, prejDan, meritev;
  bool suh, ok, prvi = true;

  do
  {
    /* Poskusimo prebrati naslednjo meritev. */
    ok = (2 == scanf("%d %d", &dan, &meritev));
    if (ok && ! prvi && dan == prejDan) {
      /* Nadaljuje se isti dan kot pri prejšnji meritvi.
         Če je meritev neničelna, si zapomnimo, da dan ni suh. */
      if (meritev != 0) suh = false; }
    else
    {
      /* Prejšnjega dneva je konec. */
      if (! prvi) {
        /* Če je bil ta dan suh, povečajmo števec suhih dni in dolžino trenutne
           skupine suhih dni. */
        if (suh) nVseh++, trenSkupina++;
        /* Če dan ni bil suh, postavimo dolžino skupine suhih dni na 0. */
        else trenSkupina = 0;
        /* Če je to najdaljša skupina suhih dni doslej, si jo zapomnimo. */
        if (trenSkupina > maxSkupina) maxSkupina = trenSkupina; }
      /* Začenja se nov dan. */
      if (ok) {
        prvi = false;              /* Zdaj gotovo nismo več pred prvo meritvijo. */
        prejDan = dan;             /* Zapomnimo si številko trenutnega dneva. */
        suh = (meritev == 0);      /* Ali je dan doslej suh? */ }
```

```
    }
  }
  while (ok);

  /* Izpišimo rezultate. */
  printf("%d suhih dni, največ %d skupaj.\n", nVseh, maxSkupina); return 0;
}
```

3. Virus

(*a*) Če imamo en sam računalnik, lahko okuženo zgoščenko najdemo z bisekcijo v največ 10 dneh. Prvi dan razdelimo zgoščenke na dve skupini po 500 zgoščenk in poženemo vse zgoščenke iz ene skupine. Če je računalnik okužen, vemo, da je bila okužena zgoščenka v tisti skupini, sicer pa v drugi (tisti, ki je nismo poganjali). Naslednji dan torej nadaljujemo z eno od teh dveh skupin (tisto, v kateri je okužena zgoščenka); razdelimo jo na dve skupini po 250 zgoščenk in tako naprej. Tako našo skupino vsak dan razpolovimo in deseti dan nam ostaneta največ dve zgoščenki; eno od njiju poženemo in če se računalnik sesuje, je bil virus na njej, sicer pa na tisti drugi.

(*b*) Če imamo le en dan časa, lahko virus poiščemo z 10 računalniki. Vsaki zgoščenki pripišimo enolično številko od 1 do 1000. Vsako številko zapišimo v dvojiškem sestavu kot niz 10 bitov (na primer, 123 v desetiškem je 0001111011 v dvojiškem). Na prvem računalniku poženimo vse zgoščenke, pri katerih je v dvojiškem zapisu prižgan prvi bit; na drugem vse zgoščenke, pri katerih je prižgan drugi bit; in tako naprej. Iz tega, kateri računalniki se sesujejo, kateri pa ne, lahko takoj razberemo številko zgoščenke, na kateri je bil virus.

4. Analiza enot

Ker so enote in količine predstavljene z malimi črkami angleške abecede, lahko v nalogi nastopa le 26 različnih enot in 26 različnih količin. Pripravimo si dvodimenzionalno tabelo (v spodnjem programu je to `def`), v kateri bo za vsako možno količino in za vsako možno enoto pisalo, s kakšno stopnjo se ta enota pojavlja v tej količini. Na začetku postavimo vse elemente te tabele na 0, nato pa berimo definicije količin in ustrezno povečujmo ali zmanjšujmo elemente tabele. Če v definiciji količine `k` naletimo na enoto `e`, moramo stopnjo `def[k][e]` povečati za 1, če `e` nastopa v številu, oz. zmanjšati za 1, če nastopa v imenovalcu. Vhodne podatke bomo brali znak po znak in pri tem v spremenljivkah `levo` in `zgoraj` hranili podatke o tem, ali smo levo od dvopičja ali desno od njega ter ali smo (ko smo enkrat desno od dvopičja) v številu ulomka (levo od znaka `/`) ali v imenovalcu (desno od znaka `/`).

Nato moramo prebrati še formulo samo; to je zelo podobno kot pri branju definicij količin, le da namesto dvopičja nastopa enačaj in da se lahko znak `/` pojavi tudi na levi strani, ne le na desni. Med branjem formule bomo v tabeli `stopnje` računali stopnje vseh enot v formuli. Ker formula sama v resnici ne vsebuje enot, pač pa količine, se moramo vsakič, ko v formuli naletimo na količino `k`, zapeljati v zanki po vseh enotah in k tabeli `stopnje` prišteti (ali odšteti) stopnje teh enot v definiciji količine `k` iz tabele `def[k][e]`. Prištevali bomo stopnje pri tistih količinah, ki se pojavljajo v številu leve strani formule ali v imenovalcu desne strani, odštevali pa pri tistih, ki se pojavljajo v imenovalcu leve ali v številu desne strani.

Na koncu moramo le še preveriti, če so vse stopnje v tabeli stopnje enake 0; če niso, potem vemo, da se enote na levi in desni strani formule ne ujemajo.

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

int main()
{
  int def[26][26], stopnje[26], nKolicin, k, e, c;
  bool levo, zgoraj;

  /* Inicializirajmo tabelo za definicije količin. */
  for (k = 0; k < 26; k++) for (e = 0; e < 26; e++) def[k][e] = 0;

  /* Preberimo definicije količin. */
  scanf("%d\n", &nKolicin);
  while (nKolicin-- > 0)
  {
    levo = true;

    /* Brali bomo po znakih vse do konca vrstice. */
    while ((c = fgetc(stdin)) != '\n')
      if (c == ':') levo = false, zgoraj = true;
      else if (c == '/') zgoraj = false;

      else if ('a' <= c && c <= 'z')
      {
        /* Črka levo od dvopičja nam pove količino, ki je definirana v tej vrstici;
           to si zapomnimo v spremenljivki k. */
        if (levo) k = c - 'a';
        /* Črka desno od dvopičja pa pomeni enoto, ki ji moramo zdaj stopnjo v
           definiciji količine k povečati ali zmanjšati za 1 (odvisno od tega,
           ali smo v števcu ali v imenovalcu — to nam pove spremenljivka „zgoraj“). */
        else def[k][c - 'a'] += (zgoraj) ? 1 : -1;
      }
  }

  /* V tabeli stopnje bomo hranili stopnje enot v formuli.
     Za začetek postavimo vse na 0. */
  for (e = 0; e < 26; e++) stopnje[e] = 0;

  /* Zdaj preberimo še formulo, ki jo moramo preveriti.
     Brali bomo po znakih do konca vrstice (ali EOF). */
  levo = true; zgoraj = true;
  while ((c = fgetc(stdin)) != '\n' && c != EOF)
    if (c == '=') levo = false, zgoraj = true;
    else if (c == '/') zgoraj = false;
    else if ('a' <= c && c <= 'z')

      /* Črka pomeni eno od količin; poglejmo v tabelo def, iz katerih enot je ta
         količina sestavljena, in njihove stopnje prištejmo ali odštejmo k tabeli
         stopnje. Prištevajo se stopnje količin v števcu leve strani formule
         in imenovalcu desne strani, odštevajo pa se stopnje količin v imenovalcu
         leve strani in števcu desne strani formule. */
      for (e = 0; e < 26; e++) stopnje[e] += (zgoraj == levo ? 1 : -1) * def[c - 'a'][e];

  /* Preverimo, če so vse stopnje enake 0, in izpišimo rezultat. */
  for (e = 0; e < 26; e++) if (stopnje[e] != 0) break;
  printf("Formula %s pravilna.\n", (e < 26) ? "ni" : "je");
  return 0;
}
```

5. Za žužke gre

V nalogi se skriva problem barvanja grafa z dvema barvama. Recimo, da spol žužkov označimo z 1 in 2, saj ne moremo vedeti, kateri je kateri. Zdaj lahko razmišljamo takole: izberimo si poljubnega žužka in mu dodelimo spol 1; iz tega lahko takoj zaključimo, da morajo vsi, ki so bili v kakšni interakciji z njim, imeti spol 2. Podobno pa, ko kakšnemu žužku pripišemo spol 2, lahko zaključimo, da so morali biti tisti, ki so bili v interakciji z njim, spola 1. Tako nadaljujemo in prej ali slej bodisi pripišemo spol vsem žužkom, ki se jih je dalo prek teh interakcij sploh doseči, bodisi naletimo na primer, ko nekemu žužku pripišemo en spol, kasneje pa ugotovimo, da bi mu morali pripisati še nasprotni spol. V tem primeru vemo, da žužkom ni mogoče pripisati spola v skladu z zahtevami naloge.

Paziti moramo še na možnost, da z gornjim postopkom še nismo pripisali spola vsem žužkom, ker je njihova populacija mogoče razdeljena na več manjših skupin, ki druga z drugo niso imele stikov. Zato moramo gornji postopek ponavljati, dokler je še kje kak žužek, ki mu nismo pripisali spola.

Zapišimo dobljeni postopek še s psevdokodo. Spol žužka u bomo hranili v b_u ($b_u = 0$ pomeni, da mu spola še nismo dodelili). V množici Q hranimo žužke, ki smo jim že pripisali spol, nismo pa še pregledali, kaj to pomeni za spol ostalih žužkov, ki so bili v stiku z njimi.

```
for u := 1 to n do b_u := 0;
for z := 1 to n do if b_z = 0:
  (* Žužek z še nima spola. Pripišimo mu ga in ga dodajmo v Q. *)
  b_z := 1; Q := {z};
  while Q ni prazna:
    u := poljuben element množice Q; pobriši u iz Q;
    za vsakega žužka v, ki je bil kdaj v interakciji z u:
      if b_v = 0 then
        (* v-ju pripišimo spol (nasprotnega od b_u) in ga dodajmo v Q. *)
        b_v := 3 − b_u; dodaj v v Q;
      else if b_v = b_u then
        (* Žužkom ni mogoče pripisati spola v skladu z omejitvami naloge. *)
        return false;
return true;
```

V praksi lahko Q predstavimo s kakršnim koli seznamom, skladom, vrsto ali čim podobnim. Za naštevanje žužkov v, ki so bili kdaj v interakciji z žužkom u, je načeloma koristno, če si na začetku za vsakega žužka u pripravimo seznam vseh, ki so bili kdaj v interakciji z njim (recimo temu seznamu $L[u]$). Tako nam ne bo treba vsakič iti po vhodnem seznamu vseh m interakcij v terariju, ampak bomo morali iti le po tistih, ki so res bili v interakciji z u. Zapišimo še ta postopek:[8]

```
for u := 1 to n do L[u] := prazen seznam;
for i := 1 to m:
  dodaj s_i v seznam L[t_i]; dodaj t_i v seznam L[s_i];
```

[8] Potencialno koristna literatura pri tej nalogi: M. Gregorič, K. Šuen, R.-C. Cheng, S. Kralj-Fišer, M. Kuntner, Spider behaviors include oral sexual encounters, *Nature Scientific Reports*, 6:25128 (29 April 2016).

REŠITVE NALOG ZA TRETJO SKUPINO

1. Letala

Za začetek uredimo zabojnike padajoče po teži, letala pa po nosilnosti in jih v tem vrstnem redu oštevilčimo: $m_1, \ldots m_n$ in $c_1, \ldots, c_k$.

Recimo, da si izberemo nek t in bi radi sestavili razpored zabojnikov med letala, s katerim bi vse zabojnike razvozili v t dneh. (To je seveda smiselno le, če je $k \cdot t \geq n$, drugače takoj vemo, da imamo preprosto premalo letal.) Najmočnejšemu letalu c_1 dodelimo najtežjih t zabojnikov, torej $m_1, \ldots, m_t$; drugemu najmočnejšemu letalu, c_2, dodelimo naslednjih t zabojnikov, torej $m_{t+1}, \ldots, m_{2t}$; in tako naprej. Če se pri kakšnem letalu izkaže, da kakšnega od zabojnikov, ki smo mu jih dodelili, ne more nesti, lahko takoj zaključimo, da veljavnega razporeda za izbrani t sploh ni.[9] Dovolj je, če pri vsakem letalu preverimo le najtežji zabojnik, ki smo mu ga dodelili — pri letalu c_j je to zabojnik $m_{t\cdot(j-1)+1}$. Takšno preverjanje nam torej vzame le $O(n/t)$ časa.

Najmanjši primerni t lahko zdaj poiščemo z bisekcijo. Na začetku preverimo, če je $m_1 > c_1$; če je, lahko takoj obupamo, saj je zabojnik m_1 pretežak za vsa letala. Drugače pa vemo, da bi se dalo zabojnike razvoziti v n dneh (vse s prvim letalom); po drugi strani vemo, da se jih ne da razvoziti v samo 0 dneh. Najmanjši t je torej nekje na območju $0 < t \leq n$. V nadaljevanju to območje razpolavljamo, dokler ne ostane ena sama vrednost t-ja.

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

int Primerjaj(const void *x, const void *y) { return *(const int *) y - *(const int *) x; }

#define MaxN 100000
#define MaxK 100000

int mi[MaxN], cj[MaxK], n, k;

int main()
{
  int i, j, tL, tH, t;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("letala.in", "rt");
  fscanf(f, "%d %d\n", &n, &k);
  for (i = 0; i < n; i++) fscanf(f, "%d", &mi[i]);
  for (j = 0; j < k; j++) fscanf(f, "%d", &cj[j]);
  fclose(f);

  /* Uredimo zabojnike in letala padajoče. */
```

[9] O tem se lahko prepričamo takole: naj bo c_j naše preobremenjeno letalo; najtežji zabojnik, ki smo mu ga dodelili, je m_i za $i = t \cdot (j-1) + 1$ in če je kakšen zabojnik za naše letalo pretežak, mora biti to ravno m_i. Recimo, da bi vendarle obstajal nek veljaven razpored za t dni; naj bo c_u letalo, ki v tem razporedu pelje zabojnik m_i. Ker je m_i pretežak za letalo c_j, je pretežak tudi za letala $c_{j+1}, \ldots, c_k$ (ki nimajo nič večje nosilnosti od c_j), torej mora biti c_u eno izmed letal $c_1, \ldots, c_{j-1}$. Ta letala lahko v t dneh razvozijo največ $t \cdot (j-1)$ zabojnik; ker je med njimi tudi zabojnik m_i (ki ga pelje letalo c_u) in ker je $i > t \cdot (j-1)$, to pomeni, da mora biti med zabojniki $m_1, \ldots, m_{t\cdot(j-1)}$ vsaj en tak, ki ga ne pelje nobeno od letal $c_1, \ldots, c_{j-1}$; recimo temu zabojniku m_v. Ker je $v > i$, je ta zabojnik vsaj toliko težak kot m_i; in ker je m_i pretežak za vsa letala od vključno c_j naprej, je zanje pretežak tudi m_v. Zabojnika m_v torej ne pelje nobeno od prvih $j-1$ letal, ostala pa ga sploh ne morejo peljati, torej tak razpored sploh ne more obstajati.

```
  qsort(mi, n, sizeof(mi[0]), &Primerjaj);
  qsort(cj, k, sizeof(cj[0]), &Primerjaj);
  /* Poiščimo najmanjši t z bisekcijo. */
  if (cj[0] < mi[0]) tL = −1, tH = −1; else tL = 0, tH = n;
  while (tH − tL > 1)
  {
    /* V tH dneh je mogoče razvoziti vse zabojnike, v tL dneh pa ne. */
    t = (tL + tH) / 2;
    /* Dajmo najmočnejšemu letalu najtežjih t zabojnikov, naslednjemu
       naslednjih t in tako naprej. Preverimo, če bo vsak zmogel peljati
       zabojnike, ki smo mu jih dodelili. */
    for (j = 0; j * t < n; j++)
      if (cj[j] < mi[j * t]) break;
    /* Če smo prišli do konca, ne da bi se pri kakšnem letalu izkazalo, da
       ne bo zmoglo, je mogoče tovor razvoziti v t dneh, sicer pa ne. */
    if (j * t < n) tL = t; else tH = t;
  }
  /* Izpišimo rezultat. */
  f = fopen("letala.out", "wt"); fprintf(f, "%d\n", tH); fclose(f); return 0;
}
```

Časovna zahtevnost te rešitve je $O(n \log n + k \log k)$ za urejanje obeh tabel, nato pa še $O(n \log n)$ za iskanje najmanjšega t-ja z bisekcijo. Pravzaprav bi ostali pri časovni zahtevnosti $O(n \log n)$ celo, če bi namesto bisekcije preizkušali kar vse možne t-je po vrsti od $t = n$ navzdol. Podobno bi ostali pri časovni zahtevnosti $O(n \log n)$, če bi pri posameznem t preverili vse zabojnike in ne le vsakega t-tega. Če pa naredimo obe poenostavitvi naenkrat (pregledamo vse zabojnike in ne uporabimo bisekcije), pademo v časovno zahtevnost $O(n^2)$, kar je za večje testne primere pri naši nalogi že prepočasi.

Oglejmo si zdaj še malo drugačno rešitev te naloge. Preštejmo, za koliko najtežjih zabojnikov velja, da jih lahko pelje le prvo letalo (tisto z največjo nosilnostjo, c_1), ostala letala pa ne. Recimo, da je teh zabojnikov p_1; ker jih ne bo mogoče razvoziti drugače kot tako, da prvo letalo vsak dan pelje po enega od njih, bomo za razvoz teh zabojnikov potrebovali p_1 dni, torej tudi za razvoz vsega tovora potrebujemo vsaj p_1 dni.

Nato poglejmo, koliko je takih zabojnikov, ki jih lahko peljeta najzmogljivejši dve letali, ostala pa ne; recimo, da je takih zabojnikov p_2. Za razvoz teh zabojnikov torej gotovo potrebujemo vsaj $\lceil p_2/2 \rceil$ dni, to pa je zato tudi spodnja meja za čas razvoza vsega tovora. Podobno, če se za p_3 zabojnikov izkaže, da jih lahko nesejo najzmogljivejša tri letala, ostala pa ne, lahko zaključimo, da za razvoz teh zabojnikov potrebujemo vsaj $\lceil p_3/3 \rceil$ dni, zato pa tudi za razvoz vsega tovora potrebujemo vsaj toliko dni. Tako nadaljujemo za vse več letal, vse do k, in med tako dobljenimi mejami vzamemo najvišjo: $t = \max_{0 \le j \le k} \lceil p_j/j \rceil$.

Iz dosedanjega razmisleka že vidimo, da v manj kot t dneh ne bomo mogli razvoziti vsega tovora. V t dneh pa gre: že pri prvi rešitvi smo videli, da lahko tak razpored sestavimo tako, da prvo letalo prepelje prvih t zabojnikov, drugo naslednjih t in tako naprej. Najtežji zabojnik, ki ga dobi letalo j, je zabojnik številka $t(j-1)+1$. To je naprej $\ge \lceil p_{j-1}/(j-1) \rceil (j-1) + 1 \ge p_{j-1} + 1$. Ta zabojnik torej ni eden od

najtežjih p_{j-1} zabojnikov, v prejšnjem odstavku pa smo p_{j-1} definirali tako, da letalo j lahko nese vse zabojnike razen najtežjih p_{j-1}. Torej zabojnik $t(j-1)+1$ za letalo j gotovo ni pretežak. Ker ta razmislek velja za vsako letalo, lahko zaključimo, da je naš razpored veljaven in je torej vse zabojnike res mogoče prepeljati v t dneh.

Lepo pri tej rešitvi je, da potrebujemo zanjo le en prehod po urejenem seznamu zabojnikov in hkrati še po urejenem seznamu letal. Časovna zahtevnost tega dela postopka je torej le $O(n+k)$, pred tem pa še vedno porabimo $O(n \log n + k \log k)$ časa za urejanje obeh seznamov. Zapišimo to rešitev še v C-ju:

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

#define MaxN 100000
#define MaxK 100000
int Primerjaj(const void *x, const void *y) { return *(const int *) y - *(const int *) x; }
int mi[MaxN], cj[MaxK], n, k;

int main()
{
  int i, j, t, kand;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("letala.in", "rt");
  fscanf(f, "%d %d\n", &n, &k);
  for (i = 0; i < n; i++) fscanf(f, "%d", &mi[i]);
  for (j = 0; j < k; j++) fscanf(f, "%d", &cj[j]);
  fclose(f);

  /* Uredimo zabojnike in letala padajoče. */
  qsort(mi, n, sizeof(mi[0]), &Primerjaj);
  qsort(cj, k, sizeof(cj[0]), &Primerjaj);

  /* Poiščimo najmanjši t z bisekcijo. */
  if (mi[0] > cj[0]) t = -1;
  else for (i = 0, j = 1, t = 0; j <= k; j++)
  {
    /* Za koliko zabojnikov velja, da jih lahko peljejo letala od 0 do j - 1,
       letala od j do k - 1 pa ne? */
    if (j == k) i = n;
    else while (i < n && mi[i] > cj[j]) i++;

    /* Teh i zabojnikov lahko torej razvozi le najmočnejših j letal,
       za kar bo potrebnih vsaj i/j dni. Ker mora biti število dni celo,
       bomo količnik i/j zaokrožili navzgor. */
    kand = (i + j - 1) / j;
    if (kand > t) t = kand;
  }

  /* Izpišimo rezultat. */
  f = fopen("letala.out", "wt"); fprintf(f, "%d\n", t); fclose(f); return 0;
}
```

2. Dominosa

Domine lahko polagamo z rekurzijo. Na vsakem koraku poiščemo najbolj zgornje še nepokrito polje, če je takih več, pa vzamemo najbolj levo od njih. Ker sta njegov zgornji in levi sosed (če ju sploh ima) že pokrita, lahko to polje pokrijemo le tako, da

ga povežemo z desnim ali pa s spodnjim sosedom. Za vsako od teh dveh možnosti preverimo, če take domine še nismo uporabili, in z rekurzivnim klicem poskusimo pokriti preostanek mreže.

Spodnji program uporablja globalno spremenljivko uporabljena za označevanje tega, katere domine je že uporabil. Ko položimo novo domino v mrežo, postavimo ustrezno vrednost v tej tabeli na **true**, po vrnitvi iz rekurzivnega klica pa nazaj na **false**. V tabeli pokrito pa označujemo, katera polja so že pokrita in katera ne; pri že pokritih si v tej tabeli tudi zapomnimo, kateri od njegovih sosedov pripada isti domini kot to polje — ta podatek bo prišel prav na koncu, ko bo treba mrežo domin narisati.

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

#define MaxPik 9
#define MaxW (MaxPik + 2)
#define MaxH (MaxPik + 1)

int t[MaxH][MaxW]; /* vhodna mreža */
typedef enum { Ne, N, W, E, S } PokritoT;
PokritoT pokrito[MaxH][MaxW];
int w, h, maxPik;
/* Veljavni elementi so uporabljena[p1][p2], 0 ≤ p1 ≤ p2 ≤ maxPik. */
bool uporabljena[MaxPik + 1][MaxPik + 1];

bool Rekurzija(int x, int y)
{
  int i, p1, p2, tmp;

  /* Poiščimo naslednje nepokrito polje. */
  p1 = x; p2 = y;
  while (y < h)
  {
    while (x < w && pokrito[y][x] != Ne) x++;
    if (x < w) break;
    x = 0; y++;
  }
  if (y >= h) return true; /* Vse je pokrito, našli smo resitev. */

  /* Poskusimo postaviti vodoravno domino. */
  if (x + 1 < w && pokrito[y][x + 1] == Ne) {
    p1 = t[y][x]; p2 = t[y][x + 1];
    if (p2 < p1) tmp = p1, p1 = p2, p2 = tmp;
    if (! uporabljena[p1][p2])
    {
      uporabljena[p1][p2] = true; pokrito[y][x] = E; pokrito[y][x + 1] = W;
      if (Rekurzija(x + 2, y)) return true;
      uporabljena[p1][p2] = false; pokrito[y][x] = Ne; pokrito[y][x + 1] = Ne;
    }
  }

  /* Poskusimo postaviti navpično domino. */
  if (y + 1 < h && pokrito[y + 1][x] == Ne)
  {
    p1 = t[y][x]; p2 = t[y + 1][x];
    if (p2 < p1) tmp = p1, p1 = p2, p2 = tmp;
    if (! uporabljena[p1][p2])
    {
```

```
        uporabljena[p1][p2] = true; pokrito[y][x] = S; pokrito[y + 1][x] = N;
        if (Rekurzija(x + 1, y)) return true;
        uporabljena[p1][p2] = false; pokrito[y][x] = Ne; pokrito[y + 1][x] = Ne;
      }
    }
  }
  return false;
}

int main()
{
  int y, x;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("dominosa.in", "rt");
  fscanf(f, "%d", &maxPik);
  h = maxPik + 1; w = h + 1;
  for (y = 0; y < h; y++) for (x = 0; x < w; x++) {
    fscanf(f, "%d", &t[y][x]); pokrito[y][x] = Ne; }
  fclose(f);

  for (y=0; y<=maxPik; y++) for (x=0; x<=maxPik; x++) uporabljena[y][x] = false;
  Rekurzija(0, 0);

  /* Izpišimo rezultate. */
  f = fopen("dominosa.out", "wt");
  for (y = 0; y < h; y++) {
    for (x = 0; x < w; x++) fprintf(f, "%d%s", t[y][x],
                                    (pokrito[y][x] == E) ? "-" : (x == w - 1 ? "\n" : "."));
    if (y < h - 1) for (x = 0; x < w; x++) fprintf(f, "%s%s",
                                (pokrito[y][x] == S) ? "|" : ".", (x == w - 1) ? "\n" : "."); }
  fclose(f); return 0;
}
```

3. Galaktična zavezništva

Oglejmo si najprej preprosto in malo manj učinkovito rešitev, ki je dovolj dobra za manjše testne primere (recimo do $n \approx 2000$; na našem tekmovanju bi dovolj hitro rešila polovico testnih primerov). Ker so naši nizi dolgi največ 50 bitov, lahko posamezni niz predstavimo kar kot 64-bitno celoštevilsko spremenljivko (npr. tip **long long** v C/C++, **long** v javi ipd.). S tremi gnezdenimi zankami lahko pregledamo vse možne trojice planetov, pri vsaki izračunamo xor njihovih števil in si zapomnimo največjo med njimi.

```
#include <stdio.h>

#define MaxM 50
#define MaxN 7500

int main()
{
  int n, m, i, j, tren, b, a1, a2, a3;
  long long kand, kand2, naj, nizi[MaxN];
  char buf[MaxM + 3];

  /* Preberimo n in m. */
  FILE *f = fopen("xor.in", "rt");
  fscanf(f, "%d %d\n", &n, &m);

  /* Preberimo nize. */
```

```
  for (i = 0; i < n; i++) {
    fgets(buf, sizeof(buf), f);
    nizi[i] = 0;
    for (j = 0; j < m; j++) if (buf[j] == '1')
      nizi[i] |= ((long long) 1) << (m - 1 - j); }
  fclose(f);
  /* Preglejmo vse trojice. */
  for (naj = 0, a1 = 0; a1 < n - 2; a1++) for (a2 = a1 + 1; a2 < n - 1; a2++)
    for (a3 = a2 + 1, kand2 = nizi[a1] ^ nizi[a2]; a3 < n; a3++) {
      kand = kand2 ^ nizi[a3];
      if (kand > naj) naj = kand; }
  /* Izpišimo rezultate. */
  f = fopen("xor.out", "wt");
  for (j = 0; j < m; j++) fputc('0' + (int) ((naj >> (m - 1 - j)) & 1), f);
  fclose(f); return 0;
}
```

Težava pri tem postopku je, da je vseh trojic $\binom{n}{3} = n(n-1)(n-2)/6$, tako da ima ta postopek časovno zahtevnost $O(n^3)$. (V splošnem pravzaprav $O(n^3 m)$, ker za xor dveh m-bitnih nizov potrebujemo $O(m)$ časa, če naši nizi niso tako kratki, da gredo v eno samo celoštevilsko spremenljivko.) Pri večjih n zato ta postopek postane prepočasen.

Razmislimo za začetek o malo poenostavljeni različici naloge: recimo, da namesto zavezništev treh planetov gledamo zavezništva dveh. Iščemo torej dva niza a in b tako, da bo $c = a \operatorname{xor} b$ čim večji. Mogoče se nekateri vhodni nizi začnejo na ničlo, nekateri pa na enico. Če tedaj izberemo niza a in b tako, da se bo eden začel na ničlo, drugi pa na enico, se bo c začel na enico; če pa izberemo a in b tako, da se bosta oba začela na ničlo ali pa oba na enico, se bo c začel na ničlo. Vsak c, ki se začne na enico, je večji (zavezništvo je močnejše) od vsakega c, ki se začne na ničlo.

Podobno razmišljamo tudi pri nižjih bitih: vedno poskušamo niza a in b nadaljevati tako, da se v istoležnem bitu razlikujeta, tako da bo imel c na tistem mestu enico. Šele če to ne gre (ker primernih vhodnih nizov sploh ni), poskusimo tudi možnost, da oba (a in b) nadaljujemo z ničlo ali pa oba z enico.

Ta ideja nas pripelje do naslednjega rekurzivnega postopka. Spodnja funkcija vrne največji $a' \operatorname{xor} b'$, ki ga je mogoče dobiti, če za a' vzamemo kakšen tak vhodni niz, ki se začne na a, in za b' vzamemo kakšen tak vhodni niz, ki se začne na b. Če primernih vhodnih nizov sploh ni, funkcija vrne -1.

funkcija REK(globina g, niza $a = a_1 a_2 \ldots a_g$ in $b = b_1 b_2 \ldots b_g$):
  if se noben vhodni niz ne začne na a ali pa noben na b **then return** -1;
  if $g = m$ **then return** $a \operatorname{xor} b$;
  $c := \max\{\text{REK}(g+1, a0, b1), \text{REK}(g+1, a1, b0)\}$;
  if $c = -1$ **then**
    $c := \max\{\text{REK}(g+1, a0, b0), \text{REK}(g+1, a1, b1)\}$;
  return c;

Postopek poženemo z globino $g = 0$ in praznima nizoma a in b.

Da bomo lahko učinkovito preverjali, če obstaja kak vhodni niz, ki se začne na $a_1 \ldots a_g$ (ali $b_1 \ldots b_g$), je koristno zložiti vhodne nize v drevo (*trie*). Vsako vozlišče drevesa ima načeloma dva otroka z oznakama `0` in `1` (nista pa nujno oba prisotna);

za vsak vhodni niz obstaja v drevesu veja (pot od korena do nekega lista), na kateri so oznake vozlišč ravno biti tega niza.

Naš rekurzivni postopek lahko zdaj namesto niza $a_1a_2\ldots a_g$ uporabi vozlišče drevesa, do katerega pridemo, če začnemo v korenu in po vrsti sledimo povezavam $a_1, a_2, \ldots, a_g$. To, da se nizu a na koncu pritakne še en bit, zdaj pomeni, da se moramo iz tistega vozlišča premakniti v ustreznega otroka. Če otroka, ki ga potrebujemo, ni, pa vemo, da med vhodnimi nizi ni nobenega takega, ki bi se začel na naš novi niz a.

funkcija REK(globina g, vozlišči a in b):
1 **if** $a = \text{NIL}$ **or** $b = \text{NIL}$ **then return** -1;
2 **if** $g = m$ **then return** $a \operatorname{xor} b$;
3 $c := \max\{\text{REK}(g+1, a.otrok[0], b.otrok[1]), \text{REK}(g+1, a.otrok[1], b.otrok[0])\}$;
4 **if** $c = -1$ **then**
5 $c := \max\{\text{REK}(g+1, a.otrok[0], b.otrok[0]), \text{REK}(g+1, a.otrok[1], b.otrok[1])\}$;
6 **return** c;

Kakšna je časovna zahtevnost tega postopka? Pri rekurzivnih klicih v vrstici 3 je tako, da če kakšen od otrok manjka, bo tisti klic to takoj ugotovil (v vrstici 1) in se vrnil (takemu klicu recimo, da je trivialen); če pa sta oba otroka prisotna, bo ta klic gotovo našel neko rešitev in torej vrnil nek veljaven niz c, ne pa -1; in v tem primeru se vrstica 5 ne bo izvedla. To pa pomeni, da se na primer z otrokom $a.otrok[0]$ izvede samo en netrivialen rekurziven klic: če je to tisti v vrstici 3, se vrstica 5 sploh ne bo izvedla; če pa je tisti v vrstici 3 trivialen, se bo izvedel tisti v vrstici 5. Podobno je tudi z drugimi otroki. Iz tega vidimo, da se lahko v celotnem času izvajanja našega postopka vsako vozlišče drevesa pojavi kot parameter a pri največ enem rekurzivnem klicu, v paru z največ enim drugim vozliščem b. Vseh rekurzivnih klicev skupaj je torej le toliko, kolikor je vozlišč v drevesu, to pa je $O(n \cdot m)$. (Vsi vhodni nizi skupaj so dolgi $n \cdot m$ znakov, drevo pa ima lahko manj kot toliko vozlišč, če se več nizov ujema v prvih nekaj bitih.) Koliko dela pa imamo s posameznim klicem (če odmislimo čas, porabljen v morebitnih vgnezdenih klicih)? V vrstici 2 moramo računati xor dveh m-bitnih nizov, v vrsticah 3 in 5 pa max dveh m-bitnih nizov; vse to so načeloma operacije, ki porabijo $O(m)$ časa, vendar pa se lahko z nekaj pazljivosti temu faktorju $O(m)$ izognemo. Niz $a \operatorname{xor} b$ lahko računamo sproti, bit po bit, že med samimi rekurzivnimi klici, in ga hranimo v neki globalni spremenljivki, tako da bo, ko bi morala vrstica 2 računati $a \operatorname{xor} b$, ta vrednost v resnici že izračunana. S pomočjo take globalne spremenljivke lahko funkcija REK tudi sproti preverja, ali ima vrednost $a \operatorname{xor} b$, ki nam trenutno nastaja, sploh kakšne možnosti, da bi bila boljša od najboljše doslej znane. Tako nam v vrsticah 3 in 5 ne bo treba računati max, ker bodo rekurzivni klici sami pravočasno obupali in vrnili -1, če ne bodo mogli vrniti nove najboljše rešitve. (Podrobnosti tega si bomo ogledali v izvorni kodi rešitve malo kasneje.) Tako torej vidimo, da ima naš postopek zahtevnost le $O(nm)$, če ga pazljivo implementiramo.

Doslej smo razmišljali o iskanju največjega xor-a dveh nizov, naloga pa zahteva največji xor treh nizov. Ta problem lahko rešimo z eno dodatno zanko: prvi niz fiksirajmo, nato pa s postopkom, podobnim zgornjemu, poiščimo še dva taka niza, ki bosta skupaj s prvim dala čim večji xor.

$naj := -1$;

za vsakega od n vhodnih nizov, recimo mu p:
  pobriši p iz drevesa;
  $c :=$ največja možna vrednost $p \operatorname{xor} a \operatorname{xor} b$ po vseh preostalih nizih a, b; (†)
  if $c > naj$ **then** $naj := c$;
  dodaj p nazaj v drevo;

Brisanje iz drevesa in dodajanje vanj vzameta vsakič po $O(m)$ časa, prav tako primerjava rešitve c z najboljšo rešitvijo doslej. Vrstico (†) pa lahko izvedemo z rekurzivnim postopkom, zelo podobnim tistemu, ki smo ga razvili malo prej za iskanje največje vrednosti $a \operatorname{xor} b$. Zdaj nas namesto te zanima $p \operatorname{xor} a \operatorname{xor} b$. Razlika je predvsem v tem, da če ima p na nekem indeksu enico, si zdaj od a-ja in b-ja želimo, da bi imela tam oba enako vrednost (oba ničlo ali oba enico), ne pa eden ničlo in eden enico; takrat moramo torej vrstici 3 in 5 funkcije Rek ravno zamenjati. Še ena razlika pa je naslednja: pri dosedanji funkciji Rek se je lahko zgodilo, da kažeta a in b na eno in isto vozlišče drevesa, ki predstavlja en sam vhodni niz; to načeloma ni dopustna rešitev, saj ne predstavlja zavezništva dveh planetov. Vendar je bil v takem primeru $a \operatorname{xor} b = a \operatorname{xor} a = 0$, mi pa smo iskali maksimalni xor, tako da to ni moglo vplivati na naše rezultate. Zdaj, pri zavezništvih treh planetov, pa moramo biti bolj pazljivi. Če je v p veliko bitov prižganih, a in b pa predstavljata en in isti vhodni niz, bo $p \operatorname{xor} a \operatorname{xor} b = p$ lahko precej velika vrednost, mogoče celo večja od katerega koli xora treh različnih nizov. To bi lahko povzročilo, da bi naš postopek neupravičeno vrnil rezultat, ki ga v resnici ni mogoče doseči z nobenim zavezništvom treh planetov, ampak ga lahko doseže le nek planet sam zase. Da se tej težavi izognemo, mora funkcija Rek, preden v vrstici 2 vrne rezultat, preveriti, če a in b kažeta na isto vozlišče; če da in če do tega vozlišča pripelje en sam vhodni niz, potem rezultat $a \operatorname{xor} b$ ne bi bil veljaven in moramo namesto njega vrniti -1. Če pa imamo v vhodnih podatkih res dva popolnoma enaka niza, je rešitev z $a = b$ lahko čisto sprejemljiva.

Skupaj je torej časovna zahtevnost naše rešitve za zavezništva treh planetov $O(n^2 m)$. Zapišimo dobljeni postopek še v C-ju:

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

#define MaxM 50
#define MaxN 7500
typedef struct { int otrok[2], listov; } Vozlisce;
Vozlisce *vozl;
int koren, nVozlisc, n, m, prvi, naj[MaxM];
char nizi[MaxN][MaxM + 3];

bool Rekurzija(int u, int v, int g, bool jeNaj)
{
  /* Na tem mestu velja: u in v sta vozlišči v drevesu na globini g (koren je na globini 0,
     listi na globini m). Xor nizov, ki ju predstavljata ti dve vozlišči, in prvih g bitov niza
     „prvi“, je shranjen v prvih g elementih tabele „naj“ in je enak (če je jeNaj = false) oz.
     večji (če je jeNaj = true) od prvih g bitov najmočnejšega doslej znanega zavezništva.
     Če je jeNaj = true, to pomeni, da moramo z rekurzijo nadaljevati, tudi če bomo v
     preostalih bitih dobili manjše vrednosti od dosedanje „naj“, saj bo trenutno
     zavezništvo že zaradi prvih g bitov boljše od najboljšega doslej znanega.
```

```
    Funkcija Rekurzija vrne true, če uspe najti novo najboljšo rešitev doslej. */
  int pb, prejNajBit; bool kajNasli, prejJeNaj;
  Vozlisce *U = vozl + u, *V = vozl + v;

  /* Če teh vozlišč sploh ni (ali pa sta prazni, ker smo tam iz
     drevesa začasno zbrisali niz „prva“), se takoj vrnimo. */
  if (u < 0 || v < 0) return false;
  if (u == v) { if (U->listov < 2) return false; }
  else if (U->listov == 0 || V->listov == 0) return false;
  /* Če smo prišli do listov, se ta veja rekurzije konča. */
  if (g == m) return jeNaj;

  pb = nizi[prvi][g]; /* Trenutni bit planeta „prvi“. */

  /* Poskusimo najprej nadaljevati pot tako, da bo imel xor vseh
     treh planetov na mestu g prižgan bit. */
  prejJeNaj = jeNaj; prejNajBit = naj[g];
  /* Če je imelo najboljše dosedanje zavezništvo tu ugasnjen bit, bo naše nastajajoče
     zavezništvo (ki bo imelo tu prižgan bit) gotovo boljše od njega. */
  if (naj[g] == 0) jeNaj = true;
  naj[g] = 1; /* Naše zavezništvo bo imelo tu prižgan bit. */
  /* Izvedimo zdaj oba rekurzivna klica. */
  kajNasli = false; /* Ali smo v gnezdenih klicih našli kakšno rešitev? */
  if (Rekurzija(U->otrok[0], V->otrok[pb ^ 1], g + 1, jeNaj))
    kajNasli = true, jeNaj = false;
  if (u != v || pb == 1)
    if (Rekurzija(U->otrok[1], V->otrok[pb], g + 1, jeNaj)) kajNasli = true;
  /* Če smo našli kakšno rešitev, se lahko takoj vrnemo in nam ni treba preizkušati
     še tistih, ki bi imele na mestu g ničlo namesto enice. */
  if (kajNasli) return true;
  /* Vrnimo spremenljivki jeNaj in naj[g] nazaj v prvotno stanje. */
  jeNaj = prejJeNaj; naj[g] = prejNajBit;

  /* Zdaj poskusimo nadaljevati tako, da bo imel xor vseh treh planetov
     na mestu g ugasnjen bit. */
  /* Takšno nadaljevanje pa nima smisla, če že poznamo neko drugo rešitev,
     ki se na višjih bitih ujema z našo, na trenutnem mestu pa ima prižgan bit. */
  if (! jeNaj && naj[g] == 1) return false;
  naj[g] = 0; /* Naše zavezništvo bo imelo tu ugasnjen bit. */
  /* Izvedimo zdaj oba rekurzivna klica. */
  if (Rekurzija(U->otrok[0], V->otrok[pb], g + 1, jeNaj)) kajNasli = true, jeNaj = false;
  if (u != v || pb == 0)
    if (Rekurzija(U->otrok[1], V->otrok[pb ^ 1], g + 1, jeNaj)) kajNasli = true;
  if (kajNasli) return true;
  /* Če nismo našli nove najboljše rešitve, povrnimo vrednost naj[g]
     v prvotno stanje. */
  naj[g] = prejNajBit; return false;
}

int main()
{
  int i, j, tren, b;

  /* Preberimo n in m. */
  FILE *f = fopen("xor.in", "rt");
  fscanf(f, "%d %d\n", &n, &m);

  /* Pripravimo tabelo za vozlišča drevesa in ustvarimo koren. */
  vozl = (Vozlisce *) malloc(sizeof(Vozlisce) * m * n);
```

```
  koren = 0; nVozlisc = 1;
  vozl[koren].otrok[0] = −1; vozl[koren].otrok[1] = −1; vozl[koren].listov = n;
  /* Preberimo vseh n nizov in jih dodajmo v drevo. */
  for (i = 0; i < n; i++) {
    fgets(nizi[i], sizeof(nizi[i]), f);
    tren = koren;
    for (j = 0; j < m; j++) {
      b = nizi[i][j] −= '0';
      /* Če še ni otroka, v katerem moramo nadaljevati pot po drevesu, ga zdaj */
      if (vozl[tren].otrok[b] < 0) {                          /* ustvarimo. */
        vozl[nVozlisc].otrok[0] = −1; vozl[nVozlisc].otrok[1] = −1;
        vozl[nVozlisc].listov = 0; vozl[tren].otrok[b] = nVozlisc++; }
      /* Premaknimo se v otroka, ki ustreza trenutnemu bitu našega niza. */
      tren = vozl[tren].otrok[b]; vozl[tren].listov++; }}
  fclose(f);
  /* Za vsak možen izbor prvega planeta izberimo ostala dva tako,
     da bosta dala skupaj s prvim čim večji xor vseh treh nizov. */
  for (j = 0; j < m; j++) naj[j] = 0;
  for (prvi = 0; prvi < n; prvi++)
  {
    /* Pobrišimo planet „prvi“ iz drevesa. */
    for (j = 0, tren = koren; j < m; j++) {
      tren = vozl[tren].otrok[nizi[prvi][j]]; vozl[tren].listov−−; }
    /* Z rekurzijo poiščimo najboljše zavezništvo tega planeta s še dvema drugima. */
    Rekurzija(koren, koren, 0, false);
    /* Dodajmo planet „prvi“ nazaj v drevo. */
    for (j = 0, tren = koren; j < m; j++) {
      tren = vozl[tren].otrok[nizi[prvi][j]]; vozl[tren].listov++; }
  }
  /* Izpišimo rezultate. */
  f = fopen(("xor.out", "wt");
  for (j = 0; j < m; j++) fputc('0' + naj[j], f);
  fclose(f); free(vozl); return 0;
}
```

Drevo lahko uporabimo tudi še drugače. Z dvema gnezdenima zankama si na vse možne načine izberimo prva dva planeta, recimo p in a; potem pa se vprašajmo: kateri b moramo vzeti, da bo skupaj s tema p in a dal najmočnejše zavezništvo $p \operatorname{xor} a \operatorname{xor} b$? Takega b-ja z drevesom ni težko poiskati: začnemo v korenu drevesa, nato pa se v vsakem koraku spustimo v tistega otroka, pri katerem se bo trenutni bit b-ja razlikoval od istoležnega bita v $p \operatorname{xor} a$ (tako da bo ta bit v $p \operatorname{xor} a \operatorname{xor} b$ prižgan). Šele če takega otroka ni, uporabimo tistega drugega, pri katerem se bo trenutni bit b-ja ujemal z istoležnim bitom v $p \operatorname{xor} a$. Tudi ta postopek ima časovno zahtevnost $O(n^2 m)$, le za nek konstantni faktor je slabši od prejšnjega.

4. Asteroidi

Preprosta, vendar neučinkovita rešitev je, da predstavimo igralno površino z dvodimenzionalno tabelo velikosti $w \times h$. V njej označimo, katere celice (kvadratki) so proste, katere pa zasedajo asteroidi (to nam vzame $O(wh)$ časa, ker se asteroidi ne prekrivajo in ne štrlijo iz mreže, tako da je vsota njihovih ploščin $\leq w \times h$). V tej

tabeli nato z iskanjem v širino poiščemo najkrajšo pot od začetnega položaja $(0, y_0)$ do desnega roba, torej do poljubne celice oblike $(w-1, y)$. Ta postopek torej porabi $O(wh)$ časa in tudi $O(wh)$ pomnilnika, tako da je primeren le za manjše testne primere (na našem tekmovanju bi z njim rešili polovico testnih primerov).

Porabo pomnilnika lahko sicer malo zmanjšamo, če ne predstavimo cele mreže naenkrat; ker se lahko naše vesoljsko plovilo premika le v desno, ne pa v levo, to pomeni, da ko iščemo najkrajše poti do celic v stolpcu x, se nam ni treba ukvarjati s tem, kaj se dogaja v mreži desno od njega (torej v stolpcih od $x+1$ naprej), pa tudi ne s tem, kaj se dogaja levo od stolpca $x-1$, saj plovilo ne more kar skočiti za več kot eno enoto v desno. Ni nam torej treba imeti v pomnilniku cele tabele velikosti $w \times h$, ampak le po dva stolpca naenkrat. Poraba pomnilnika se tako zmanjša na $O(h)$. Pri vsakem stolpcu moramo iti po vseh asteroidih, da vidimo, kateri so prisotni v tem stolpcu; časovna zahtevnost je tako $O(w(h+n))$, kar je pravzaprav še vedno $O(wh)$, saj je pri večjih testnih primerih n veliko manjši od h. Za takšne primere je ta postopek še vedno veliko prepočasen.

Primerjajmo v mislih dva sosednja stolpca, recimo x in $x+1$. Razlikujeta se lahko le v primeru, če se kakšen asteroid konča v stolpcu x in/ali če se kakšen asteroid začne v stolpcu $x+1$. Vsak asteroid torej lahko povzroči takšno spremembo pri največ dveh x-ih (na svojem levem robu in na svojem desnem robu). Sprememb je torej največ $2n$, vseh stolpcev pa je w. Pri naši nalogi gre lahko w do 10^6, število asteroidov n pa je največ 300. Neizogibno je torej, da nastopijo v naši mreži velike skupine zaporednih enakih stolpcev. Če sta dva sosednja stolpca enaka, to pomeni, da lahko v obeh opravimo popolnoma enake premike v smeri gor in dol; vse, kar lahko naredimo v enem, lahko naredimo tudi v drugem in obratno. Zato se lahko dogovorimo, da bomo v taki skupini več zaporednih enakih stolpcev izvedli premike gor ali dol le v zadnjem od njih, v prejšnjih stolpcih pa se bomo premikali samo v desno.

Našo prejšnjo rešitev lahko torej izboljšamo tako, da ne gledamo vseh stolpcev, ampak le tiste, pri katerih nastopi kakšna sprememba. Namesto časovne zahtevnosti $O(wh)$ imamo zdaj le še $O(nh)$.

Podoben razmislek pride prav tudi pri vrsticah. Recimo, da naše plovilo nekje na svoji poti naredi nekaj premikov desno v vrstici y, nato pa se premakne navzdol v vrstico $y+1$. Če se v vrstici $y+1$ ne začne noben asteroid (torej če to ni najvišja vrstica kakšnega asteroida), so vsa polja, ki so bila prosta v y, prosta tudi v $y+1$, torej bi lahko iz vrstice y takoj (pred premiki desno) naredili še premik dol in se nato premikali v desno po vrsici $y+1$ namesto po vrstici y. Podoben razmislek lahko naredimo tudi, če premikom desno sledi premik navzgor (in če se v vrstici $y-1$ ne konča noben asteroid). Tako torej vidimo, da se smemo omejiti na take poti, pri kateri vsi premiki desno potekajo po takih vrsticah, ki ležijo tik nad ali pa tik pod kakšnim asteroidom. (Poseben primer je le še možnost, da plovilo celotno pot opravi kar v svoji začetni vrstici y_0, če mu pri tem ni v napoto nobeden od asteroidov.)

Zdaj smo torej od vseh h vrstic obdržali največ $2n+1$ vrstic (vrstico y_0 in tiste tik pod in tik nad asteroidi). Da bomo lažje preverjali, kje nas kakšen asteroid ovira pri navpičnih premikih, je koristno od vsakega asteroida obdržati tudi eno notranjo vrstico (torej tako, v kateri ta asteroid vsaj delno leži). Tako nam ostane

največ $3n + 1$ vrstic — lahko jih je manj, če se več asteroidov začne ali konča v isti vrstici ipd.; spodnji program ima za to spremenljivko nys. Vse ostale vrstice pa lahko ignoriramo. Spodnji program koordinate vrstic kar preštevilči iz prvotnega območja $0, \ldots, h-1$ v $0, \ldots,$ nys -1. Prvotne koordinate vrstic moramo uporabljati le takrat, ko računamo, koliko navpičnih premikov je treba za premik iz ene vrstice v drugo.

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

int Primerjaj(const void *a, const void *b) {
  return *(const int *) a - *(const int *) b; }

#define MaxN 300

int w, h, n, y0, X1[MaxN], X2[MaxN], Y1[MaxN], Y2[MaxN];
int xs[2 * MaxN + 2], ys[3 * MaxN + 1], nxs, nys;
long long naj[3 * MaxN + 1]; bool zasedeno[3 * MaxN + 1];

int main()
{
  int i, ix, iy; long long kand;

  /* Preberimo vhodne podatke. */
  FILE *f = fopen("asteroidi.in", "rt");
  fscanf(f, "%d %d %d %d", &h, &w, &y0, &n);
  for (i = 0, nxs = 0; i < n; i++)
  {
    fscanf(f, "%d %d %d %d", &X1[i], &Y1[i], &X2[i], &Y2[i]);
    if (X1[i] > 0) xs[nxs++] = X1[i] - 1;
    xs[nxs++] = X2[i]; ys[nys++] = Y1[i] - 1; ys[nys++] = Y1[i]; ys[nys++] = Y2[i] + 1;
  }
  fclose(f);
  xs[nxs++] = 0; xs[nxs++] = w - 1; ys[nys++] = y0;

  /* Uredimo seznama zanimivih vrstic in stolpcev. */
  qsort(xs, nxs, sizeof(xs[0]), &Primerjaj);
  qsort(ys, nys, sizeof(ys[0]), &Primerjaj);

  /* Iz seznama zanimivih vrstic pobrišimo morebitne duplikate. */
  for (iy = 0, i = 1; i < nys; i++) if (iy == 0 || ys[iy - 1] != ys[i]) ys[iy++] = ys[i];
  nys = iy;

  /* Preštevilčimo y-koordinate asteroidov z območja 0... h − 1 na 0... nys − 1. */
  for (i = 0; i < n; i++) {
    iy = 0; while (ys[iy] < Y1[i]) iy++;
    Y1[i] = iy; while (ys[iy] <= Y2[i]) iy++;
    Y2[i] = iy; }
  /* Asteroid i zdaj pokriva vrstice od vključno ys[y1] do vključno ys[y2] − 1. */

  /* Na levem robu je najprej dosegljivo le polje v vrstici y0. */
  for (iy = 0; iy < nys; iy++) naj[iy] = (ys[iy] == y0) ? 0 : -1;

  /* Preglejmo zanimive stolpce od leve proti desni. */
  for (ix = 0; ix < nxs; ix++)
  {
    if (ix > 0 && xs[ix - 1] == xs[ix]) continue; /* Preskočimo duplikate v xs[]. */

    /* Poglejmo, katera polja v tem stolpcu zasedajo asteroidi.
       Tam premik desno iz xs[ix − 1] v xs[ix] ni mogoč. */
```

```
    for (iy = 0; iy < nys; iy++) zasedeno[iy] = (ys[iy] < 0 || ys[iy] >= h);
    for (i = 0; i < n; i++) if (X1[i] <= xs[ix] && xs[ix] <= X2[i])
      for (iy = Y1[i]; iy < Y2[i]; iy++) zasedeno[iy] = true, naj[iy] = -1;
    /* Upoštevajmo premike dol. */
    for (iy = 1; iy < nys; iy++) if (naj[iy - 1] >= 0 && ! zasedeno[iy]) {
      kand = naj[iy - 1] + ys[iy] - ys[iy - 1];
      if (naj[iy] < 0 || kand < naj[iy]) naj[iy] = kand; }
    /* Upoštevajmo premike gor. */
    for (iy = nys - 2; iy >= 0; iy--) if (naj[iy + 1] >= 0 && ! zasedeno[iy]) {
      kand = naj[iy + 1] + ys[iy + 1] - ys[iy];
      if (naj[iy] < 0 || kand < naj[iy]) naj[iy] = kand; }
  }
  /* Izpišimo najboljši rezultat. */
  for (kand = -1, iy = 0; iy < nys; iy++)
    if (naj[iy] >= 0) if (kand < 0 || naj[iy] < kand) kand = naj[iy];
  f = fopen("asteroidi.out", "wt"); fprintf(f, "%lld\n", kand); fclose(f); return 0;
}
```

5. Brisanje niza

Nalogo lahko rešujemo z dinamičnim programiranjem. Označimo i-ti znak našega niza s s_i, torej imamo niz $s = s_1 s_2 \ldots s_n$. Naloga sprašuje po najmanjšem številu brisanj, s katerim pobrišemo celoten niz, koristno pa jo je malo posplošiti in računati potrebno število brisanj tudi za s-jeve podnize. Naj bo torej $f(i, j)$ najmanjše število brisanj, s katerim je mogoče popolnoma pobrisati niz $s_i s_{i+1} \ldots s_j$. Pri izračunu $f(i, j)$ lahko razmišljamo takole: eno od teh brisanj bo moralo pobrisati tudi znak s_i. Mogoče ne bo pobrisalo nobenega drugega; tedaj bodo morala ostala brisanja v celoti pobrisati niz $s_{i+1} \ldots s_j$, za to pa je potrebnih vsaj $f(i + 1, j)$ brisanj. Skupaj z brisanjem znaka s_i imamo torej $1 + f(i + 1, j)$ brisanj, kar je eden od kandidatov za $f(i, j)$.

Druga možnost pa je, da tisto brisanje, ki pobriše s_i, pobriše tudi še kakšen kasnejši znak (ki mora biti seveda enak znaku s_i). Naj bo s_k (za nek k z območja $i < k \leq j$) prvi naslednji znak, ki ga pobriše isto brisanje kot znak s_i. Preden postane tako brisanje sploh mogoče, smo morali v celoti pobrisati podniz med znakoma s_i in s_k, torej podniz $s_{i+1} \ldots s_{k-1}$; za to potrebujemo $f(i + 1, k - 1)$ brisanj. Po tistem nam od niza $s_i s_{i+1} \ldots s_{k-1} s_k \ldots s_j$ ostane le $s_i s_k \ldots s_j$; ker sta znaka s_i in s_k enaka, bomo lahko s_i vsekakor pobrisali z istim brisanjem, ki bo pobrisalo s_k; torej je potrebno število brisanj za niz $s_i s_k \ldots s_j$ enako kot za niz $s_k \ldots s_j$, to pa je $f(k, j)$ brisanj. Tako imamo torej skupaj $f(i + 1, k - 1) + f(k, j)$ brisanj, kar je še eden od kandidatov za $f(i, j)$.

Med vsemi tako dobljenimi kandidati za $f(i, j)$ vzamemo najmanjšega:

$$f(i,j) = \min\{\ 1 + f(i+1, j-1),\\ \min\{f(i+1, k-1) + f(k, j-1) : i < k \leq j,\ s_i = s_k\}\ \}.$$

Robni primeri nastopijo pri $j < i$, ko imamo v resnici prazen podniz in brisanj sploh ne potrebujemo; takrat je torej $f(i, j) = 0$. Opazimo lahko, da za izračun $f(i, j)$ potrebujemo vrednosti funkcije f za krajše podnize znotraj $s_i \ldots s_j$, torej je koristno vrednosti funkcije f računati od krajših podnizov proti daljšim in si jih shranjevati

v neko tabelo, odkoder jih potem lahko prebiramo, kadarkoli jih spet potrebujemo. Tako pridemo do naslednje rešitve s časovno zahtevnostjo $O(n^3)$:

```
#include <stdio.h>

#define MaxN 1000

/* f[i][j] bo najmanjše potrebno število brisanj, s katerim
   lahko popolnoma pobrišemo niz s[i...j-1]. */
char s[MaxN + 3];
int f[MaxN + 1][MaxN + 1];

int main()
{
  int i, j, d, n, naj, kand;

  /* Preberimo vhodni niz. */
  FILE *g = fopen("brisanje.in", "rt");
  fscanf(g, "%d\n", &n);
  fgets(s, sizeof(s), g); fclose(g);

  /* Pri podnizih dolžine 0 ni treba nobenih brisanj. */
  for (i = 0; i <= n; i++) f[i][i] = 0;

  /* Daljše podnize obdelajmo po naraščajoči dolžini. */
  for (d = 1; d <= n; d++) for (i = 0; i + d <= n; i++)
  {
    /* Kako najceneje pobrisati s[i...i+d-1]? Ena možnost je,
       da pobrišemo s[i] samega zase in nato brišemo s[i+1...i+d-1]. */
    naj = 1 + f[i + 1][i + d];
    for (j = i + 1; j < i + d; j++) if (s[j] == s[i])
    {
      /* Lahko pa poskušamo s[i...i+d] pobrisati tako, da tisto brisanje,
         ki pobriše s[i], pobriše tudi s[j] in nobenega od vmesnih znakov.
         Vmes moramo torej s[i+1...j-1] pobrisati v celoti, cena brisanja tega, kar
         ostane — to je s[i] + s[j...i+d-1] — pa je enaka kot cena brisanja
         s[j...i+d-1] samega po sebi, ker sta znaka s[i] in s[j] enaka in bomo lahko
         s[i] pobrisali z istim brisanjem, ki pobriše tudi s[j]. */
      kand = f[i+1][j] + f[j][i + d];
      if (kand < naj) naj = kand;
    }
    f[i][i + d] = naj;
  }
  /* Izpišimo rezultat. */
  g = fopen("brisanje.out", "wt"); fprintf(g, "%d\n", f[0][n]); fclose(g); return 0;
}
```

Za kratke nize, kakršni so bili pri našem tekmovanju pri 40 % testnih primerov, pa bi deloval dovolj hitro tudi kakšen preprost rekurzivni postopek, ki poskuša na vse možne načine izbrisati nek podniz strnjenih znakov iz niza s in nato izvede rekurzivni klic za niz, ki ostane po tem brisanju (če še ni prazen).

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA

1. Nadležne besede

Naša rešitev v zanki bere besede eno po eno, pri vsaki prebrani besedi pa gre v še eni gnezdeni zanki po znakih te besede in šteje, kolikokrat se zgodi, da se dve zaporedni črki tipkata z isto tipko (spremenljivka stNadleznih). Za ugotavljanje, s katero tipko se tipka posamezna črka, si pomagamo s tabelo tipke, v kateri kot indekse uporabimo kar položaje črk v angleški abecedi (tipke[0] se nanaša na črko a, tipke[1] na črko b in tako naprej). Najbolj nadležno besedo doslej si zapomnimo v spremenljivki najBeseda, število nadležnih parov črk v njej pa v maxNadleznih. Ko za trenutno besedo ugotovimo, koliko nadležnih parov vsebuje, moramo preveriti, če je to več kot pri najbolj nadležni besedi doslej, in če je, si trenutno besedo zapomnimo (v najBeseda in maxNadleznih).

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

const char *tipke = "22233344455566677778889999";

int main()
{
  char beseda[101], najBeseda[101];
  int i, stNadleznih, maxNadleznih = 0;
  najBeseda[0] = 0;
  while (gets(beseda))
  {
    for (i = 0, stNadleznih = 0; beseda[i]; i++)
      if (i > 0 && tipke[beseda[i − 1] − 'a'] == tipke[beseda[i] − 'a'])
        stNadleznih++;
    if (stNadleznih > maxNadleznih)
      maxNadleznih = stNadleznih, strcpy(najBeseda, beseda);
  }
  printf("%s\n", najBeseda); return 0;
}
```

2. Prepisovanje

Števila, ki so jih napisali naši učenci, berimo po vrsti v zanki; pri tem si v spremenljivki prejsnji zapomnimo število prejšnjega učenca, da ga bomo lahko primerjali s številom trenutnega učenca in tako videli, če ju moramo osumiti prepisovanja. Pri tem pazimo na to, da ko smo pri prvem učencu (i == 0 v spodnjem programu), prejšnjega učenca še ni.

Če vidimo, da sta trenutni in prejšnji učenec oddala preveč podobni števili, ju osumimo prepisovanja; to pomeni, da moramo ustrezno povečati števec osumljenih (spremenljivka stOsumljenih). Povečati ga moramo vsaj za 1, da zajamemo dejstvo, da je trenutni učenec zdaj osumljen (doslej pa ni bil); preveriti pa moramo še, ali je bil prejšnji učenec osumljen že prej (npr. ker je bilo njegovo število preblizu števila predprejšnjega učenca) ali ne. Če prej še ni bil osumljen, moramo števec osumljenih zdaj povečati tudi zaradi njega, drugače pa ne (da ne bi istega učenca šteli dvojno). Torej potrebujemo podatek o tem, ali je bil prejšnji učenec že osumljen ali ne; v ta namen ima spodnji program spremenljivko prejOsumljen.

Ko končamo z obdelavo trenutnega učenca, si podatke o njem zapomnimo v spremenljivkah prejsnji in prejOsumljen, da bomo pripravljeni na obdelavo naslednjega učenca (tedaj bo učenec, ki je zdaj še trenutni, postal prejšnji).

Na koncu imamo v spremenljivki stOsumljenih želeni rezultat, torej skupno število osumljenih učencev, in ga moramo le še izpisati.

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

int main()
{
  int n, t, stOsumljenih = 0, i, prejsnji, trenutni;
  bool prejOsumljen;
  scanf("%d %d", &n, &t);
  /* Pojdimo v zanki po vseh učencih. */
  for (i = 0; i < n; i++)
  {
    scanf("%d", &trenutni);
    /* Ali se številka trenutnega in prejšnjega učenca premalo razlikujeta? */
    if (i > 0 && abs(prejsnji - trenutni) <= t)
    {
      /* Trenutni učenec je zdaj osumljen. */
      stOsumljenih++;
      /* Če prejšnji še ni bil osumljen, je zdaj osumljen tudi on
         in moramo števec osumljenih povečati tudi zaradi njega. */
      if (! prejOsumljen) stOsumljenih++;
      /* Ko bo trenutni učenec v naslednji iteraciji postal prejšnji,
         si zapomnimo, da je bil osumljen. */
      prejOsumljen = true;
    }
    /* Sicer pa trenutni učenec zaenkrat ni osumljen;
       zapomnimo si to za naslednjo iteracijo. */
    else prejOsumljen = false;
    /* Zapomnimo si številko trenutnega učenca za v naslednjo iteracijo. */
    prejsnji = trenutni;
  }
  /* Izpišimo rezultat. */
  printf("%d\n", stOsumljenih);
  return 0;
}
```

3. Riziko

Podatke o metih posameznih kock lahko shranimo v dve tabeli — tabela a (s tremi elementi) za prvega igralca in tabela b (z dvema elementoma) za drugega igralca. Zdaj moramo obe tabeli urediti padajoče; pri tem bi načeloma lahko uporabili katerega koli od številnih znanih postopkov urejanja (ali pa kakšno funkcijo za urejanje iz standardne knjižnice našega programskega jezika — v C-ju je to na primer qsort); ker pa sta obe tabeli tako zelo majhni in je njuna velikost znana vnaprej, ju lahko uredimo tudi s pomočjo čisto preprostega razmisleka. Pri tabeli b, ki ima le dva elementa, moramo le preveriti, če je prvi element manjši od drugega, in če je, ju zamenjamo.

Pri tabeli a s tremi elementi je stvar podobna, le malo bolj zapletena: če je $a[0] < a[1]$, ju zamenjamo; po tem vemo, da je manjši od prvih dveh elementov zdaj v $a[1]$; zdaj lahko primerjamo $a[1]$ z $a[2]$ in vemo, da bo manjši od teh dveh tudi najmanjši element cele tabele, ta pa mora biti na koncu tabele. Če se torej izkaže, da je $a[1] < a[2]$, ju moramo zamenjati, sicer pa lahko $a[2]$ ostane tam, kjer je bil. Zdaj vemo, da je v $a[2]$ najmanjši element tabele, v $a[0]$ in $a[1]$ pa sta ostala dva elementa, ki pa še nista nujno v pravem vrstnem redu, zato ju moramo še enkrat primerjati in po potrebi zamenjati med sabo.

Ko sta tako obe tabeli urejeni, moramo le še primerjati istoležne elemente — $a[0]$ z $b[0]$ ter $a[1]$ z $b[1]$ — in ustrezno dodeliti točke obema igralcema. Spodnji program šteje točke prvega igralca v spremenljivki ta, točke drugega pa v tb. Na koncu moramo le še izpisati vrednosti obeh spremenljivk.

```
#include <stdio.h>

int main()
{
  int a[3], b[2], t, ta, tb, i;
  scanf("%d %d %d %d %d %d", &a[0], &a[1], &a[2], &b[0], &b[1]);
  /* Uredimo kocke prvega igralca padajoče. */
  if (a[0] < a[1]) t = a[1], a[1] = a[0], a[0] = t;
  if (a[1] < a[2]) t = a[2], a[2] = a[1], a[1] = t;
  if (a[0] < a[1]) t = a[1], a[1] = a[0], a[0] = t;
  /* Uredimo kocki drugega igralca padajoče. */
  if (b[0] < b[1]) t = b[1], b[1] = b[0], b[0] = t;
  /* Primerjajmo istoležne kocke obeh igralcev. */
  ta = 0; tb = 0;
  for (i = 0; i < 2; i++)
    if (a[i] > b[i]) ta += 2;
    else if (a[i] < b[i]) tb += 2;
    else ta += 1, tb += 1;
  /* Izpišimo rezultate. */
  printf("Prvi igralec dobi %d točk, drugi pa %d.", ta, tb);
  return 0;
}
```

4. Eksplozija

Označimo koordinate središča eksplozije z (x_e, y_e). Opazimo lahko, da se predmeti, ki so že pred eksplozijo ležali levo od te točke ($x < x_e$), pri eksploziji premaknejo še dlje proti levi; tisti, ki so ležali desno od nje ($x > x_e$), se premaknejo še dlje proti desni; tisti, ki so ležali nad točko eksplozije ($y > y_e$), se pomaknejo še bolj gor; tisti pa, ki so ležali pod točko eksplozije ($y < y_e$), se pomaknejo še bolj navzdol

Oglejmo si zdaj enega od šestnajstih enotskih kvadratkov naše mreže 4×4. Pred eksplozijo je v vsakem od njegovih štirih oglišč stal po en predmet. Kje so zdaj ti predmeti po eksploziji?

Če naš kvadratek leži levo zgoraj od tistega, v katerem je prišlo do eksplozije, se je tisti predmet, ki je bil prej v spodnjem desnem kotu kvadratka, premaknil gor in levo, torej zdaj leži v notranjosti našega kvadratka (ker je dolžina premika le 1/2, je nemogoče, da bi se tak predmet pomaknil čez naš kvadratek — za kaj takega bi moral biti premik daljši od 1).

Podoben razmislek lahko opravimo tudi v primerih, ko leži naš kvadratek desno zgoraj, levo spodaj ali desno spodaj od tistega, v katerem pride do eksplozije. V vsakem primeru torej nek predmet po eksploziji leži v notranjosti našega kvadratka.

Če pa naš kvadratek leži v istem stolpcu kot tisti, v katerem je prišlo do eksplozije, to pomeni, da sta se od predmetov, ki so pred eksplozijo ležali v njegovih ogliščih, leva dva premaknila v levo, desna dva pa v desno, torej zdaj nobeden od njih ne leži v notranjosti našega kvadratka. Podoben razmislek lahko uporabimo tudi v kvadratkih, ki ležijo v isti vrstici kot tisti, v katerem je prišlo eksplozije.

Opazimo lahko tudi, da ni mogoče, da bi se po eksploziji v notranjosti nekega kvadratka nahajal kakšen tak predmet, ki pred eksplozijo ni ležal v enem od oglišč tega kvadratka. Vsi ostali predmeti so namreč od našega kvadratka oddaljeni za vsaj 1 dolžinsko enoto, premaknejo pa se le za 1/2 enote, torej našega kvadratka ne morejo niti doseči, kaj šele se premakniti v njegovo notranjost.

Nalogo lahko torej rešimo tako, da za vsako vrstico in vsak stolpec mreže pogledamo, ali po eksploziji v njej leži kakšen predmet ali ne. Pri natanko eni vrstici in enem stolpcu bomo opazili, da ne vsebuje nobenega predmeta; potem vemo, da je do eksplozije prišlo v središču kvadratka, ki leži na preseku te vrstice in tega stolpca.

Zapišimo našo rešitev še v C-ju. V zanki bomo brali koordinate točk in vsako zaokrožili navzdol na celo število, da vidimo, v notranjosti katere vrstice oz. stolpca leži ta točka. Nato povečamo ustrezni števec točk za tisto vrstico in stolpec (tabeli vrstice in stolpci). Ko na ta način obdelamo vse točke, moramo le še poiskati v vsaki od teh dveh tabel element z vrednostjo 0.

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

int main()
{
  int vrstice[4], stolpci[4], i, xx, yy;
  double x, y;

  /* Inicializirajmo števce v obeh tabelah na 0. */
  for (xx = 0; xx < 4; xx++) stolpci[xx] = 0;
  for (yy = 0; yy < 4; yy++) vrstice[yy] = 0;
  /* Preberimo koordinate vseh 25 predmetov. */
  for (i = 0; i < 25; i++)
  {
    scanf("%lf %lf", &x, &y);
    /* V kateri vrstici in stolpcu leži zdaj ta predmet? */
    xx = (int) floor(x); yy = (int) floor(y);
    /* Povečajmo števce (če leži zunaj mreže, ga ignorirajmo). */
    if (0 <= xx && xx < 4) stolpci[xx]++;
    if (0 <= yy && yy < 4) vrstice[yy]++;
  }
  /* Poglejmo, v kateri vrstici in stolpcu ni nobenega predmeta. */
  for (xx = 0; xx < 4; xx++) if (stolpci[xx] == 0) x = xx + 0.5;
  for (yy = 0; yy < 4; yy++) if (vrstice[yy] == 0) y = yy + 0.5;
  /* Izpišimo rezultat. */
  printf("Do eksplozije je prišlo v točki (%.1f, %.1f).\n", x, y);
  return 0;
}
```

Razmislimo še o posplošenih različicah naloge, ki jih omenja opomba na koncu besedila naloge.

(*a*) Če nas zanima samo to, v katerem kvadratku mreže je prišlo do eksplozije, lahko uporabimo enak postopek kot za prvotno nalogo. Bolj zanimiva pa naloga postane, če hočemo tudi izračunati položaj eksplozije in pokriti tudi primere, ko do eksplozije pride zunaj mreže.

Naše predmete lahko glede na njihov položaj pred eksplozijo razdelimo na 5 stolpcev s po 5 predmeti v vsakem stolpcu. Kaj lahko povemo o predmetih v stolpcu, ki je imel pred eksplozijo x-koordinato a? Če je $a < x_e$, ležijo x-koordinate teh predmetov po eksploziji na intervalu $[a - 1/2, a)$; če je $a > x_e$, ležijo na $(a, a + 1/2]$; sicer pa so njihove koordinate po eksploziji še vedno enake a.

Če izračunamo takšen interval pri dveh sosednjih stolpcih, recimo a in $a + 1$, vidimo, da nastaneta v vsakem primeru dva disjunktna intervala. Z drugimi besedami, predmeti iz različnih stolpcev se ob eksploziji ne morejo premešati med sabo. Če predmete po eksploziji uredimo po njihovi x-koordinati, bo prvih 5 mest v tem vrstnem redu pripadlo predmetom iz najbolj levega stolpca, naslednjih 5 mest predmetom iz drugega najbolj levega stolpca in tako naprej.

Podoben razmislek lahko opravimo tudi po vrsticah, kjer namesto x-koordinat gledamo y-koordinate. Zdaj torej znamo za vsak predmet po eksploziji povedati, v kateri vrstici in stolpcu je ležal pred eksplozijo; z drugimi besedami torej poznamo njegove koordinate tudi pred eksplozijo, ne le po njej. Ker se pri eksploziji premakne predmet v ravni črti stran od točke eksplozije, to pomeni, da ležijo točka eksplozije, prvotni položaj predmeta in njegov novi položaj na isti premici. Če skozi položaj predmeta pred eksplozijo in po njej speljemo premico, gre ta premica gotovo tudi skozi točko eksplozije. To naredimo za vsak predmet in dobimo 25 premic, ki gredo vse skozi točko eksplozije; izračunati moramo torej le presečišče poljubnih dveh izmed njih in je točka eksplozije. (Pravzaprav se lahko zgodi, da pri več predmetih nastane ena in ista premica, zato moramo paziti, da za računanje presečišča vzamemo dve različni premici.) V praksi bi bilo, da se zmanjša težave zaradi morebitnih zaokrožitvenih napak pri računanju, koristno izračunati presečišče za več parov premic in potem vrniti povprečje (centroid) tako dobljenih presečišč.

(*b*) Podobno kot pri (*a*) lahko ugotovimo, kateri vrstici in stolpcu pripada posamezni predmet; če označimo vrstice z $r = 0, \ldots, 4$ in stolpce s $c = 0, \ldots, 4$, je predmet na preseku vrstice r in stolpca c pred eksplozijo imel koordinati (ru, cu), težava pa je, da u-ja ne poznamo.

Recimo v splošnem, da gledamo nek predmet i, ki je imel pred eksplozijo koordinati (x_i, y_i), po njej pa $(\hat{x}_i, \hat{y}_i)$. Vemo, da se je predmet pri eksploziji premikal v ravni črti stran od točke eksplozije, torej v smeri $(\triangle x_i, \triangle y_i)$ za $\triangle x_i = x_i - x_e$ in $\triangle y_i = y_i - y_e$; dolžina tega premika pa je bila $u/2$. Stari in novi položaj sta torej povezana z enačbama

$$\hat{x}_i = x_i + (u/2)\triangle x_i/(\triangle x_i^2 + \triangle y_i^2)^{1/2} \text{ in}$$
$$\hat{y}_i = y_i + (u/2)\triangle y_i/(\triangle x_i^2 + \triangle y_i^2)^{1/2}.$$

Prvo od njiju lahko predelamo v

$$(\hat{x}_i - x_i)(\triangle x_i^2 + \triangle y_i^2)^{1/2} - (u/2)\triangle x_i = 0,$$

podobno pa tudi drugo.

Ta razmislek lahko opravimo pri vsakem predmetu in tako dobimo 50 enačb. Ko upoštevamo še, da za vsak predmet poznamo (r, c) in da sta njegovi koordinati (pred eksplozijo) oblike $x_i = ru$, $y_i = cu$, vidimo, da so v tem sistemu enačb neznanke le x_e, y_e in u. Rešujemo ga lahko numerično; če izraze na levi strani enačb označimo z X_i in Y_i, lahko sistem rešujemo tako, da iščemo minimum funkcije $J(x_e, y_e, u) = \sum_i (X_i^2 + Y_i^2)$. To lahko počnemo na primer z gradientnim spuščanjem.

Da se ne bomo zaplezali v kakšen lokalni ekstrem, je koristno še malo razmisliti o tem, kako izbrati začetni nabor vrednosti (x_e, y_e, u), iz katerega bomo potem začeli z gradientnim spuščanjem.

Če je bila razdalja med dvema predmetoma pred eksplozijo d, je po eksploziji ta razdalja nekje na območju $[d, d + u]$. Pred eksplozijo je za katerikoli par predmetov veljalo, da sta bila vsaj za u narazen; to velja torej tudi po eksploziji; zato lahko kot zgornjo mejo za u (in kot primerno začetno vrednost pri gradientnem spuščanju) vzamemo minimalno oddaljenost dveh predmetov po eksploziji (ta minimum izračunamo po vseh možnih parih predmetov).

Za začetno vrednost x_e in y_e pa lahko razmišljamo takole. Ker za vsak predmet vemo, kateri vrstici in stolpcu pripada, lahko pogledamo za vsak kvadratek naše prvotne mreže štiri predmete iz oglišč tega kvadrata: (r, c), $(r, c + 1)$, $(r + 1, c)$ in $(r + 1, c + 1)$. Za možno začetno vrednost (x_e, y_e) vzemimo povprečje koordinat teh štirih predmetov po eksploziji, torej nekaj, kar gotovo leži bolj ali manj znotraj tega kvadrata. To naredimo za vseh 16 kvadratov naše mreže in na koncu obdržimo tisti (x_e, y_e), ki je dal (v kombinaciji z malo prej izbranim u-jem) najmanjšo vrednost funkcije J.

(c) Razmišljamo lahko podobno kot pri (b). Recimo, da je naša mreža zasukana za kót α in zamaknjena za zamik (x_0, y_0) od koordinatnega izhodišča. Položaj (pred eksplozijo) predmeta na presečišču vrstice r in stolpca c je bil torej $x_i = x_0 + cu\cos\alpha - ru\sin\alpha$ in $y_i = y_0 + cu\sin\alpha + ru\cos\alpha$. Od tu naprej lahko s podobnim razmislekom kot pri (b) dobimo sistem enačb, le da je v njem zdaj 6 neznank (x_0, y_0, x_e, y_e, u, α).

Vprašanje je le, kako ugotoviti, kateri vrstici in stolpcu pripada posamezni predmet; ne moremo jih preprosto urediti po eni od koordinat, kot smo storili pri (b), ker je mreža zdaj zasukana za nek kót α, ki ga ne poznamo. Če bi α poznali in zasukali mrežo nazaj za kot $-\alpha$, bi prišli v podobno situacijo kot pri (b): prvih pet predmetov po x-koordinati tvori en stolpec, naslednjih pet tvori drugi stolpec in tako naprej. Pri tem bi tudi enako kot takrat veljalo, da vsak stolpec pokriva interval x-koordinat, ki je širok manj kot $u/2$, med dvema stolpcema pa je razmik, ki je širok vsaj $u/2$ (kjer ni nobenega predmeta). Z drugimi besedami, če si mislimo predmete oštevilčene naraščajoče po x-koordinati, bodo razmiki $x_5 - x_1$, $x_{10} - x_6$ itd. vsi manjši od razmikov $x_6 - x_5$, $x_{11} - x_{10}$ itd. Podobno bo veljalo tudi za y-koordinate. Če kaj od tega dvojega ne drži, to pomeni, da smo vzeli napačno vrednost α.

Še drugače lahko razmislek iz prejšnjega odstavka opišemo tako, da smo vzeli premico, ki gre pod kotom α (glede na pozitivno x-os), projicirali naše predmete nanjo in jih uredili po vrsti od leve proti desni glede na to, kam na premico je padla njihova projekcija; nato pa smo vsako skupino petih predmetov v tem vrstnem redu

projicirali še na premico pod kotom $\alpha+90^\circ$ in jih uredili glede na to, kam je njihova projecija padla zdaj. Ker je predmetov končno mnogo, je tudi možnih vrstnih redov, ki jih na ta način dobimo, le končno mnogo. Možnih kotov α pa je neskončno; torej je dovolj, če se ukvarjamo le s tistimi koti α, pri katerih pride do spremembe v opisanem vrstnem redu predmetov. Začnemo lahko z $\alpha = 0$, nato pa na vsakem koraku povečamo α do prvega naslednjega kota, pri katerem se vrstni red projekcij na premici pod kotom α ali $\alpha + 90^\circ$ kaj spremeni. Tega, pri kateri α se vrstni red projekcij dveh točk zamenja, pa ni težko ugotoviti: za poljubni dve točki U in V pride do spremembe v vrstnem redu njunih projekcij na premico pod kotom α takrat, ko je slednja ravno pravokotna na premico skozi U in V.

Zdaj torej znamo poiskati neko α, ki sicer še ni nujno čisto tisti kót, za katerega je bila zasukana naša mreža, je pa vseeno dovolj dober, da znamo predmete razdeliti na vrstice in stolpce. To α lahko vzamemo tudi kot začetno vrednost pri našem gradientnem spuščanju; za začetno vrednost x_0 in y_0 lahko vzamemo koordinati (po eksploziji) predmeta v spodnjem levem kotu mreže ($r = c = 0$), začetne vrednosti x_e, y_e in u pa lahko izberemo tako kot pri (b).

5. Barvanje plošče

Brez izgube za splošnost lahko predpostavimo, da je začetni položaj plošče enak $z = 0$. Če ni, je vse, kar moramo narediti, to, da tabelo pobarvaj ciklično zamaknemo za z mest navzdol: izsek, ki je bil prej na indeksu z, je zdaj na indeksu 0; tisti, ki je bil prej na indeksu $(z+1)$ mod n, je zdaj na indeksu 1 in tako naprej. V nadaljevanju bomo torej predpostavili, da je $z = 0$.

Na število operacij Pobarvaj ne moremo kaj dosti vplivati: vsak izsek, ki bo moral biti v končnem stanju plošče pobarvan, moramo pobarvati vsaj enkrat (na primer takrat, ko pridemo prvič do njega), nobene koristi pa ni od tega, da bi ga pobarvali več kot enkrat. Število izvajanj operacije Pobarvaj bo torej kar enako številu izsekov, ki jih je treba pobarvati črno. Tega ni težko izračunati tako, da se v zanki zapeljemo po tabeli pobarvaj in štejemo vrednosti **true**.

Vplivamo lahko torej le na število zasukov plošče. Rekli bomo, da pri našem barvanju plošče *obiščemo* nek izsek, če se ta izsek vsaj enkrat znajde pod barvno glavo. Ker lahko ploščo vedno obračamo le za en izsek levo ali desno, to pomeni, da obiskani izseki tvorijo neko strnjeno skupino — vsi obiskani izseki se držijo skupaj v enem kosu, ravno tako pa se tudi vsi ostali, neobiskani izseki držijo skupaj v enem kosu. Med obiskanimi izseki je seveda vedno tudi izsek 0, ki je pod barvno glavo že na začetku postopka.

Obiskano območje lahko torej opišemo preprosto tako, da povemo, pri katerem izseku se začne in pri katerem se konča. Recimo, da nazaj od $z = 0$ obsega izseke $n-1, n-2, \ldots, n-a$ (če nazaj od $z = 0$ nismo obiskali sploh nobenega izseka, si mislimo $a = 0$), naprej od $z = 0$ pa izseke $1, 2, \ldots, b$ (če naprej od $z = 0$ nismo obiskali sploh nobenega izseka, si mislimo $b = 0$). To je seveda smiselno le, če je $b < n - a$, saj se drugače začneta levi in desni del prekrivati, kar ni smiselno, ker že pri $b = n - a - 1$ obiščemo čisto vse izseke na plošči. Da bo rešitev sploh lahko veljavna, mora tudi veljati, da nobenega od odsekov $b+1, b+2, \ldots, n-a-1$ ni treba pobarvati (saj jih med obračanjem plošče nikoli ne bomo dosegli in jih torej tudi pobarvati ne bi mogli).

Pri danih a in b je, kar se tiče samega barvanja, vseeno, v kakšnem vrstnem redu obiščemo te izseke, saj jih bomo v vsakem primeru lahko pobarvali (vsak izsek, ki ga je treba pobarvati, lahko pobarvamo takrat, ko prvič pridemo do njega). Smiselno je torej, da si zaporedje zasukov izbremo tako, da bomo vse te izseke dosegli s čim manj zasuki plošče. Ena možnost je na primer, da najprej izvedemo b zasukov v levo (pri tem pridejo pod barvno glavo po vrsti izseki od 1 do b) in nato še $b+a$ zasukov v desno (s čimer pridejo pod barvno glavo po vrsti izseki $b-1, \ldots, 0, n-1, \ldots, n-a$). Druga možnost je, da najprej izvedemo a zasukov v desno in nato $a+b$ zasukov v levo. Med tema dvema možnostma vzemimo tisto, ki zahteva manj zasukov — to bo $\min\{2a+b, a+2b\}$ zasukov.

Ali obstaja še kakšno krajše zaporedje zasukov, s katerim bi lahko obiskali vse te izseke? Razmišljamo lahko takole: vsaj enkrat med našim sukanjem plošče moramo obiskati izsek $n-a$ in vsaj enkrat izsek b. Ločimo dve možnosti: (1) mogoče obiščemo izsek $n-a$ prej kot izsek b. Ker je na začetku naša plošča na položaju 0, izseka $n-a$ ne moremo obiskati prej kot v a zasukih od začetka postopka; ko pa smo enkrat na izseku $n-a$, od tam ne moremo priti do izseka b v prej kot v $a+b$ zasukih. Postopek tega tipa torej gotovo izvede vsaj $2a+b$ zasukov. (2) Mogoče pa obiščemo izsek b prej kot izsek a. Podoben razmislek kot pri (1) nam zdaj pokaže, da tak postopek gotovo izvede vsej $a+2b$ zasukov. Če ugotovitvi (1) in (2) združimo, vidimo, da je nemogoče, da bi dosegli vse izseke od $n-a$ do b z manj kot $\min\{2a+b, a+2b\}$ zasuki. Razpored iz prejšnjega odstavka pa je porabil natanko toliko zasukov, torej boljšega razporeda ni.

Vprašanje je še, kako naj si izberemo a in b. Recimo, da je $b > 0$ in da izseka b ni treba pobarvati črno. V tem primeru ni nobene koristi od tega, da izsek b sploh obiščemo — število b lahko zmanjšamo za 1 in potrebno število premikov, $\min\{2a+b, a+2b\}$, se lahko ob tem le zmanjša ali ostane enako, gotovo pa se ne more povečati. Podobno velja tudi, če je $a > 0$ in $n-a$ ni treba pobarvati črno — takrat lahko zmanjšamo a za 1 in rešitve gotovo ne poslabšamo.

Vidimo torej, da je smiselno za b izbrati le tako številko izseka, ki ga je treba pobarvati črno (poleg tega pa še $b = 0$, kar pokrije možnost, da ploščo od začetnega položaja vrtimo le v desno); in da je potem za a smiselno izbrati število naslednjega izseka (od $b+1$ do $n-1$), ki ga je treba pobarvati črno; če pa takega ni, lahko vzamemo $a = n$. Med vsemi tako dobljeni pari (a, b) bomo uporabili tistega, ki nam dá najmanjšo vrednost $\min\{2a+b, a+2b\}$.

Zapišimo našo rešitev še v C-ju (če odmislimo del, ki bi moral v primerih, ko je prvotni z različen od 0, zamakniti tabelo pobarvaj in potem postaviti z na 0):

```
int KolikoOperacij(int n, bool pobarvaj[])
{
  int stBarvanj = 0, stZasukov = n, na, a, b;
  /* Preštejmo, koliko barvanj potrebujemo. */
  for (b = 0; b < n; b++) if (pobarvaj[b]) stBarvanj++;
  if (stBarvanj == 0) return 0;
  /* Preglejmo vse primerne b in izračunajmo potrebno število zasukov. */
  for (b = 0; b < n; )
  {
    /* b = 0 je koristen tudi, če tega izseka ni treba pobarvati; s tem pokrijemo
       primere, ko sukamo ploščo le desno od začetnega položaja. */
    /* Naj bo „na“ (= n − a) naslednji izsek, ki ga je treba pobarvati. */
```

```
    na = b + 1;
    while (na < n && ! pobarvaj[na]) na++;
    a = n − na;
    /* Izračunajmo potrebno število zasukov; če je najboljše doslej,
       si ga zapomnimo. */
    if (2 * a + b < stZasukov) stZasukov = 2 * a + b;
    if (a + 2 * b < stZasukov) stZasukov = a + 2 * b;
    /* Premaknimo se na naslednji izsek, ki ga je treba pobarvati. */
    b = na;
  }
  return stBarvanj + stZasukov;
}
```

Naloge so sestavili: riziko — Nino Bašić; tipkanje, H-indeks — Tomaž Hočevar; sorodstvo — Boris Horvat; virus — Vid Kocijan; asteroidi — Jurij Kodre; letala — Matjaž Leonardis; izštevanka, suhi dnevi — Mark Martinec; dominosa, eksplozija — Mitja Lasič; zoom — Matija Lokar; analiza enot, za žužke gre, prepisovanje — Jure Slak; galaktična zavezništva — Patrik Zajec; zaklepajski izrazi, brisanje niza, nadležne besede, barvanje plošče — Janez Brank.

REŠITVE NEUPORABLJENIH NALOG IZ LETA 2014

1. Ganttov diagram

Čez vhodne podatke se moramo sprehoditi dvakrat: prvič, da izračunamo, kako širok mora biti naš diagram (kako dolge vrstice potrebujemo), in drugič, da ga tudi zares izpišemo. Če se podproces konča ob času k in ima ime, dolgo n znakov, se bo v diagramu zaporedje znakov `#` končalo v stolpcu k, nato bo v stolpcu $k+1$ presledek in v $k+2$ se bo začelo ime; ker je dolgo n znakov, se bo torej končalo v stolpcu $k+n+1$. To izračunamo za vse procese v naših vhodnih podatkih in si zapomnimo maksimum po vseh procesih. Na koncu dolžino še zaokrožimo navzgor do prvega večkratnika števila 10, da se bo diagram na desnem robu končal z navpično črto, tako kot kaže tudi primer v besedilu naloge.[10]

Ko vemo, kako širok diagram potrebujemo, lahko izpišemo glavo diagrama, torej vodoravno črto (iz znakov `'-'`) s številkami na vsakih 10 stolpcev. Vsako število pretvorimo v niz, pogledamo, kako dolg niz je nastal, in potem pred njim izpišemo toliko minusov, da bo s tistim nizom vred nastalo ravno 10 znakov.

Nato se postavimo nazaj na začetek vhodne datoteke in jo preberemo še enkrat, pri tem pa ob vsakem procesu izpišemo ustrezno izhodno vrstico. Recimo, da se proces začne ob času z, konča ob času k in ima ime, dolgo n znakov. Z zanko po x najprej izpišemo prvih z znakov vrstice (v stolpcih od 0 do $z-1$); vsak od teh znakov je bodisi `'|'` (če je x večkratnik 10; tako nastanejo navpične črte na diagramu) bodisi preseledek. Nato izpišemo znake # v stolpcih od z do k, nato pa še presledek (v stolpcu $k+1$) in ime procesa (v stolpcih od $k+2$ do $k+n+1$). Ostanejo nam še stolpci od $k+n+2$ do konca vrstice, v katerih izpisujemo znake `'|'` in presledke, enako kot na začetku vrstice.

Oglejmo si implementacijo te rešitve v C-ju:

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

int main()
{
  int casOd, casDo, dolzina = 0, x, n;
  char s[101];
  FILE *f = fopen("gantt.txt", "rt");
  /* Preberimo vhodno datoteko in poglejmo, kako dolge vrstice potrebujemo v izpisu. */
  while (3 == fscanf(f, "%d %d %s", &casOd, &casDo, s))
  {
    x = casDo + 1 + strlen(s);
    if (x > dolzina) dolzina = x;
  }
  /* Zaokrožimo dolžino navzgor na večkratnik 10. */
  dolzina = 10 * ((dolzina + 9) / 10);
  /* Izpišimo glavo diagrama. */
```

[10] Vprašanje je še, ali zadnja črka imena kakšnega procesa lahko sega v ta skrajni desni stolpec diagrama (in prekine navpično črto v njem) ali ne. Besedilo naloge tega ne določa natančno. Naša rešitev to dovoli, če pa bi hoteli to preprečiti, bi morali spremenljivko `dolzina` pred zaokrožanjem še povečati za 1.

```
    putchar('0');
    for (x = 1; x <= dolzina / 10; x++)
    {
      sprintf(s, "%d", x * 10); n = strlen(s);
      while (n++ < 10) putchar('-');
      printf("%s", s);
    }
    putchar('\n');
    /* Preberimo vhodno datoteko še enkrat in izpišimo preostanek diagrama. */
    rewind(f);
    while (3 == fscanf(f, "%d %d %s", &casOd, &casDo, s))
    {
      for (x = 0; x < casOd; x++) putchar(x % 10 == 0 ? '|' : ' ');
      for (x = casOd; x <= casDo; x++) putchar('#');
      printf(" %s", s); x += 1 + strlen(s);
      for ( ; x <= dolzina; x++) putchar(x % 10 == 0 ? '|' : ' ');
      putchar('\n');
    }
    fclose(f); return 0;
}
```

2. Cezar

Vhodno datoteko berimo po vrsticah in si prebrane besede shranjujmo v neko primerno podatkovno strukturo. Pri vsaki besedi izračunajmo, v kaj se zašifrira po postopku ROT13, in poglejmo, če smo na to šifrirano različico besede v vhodnih podatkih doslej že naleteli; če smo, izpišimo obe besedi.

Besede bi lahko na primer shranjevali v seznamu ali tabeli, vendar bi se morali potem pri preverjanju, ali je neka (zašifrirana) beseda že v seznamu ali ne, zapeljati z zanko po vseh že prebranih besedah, kar postane neučinkovito, če je besed veliko (pri omejitvah iz besedila naloge to sicer ne bi bil problem, saj tam piše, da bo besed kvečjemu tisoč). Boljša rešitev, če jo naš programski jezik podpira (ali pa smo si jo pripravljeni napisati sami), je razpršena tabela (*hash table*), pri kateri za preverjanje, ali je neka beseda v tabeli ali ne, porabimo le konstantno veliko časa ne glede na to, koliko besed je že v njej. Primerna podatkovna struktura je na primer set v pythonu, unordered_set v C++, HashSet v C# in javi ipd.

Oglejmo si implementacijo takšne rešitve v pythonu:

```
def rot13(s):
    a = ord('a')
    return "".join(chr(a + ((ord(c) - a + 13) % 26)) for c in s)

besede = set()
for s in open("besede.txt"):
    s = s.strip() # pobrišimo morebitni znak za konec vrstice
    t = rot13(s)
    if t in besede: print("%s %s" % (s, t))
    else: besede.add(s)
```

Pri šifriranju niza po postopku ROT13 smo si pomagali s funkcijama ord in chr, ki pretvarjata znake v njihove številske kode (po standardu Unicode in nazaj. Malim črkam angleške abecede pri tem pripadajo številke od 97 do 126 (v abecednem vrstnem redu); tako lahko z ord(c) − ord('a') dobimo položaj črke c v abecedi (od

0 do 25), nato pa jo šifriramo tako, da ji prištejemo 13 in obdržimo le ostanek po deljenju s 26 (tako da iz številk $13, \ldots 25$ nastanejo $0, \ldots 12$ in ne $26, \ldots, 38$). Številke znakov po šifriranju pretvorimo nazaj v črke in jih staknemo skupaj (funkcija join).

3. Digitalna ura

Naša rešitev hrani opise števk v tabeli: opisStevk[d][s] pove stanje segmenta s pri prikazu števke d (zvezdica pomeni prižgan segment, pika pa ugasnjenega). Podprogram Osvezi najprej izračuna, kakšen je bil čas eno minuto pred sedanjim (običajno se minuta zmanjša za 1, ura pa se ne spremeni; paziti pa moramo na primere, kot so začetek ure in začetek dneva), nato pa za oba časa (prejšnjega in sedanjega) izračuna stanje vseh štirih števk na zaslonu ure (tabeli d in dPrej). Nato se z dvema zankama zapelje po vseh štirih števkah, pri vsaki števki pa po vseh sedmih segmentih; če mora biti nek segment v novem stanju drugačen kot v prejšnjem, pokličemo zanj funkcijo Spremeni. (Pri tem pazimo še na to, da ta funkcija pričakuje, da se številke segmentov in števk na zaslonu začnejo z 1, za preostanek naše rešitve pa je bolj prikladno, da jih začnemo z 0.)

```
extern void Spremeni(int stevka, int segment);

const char *opisiStevk[10] = {
  "xxx.xxx", "..x..x.", "x.xxx.x", "x.xx.xx", ".xxx.x.",
  "xx.x.xx", "xx.xxxx", "x.x..x.", "xxxxxxx", "xxxx.xx" };

void Osvezi(int h, int m)
{
  int hPrej = (m > 0) ? h : (h == 0) ? 23 : h - 1;
  int mPrej = (m > 0) ? m - 1 : 59;
  int d[4] = { h / 10, h % 10, m / 10, m % 10 };
  int dPrej[4] = { hPrej / 10, hPrej % 10, mPrej / 10, mPrej % 10 };
  int i, j;
  for (i = 0; i < 4; i++) for (j = 0; j < 7; j++)
    if (opisiStevk[d[i]][j] != opisiStevk[dPrej[i]][j])
      Spremeni(i + 1, j + 1);
}
```

4. Funkciji

Obe funkciji računata potenco a^b. Recimo, da prvotno vrednost parametrov a in b označimo z A oz. B. Slednjega lahko zapišemo v dvojiškem zapisu kot $B = (b_t b_{t-1} \ldots b_1 b_0)_2 = \sum_{k=0}^{t} b_k 2^k$ (pri tem predpostavimo, da je $b_t = 1$, torej da smo t izbrali tako, da v dvojiškem zapisu b-ja ni nepotrebnih vodilnih ničel).

Prva funkcija deluje takole: spomnimo se, da je potenca vsote enaka produktu potenc. Eksponent B lahko izrazimo kot vsoto nekaj potenc števila 2, torej kot $B = \sum_k b_k 2^k$. Zato je potenca A^B naprej enaka $A^{\sum_k b_k 2^k} = \prod_k A^{b_k 2^k}$. Člen $A^{b_k 2^k}$ je seveda pri $b_k = 0$ enak 1, pri $b_k = 1$ pa je enak A^{2^k}, zato lahko naš produkt poenostavimo v $A^B = \prod_k A^{2^k}$, če si predstavljamo, da gre tu k le po tistih bitih, ki so v dvojiškem zapisu B-ja prižgani (torej kjer je $b_k = 1$).

Glavna zanka naše funkcije vsakič kvadrira spremenljivko a, spremenljivko b pa zamakne za en bit v desno. Če je bilo izvedenih že k iteracij, imamo na začetku

naslednje iteracije torej $a = A^{2^k}$ in $b = (b_t b_{t-1} \ldots b_{k+1} b_k)_2 = \lfloor B/2^k \rfloor$. Ko s pogojem „**if** (b & 1)“ preverimo, če je najnižji bit v b prižgan, je to enako, kot če bi preverili, ali je prižgan bit k v prvotnem B. Če je prižgan, pomnožimo c z vrednostjo A^{2^k}, ki jo trenutno hranimo v spremenljivki a. Tako se sčasoma v c nabere zmnožek vrednosti A^{2^k} točno za tiste k, pri katerih je $b_k = 1$; ta zmnožek pa je, kot smo videli zgoraj, ravno enak A^B.

Za lažjo predstavo o delovanju funkcije jo zapišimo še enkrat, pri čemer tokrat k dodajmo vanjo eksplicitno in jo dopolnimo s komentarji o tem, kakšne so na začetku posamezne iteracije vrednosti spremenljivk:

```
int Prva(int a, int b)
{
  int c = 1, k = 0;
  while (b > 0)
  {
    /* Na tem mestu velja invarianta: a = A^(2^k), b = (b_t ... b_k)_2, c = A^((b_{k-1} ... b_0)_2). */
    if (b & 1) c *= a;
    a *= a; b >>= 1; k++;
  }
  /* Na tem mestu velja gornja invarianta, poleg tega pa je k = t + 1.
     Zato je b = 0 in c = A^B. */
  return c;
}
```

Druga funkcija pa temelji na naslednji ideji: recimo, da smo že izračunali potenco A^r za nek eksponent r. Kaj se zgodi, če r-ju v dvojiškem zapisu na koncu pritaknemo še eno ničlo? Nova vrednost eksponenta je torej $2r$, nova vrednost potence pa je $A^{2r} = (A^r)^2$, torej jo dobimo tako, da prejšnjo potenco kvadriramo. In podobno, kaj se zgodi, če r-ju v dvojiškem zapisu na koncu pritaknemo še eno enico? Nova vrednost eksponenta je $2r + 1$, nova vrednost potence pa $A^{2r+1} = (A^r)^2 \cdot A$, torej moramo staro vrednost kvadrirati in jo nato še pomnožiti z A.

Če zančnemo z $r = 0$ (in $A^r = 1$) in temu r-ju v dvojiškem zapisu na desni po vrsti dodajamo bite $b_t, b_{t-1}, \ldots, b_1, b_0$ ter pri tem sproti popravljamo A^r, kot smo videli v prejšnjem odstavku, bo na koncu tega postopka r enak B in naša potenca A^r bo ravno enaka A^B.

Naša funkcija najprej poišče najmanjšo tako potenco števila 2, ki je večja od B: na začetku postavi d na 1, nato pa ga zamika po en bit v levo, torej d po vrsti dobiva vrednost 2, 4, 8, 16 in tako naprej, dokler ne preseže B-ja; ker smo rekli, da je b oblike $b = (b_t \ldots b_0)_2$, se bo zanka ustavila pri $d = 2^{t+1}$.

Druga zanka gre potem z d-jem od tam navzdol po potencah števila 2. Recimo, da je v neki iteraciji $d = 2^k$. Na začetku te iteracije je $c = A^r$ za $r = (b_t \ldots b_{k+1})_2$. V tej iteraciji bi radi r-ju na desni pritaknili bit b_k in ustrezno popravili c. Videli smo, da bo zato treba c v vsakem primeru kvadrirati, nato pa ga, če je $b_k = 1$, še pomnožiti z a. To, ali je bit k v številu b prižgan ali ne, pa preverimo s pogojem „**if** (d & b)“ — ker je $d = 2^k$, je v d-ju prižgan le bit k in noben drug, zato je vrednost d & b enaka 0, če je v b bit k ugasnjen, sicer pa je enaka 2^k.

Ta zanka se konča, ko d pade na 0, kar pomeni, da je bil na začetku zadnje izvedene iteracije še enak 1, torej 2^k za $k = 0$, torej je bil na koncu te iteracije r že enak $(b_t \ldots b_0)_2 = B$, torej imamo zdaj v c res vrednost A^B.

Podobno kot pri prvi funkciji si oglejmo še dopolnjeno in komentirano različico druge funkcije:

```
int Druga(int a, int b)
{
  int c = 1, d = 1, k = 0;
  while (d <= b)
  {
    /* Na tem mestu velja: d = 2^k. */
    d <<= 1; k++;
  }
  while (d > 0)
  {
    /* Na tem mestu velja: d = 2^k, c = A^r za r = (b_t ... b_{k+1})_2 = ⌊B/2^{k+1}⌋. */
    c *= c;
    /* Zdaj je c = A^r za r = (b_t ... b_{k+1}0)_2. */
    if (d & b) c *= a;
    /* Zdaj je c = A^r za r = (b_t ... b_{k+1}b_k)_2. */
    d >>= 1; k--;
  }
  /* Tu je d = 0, k = -1 in c = A^B (kar je tudi A^r za r = (b_t ... b_{k+1})_2). */
  return c;
}
```

5. 3-d tiskalnik

Podprtost preverjamo po plasteh od spodaj navzgor. V najnižji plasti ($z = 0$) so podprte vse prisotne kockice. V višjih plasteh pa razmišljamo takole: za začetek označimo kot podprte vse tiste kockice, ki so prisotne v plasti z in ki imajo v plasti $z-1$ podprto sosedo. Ko je tako nekaj kockic v plasti z podprtih, lahko tudi njihove sosede v isti plasti označimo za podprte, nato sosede teh sosed in tako naprej. To lahko naredimo z iskanjem v širino: dodamo podprte kockice v vrsto, na vsakem koraku pa potem vzamemo eno kockico iz vrste, pregledamo njene sosede v plasti z in če še niso označene kot podprte (so pa prisotne v predmetu, ki ga hočemo natisniti), jih zdaj označimo kot podprte in dodamo še njih v vrsto. Tako sčasoma dosežemo vse podprte kockice v trenutni plasti; nato se le še enkrat sprehodimo čez celo plast in preverimo, če je kje prisotna kakšna nepodprta kockica. Zapišimo to rešitev še v C-ju:

```
#include <stdbool.h>
#define X ...
#define Y ...
#define Z ...

const int DX[4] = { 1, -1, 0, 0 }, DY[4] = { 0, 0, 1, -1 };
bool T[X][Y][Z];

bool Preveri()
{
  int x, y, z, d, xx, yy;
  bool podprto[X][Y];
  int vrsta[X * Y], glava, rep;

  /* V najnižji plasti so podprte vse kockice, ki so v njej sploh prisotne. */
  for (x = 0; x < X; x++) for (y = 0; y < Y; y++) podprto[x][y] = T[x][y][0];
```

```
  /* Računajmo podprtost višjih plasti od spodaj navzgor. */
  for (z = 1; z < Z; z++)
  {
    /* V plasti z so za začetek podprte vse tiste kockice, ki so v njej prisotne
       in ki so bile podprte že v plasti z - 1. Dodajmo jih v vrsto. */
    glava = 0; rep = 0;
    for (x = 0; x < X; x++) for (y = 0; y < Y; y++)
      if (! T[x][y][z]) podprto[x][y] = false;
      else if (podprto[x][y]) vrsta[rep++] = x * Y + y;
    /* Z iskanjem v širino določimo še vse ostale podprte kockice v tej plasti. */
    while (glava < rep)
    {
      x = vrsta[glava] / Y; y = vrsta[glava++] % Y;
      /* Kockica (x, y, z) je podprta; preglejmo njene sosede v plasti z in
         označimo tudi njih za podprte. */
      for (d = 0; d < 4; d++)
      {
        xx = x + DX[d]; yy = y + DY[d];
        if (xx < 0 || xx >= X || yy < 0 || yy >= Y) continue;
        if (T[xx][yy][z] && ! podprto[xx][yy])
          podprto[xx][yy] = true, vrsta[rep++] = xx * Y + yy;
      }
    }
    /* Preverimo, če je v plasti z prisotna kakšna nepodprta kockica. */
    for (x = 0; x < X; x++) for (y = 0; y < Y; y++)
      if (T[x][y][z] && ! podprto[x][y]) return false;
  }
  /* Če smo prišli do konca, vemo, da so vse kockice v mreži podprte. */
  return true;
}
```

6. Kontrolne naloge

Razdelimo pri vsakem predmetu naloge na skupine po n nalog (tem bomo rekli velike skupine), na koncu pa ostane mogoče še ena manjša skupina. Naj bo v skupno število velikih skupin, m pa število manjših skupin. Gotovo velja $m \leq p$, ker nastane pri vsakem od p predmetov kvečjemu ena manjša skupina. Uredimo te skupine naraščajoče po velikosti; naj bo a_i velikost i-te med njimi. Imamo torej zaporedje $0 < a_1 \leq a_2 \leq \ldots \leq a_m < n$. Naj bo $A_i = a_1 + \ldots + a_i$ skupno število nalog v prvih i manjših skupinah.

Recimo, da si ogledamo nek konkretni razpored nalog med dijake in da dobri dijaki rešijo naloge, ki so jim bile dodeljene, v t dneh, slabi dijaki pa svoje naloge v t' dneh. Čas reševanja za razpored kot celoto je torej $\max\{t, t'\}$ dni. Toda če je $t' > t$, lahko razpored spremenimo tako, da dobri dijaki $(t+1)$-vi dan rešijo nekaj nalog, ki bi jih drugače morali rešiti slabi dijaki na t'-ti dan; zaradi tega se čas reševanja pri slabih dijakih spremeni s t' na t'', pri čemer gotovo velja $t'' \leq t'$ (čas reševanja se lahko zmanjša ali ostane nespremenjen, gotovo pa se ne more povečati). Skupni čas reševanja je torej zdaj $\max\{t+1, t''\}$; oba člena sta $\leq t'$, zato je tudi $\max\{t+1, t''\} \leq t' \leq \max\{t, t'\}$. Tako torej vidimo, da novi razpored ni nič slabši od prvotnega, lahko je celo še boljši od njega. Če pri novem razporedu velja $t'' > t+1$, lahko prevalimo na dobre dijake še nekaj več nalog in tako pridemo do razporeda (ki

tudi ni nič slabši od prvotnega), pri katerem dobri dijaki rešujejo $t+2$ dni, slabi pa t''' dni (za nek $t''' \leq t''$). Tako lahko nadaljujemo, dokler ne pridemo do razporeda, v katerem dobri dijaki porabijo vsaj toliko dni kot slabi. Ker se pri nobeni od teh sprememb razpored ni poslabšal, to pomeni, da se lahko pri iskanju najboljšega možnega razporeda omejimo na take razporede, pri katerih dobri dijaki rešujejo vsaj toliko dni kot slabi.

Ko delimo naloge med dijake, je smiselno dati manjše skupine slabim dijakom, večje pa dobrim. O tem se prepričamo takole: mislimo si razpored, v katerem so neko skupino velikosti u rešili slabi dijaki, neko manjšo skupino velikosti v (za nek $v < u$) pa en dober dijak; tak razpored lahko zdaj izboljšamo oz. ga vsaj ne poslabšamo, če dobremu dijaku damo skupino u, skupino v pa prepustimo slabim dijakom: dobri dijak v vsakem primeru svojo skupino nalog reši v enem dnevu, slabi dijaki pa za v nalog porabijo manj časa kot za u nalog, tako da bodo lahko zdaj v istem času rešili celo še dodatnih $u - v$ nalog iz kakšne druge skupine.

Opazimo lahko tudi, da ni nobene koristi od tega, da neko skupino nalog razdelimo malo med slabe in malo med dobre dijake; dober dijak lahko reši celo skupino v enem dnevu, torej ni nobene koristi od tega, da bi se ukvarjal le z nekaj nalogami te skupine. Vsako skupino nalog bomo torej bodisi v celoti prepustili slabim dijakom ali pa v celoti enemu dobremu dijaku.

Pri razporejanju nalog med dijake lahko zdaj ločimo dva tipa razporedov: ena možnost je, da slabi dijaki rešujejo le naloge iz manjših skupin (vse naloge iz velikih skupin pa rešijo dobri dijaki), druga pa je, da rešijo tudi kakšno nalogo iz kakšne od velikih skupin (tistih s po n nalogami).

Oglejmo si najprej razporede prvega tipa. Recimo, da slabi dijaki rešijo vse naloge iz prvih r skupin, ostale naloge pa ostanejo dobrim dijakom. Slabi dijaki morajo torej rešiti A_r nalog, za kar porabijo $\lceil A_r/s \rceil$ dni; dobri dijaki pa morajo rešiti $v + (m - r)$ skupin nalog, za kar porabijo $\lceil (v + m - r)/d \rceil$ dni. Skupni čas reševanja nalog je maksimum od tega dvojega. Preizkusiti moramo vse r od 0 do m in si zapomniti najmanjši skupni čas reševanja po vseh teh r.

Pri razporedih drugega tipa pa, kot smo rekli, slabi dijaki rešijo vse naloge iz manjših skupin in mogoče še kakšno iz velikih. Recimo, da dobri dijaki rešujejo naloge t dni; v tem času torej rešijo $t \cdot d$ velikih skupin, tako da za slabe dijake ostane $m + (v - td)$ skupin ($v - td$ velikih skupin, ki jih ne rešijo dobri dijaki, in še vseh m manjših skupin) s skupno $A_m + (v - td)n$ nalogami. Za te naloge torej slabi dijaki porabijo $t' = \lceil x \rceil$ dni za $x = (A_m + (v - td)n)/s$. Zgoraj smo že videli, da se smemo omejiti na razporede, pri katerih je $t \geq t'$; ta pogoj je izpolnjen natanko tedaj, ko velja $t \geq x$; tako imamo $t \geq (A_m + (v - td)n)/s$, kar lahko z nekaj premetavanja predelamo v $t \geq (A_m + vn)/(s + dn)$. Najmanjši možni t je torej načeloma $t = \lceil (A_m + vn)/(s + dn) \rceil$ in to je potem tudi najkrajši čas, v katerem je mogoče rešiti vse naloge z razporedom drugega tipa.

Paziti pa moramo še na naslednje: lahko se zgodi, da je tako dobljeni t prevelik. V t dneh rešijo dobri dijaki $t \cdot d$ velikih skupin, vseh velikih skupin pa je le v. Največji možni t, pri katerem sploh še dobimo razpored drugega tipa, je torej $\lfloor v/d \rfloor$; če je naš t večji od tega, to pomeni, da bodo dobri dijaki imeli dovolj časa, da rešijo vse velike skupine (in mogoče še kakšno manjšo), torej naš razpored sploh ne spada več pod drugi tip, ampak pod prvega. V takem primeru tudi vemo, da pri vseh razporedih,

ki so res drugega tipa, porabijo slabi dijaki več dni kot dobri, za take razporede pa smo že zgoraj videli, da jih lahko brez škode ignoriramo.

Zapišimo tako dobljeni postopek še s psevdokodo:

$m := 0$; $v := 0$;
$a :=$ prazen seznam; (* *a bodo skupine z manj kot n nalogami* *)
for $i := 1$ **to** p:
  $v := v + \lfloor k_i/n \rfloor$;
  if $k_i \bmod n \neq 0$ **then** dodaj ($k_i \bmod n$) v seznam a in povečaj m za 1;
uredi seznam a naraščajoče;
v tem vrstnem redu označimo njegove elemente z $a_1, a_2, \ldots, a_m$;
$A_0 := 0$; **for** $r := 1$ **to** m **do** $A_r := A_{r-1} + a_r$;

$t^\star := \infty$;
for $r := 0$ **to** m:
  (* *Kaj če slabi dijaki rešijo vse naloge iz prvih r skupin, dobri pa vse ostale?* *)
  $c := \max\{\lceil A_r/s \rceil, \lceil (v + m - r)/d \rceil\}$;
  if $c < t^\star$ **then** $t^\star := c$;

(* *Upoštevajmo še razporede drugega tipa.* *)
$t := \lceil (A_m + v \cdot n)/(s + d \cdot n) \rceil$;
if $t \cdot d \leq v$ **and** $t < t^\star$ **then** $t^\star := t$;

return $t^\star$;

Kakšna je časovna zahtevnost tega postopka? Če odmislimo čas urejanja seznama a, ima vse ostalo v tem postopku časovno zahtevnost $O(p)$. Čas, ki ga potrebujemo za urejanje seznama a, pa je odvisen od tega, kakšen postopek uporabimo; šlo bi na primer v $O(p \log p)$ časa (quicksort, heapsort ipd.), v $O(p \log \log n)$ časa (van Emde Boasovo drevo) ali v $O(n)$ časa (urejanje s štetjem). V vsakem primeru pa vidimo, da ni ta postopek nič počasnejši od tistega, s katerim smo pri prvotni različici naloge (tisti s šolskega tekmovanja 2015) le preverili, če je za nek konkretni t mogoče rešiti vse naloge v največ t dnevih.

7. Hišna številka

(*a*) Naivna rešitev, ki povečuje n po 1 in vsakič preveri, ali je številka zdaj že primerna (torej taka, ki se ob vrtenju za 180 stopinj ne spremeni), je veliko prepočasna, saj imamo pri tej nalogi opravka s števili, ki imajo lahko kar milijon števk.

Recimo, da je vhodno število n dolgo k števk. Če ne bomo našli primernega rezultata (recimo mu N) s k števkami, potem lahko vsekakor najdemo primernega s $k+1$ števkami, namreč $100\ldots001$. Razmislimo zdaj o morebitnem rezultatu s k števkami.

Ena možnost je, da je k sod, torej $k = 2m$. Če izberemo zgornjih m števk našega števila N, bo s tem tudi spodnjih m števk enolično določenih (zaradi omejitve, da se mora N ob obratu za 180° preslikati sam vase). Recimo, da se bo N ujemal z n-jem v zgornjih t števkah, nato pa nastopi neujemanje. To neujemanje mora seveda biti takšno, da ima N tam večjo števko kot n; torej če ima na primer n tam 4, potem ima N lahko 6, 8 ali 9, ne pa 0 ali 1. Čim povzročimo takšno neujemanje, vemo, da bo N večji od n, ne glede na to, kaj se zgodi pri nižjih števkah. Da bo N čim

manjši, postavimo torej vse nižje števke na 0 (v prvi polovici N-ja, spodnja polovica pa je nato tako ali tako enolično določena z zgornjo).

Vprašanje je zdaj, kakšen t vzeti. Večji t pomeni, da se N ujema z n-jem v več najvišjih števkah, torej je N za manj večji od n-ja, kot bi bil, če bi imeli manjši t. Torej si želimo čim večji t (ker hočemo čim manjši N). Če gremo po n-ju od zgornjih števk proti nižjim (ves čas znotraj gornje polovice števila), se vsekakor moramo ustaviti, čim v n-ju opazimo kakšno od števk 2, 3, 4, 5 in 7, kajti N bo tam *moral* imeti kaj drugega (ker te števke ob vrtenju za 180° postanejo neveljavne); tisto je potem naš t.

Če pa v zgornji polovici n-ja te števke ne nastopajo, lahko načeloma pustimo gornjo polovico n-ja čisto pri miru, torej rečemo, da bo v N-ju enaka; nato izračunamo spodnjo polovico N-ja (kot za 180° zasukano sliko gornje) in preverimo, če je tako dobljen N večji od n; če je, je to odgovor, ki ga iščemo. Če pa ni, potem moramo v gornji polovici N-ja eno števko vendarle povečati, seveda čim nižjo. Števke 9 ne moremo povečati, katerokoli drugo pa lahko. Če se torej gornja polovica N-ja konča na $\ldots d9999\ldots9$, pri čemer je $d < 9$, potem povečajmo d na naslednjo primerno števko ($0 \to 1 \to 6 \to 8 \to 9$), vse devetke desno od njega pa spremenimo v ničle. Nato spet izračunamo še spodnjo polovico N-ja kot zasukano sliko zgornje. Če pa so v gornji polovici N-ja same devetke, nam ne bo ostalo drugega, kot da za rezultat vzamemo za eno števko daljše število od n-ja (in seveda oblike $100\ldots001$, da bo N najmanjši možni).

Doslej smo razmišljali o primeru, ko je k sod. Pri lihem k, recimo $k = 2m + 1$, je razmislek podoben, le na srednjo števko moramo paziti posebej. Lahko jo obravnavamo kot del zgornje polovice N-ja, le da pri obračanju za 180° nič ne prispeva k spodnji polovici; poleg tega velja zanjo še dodatna omejitev, da mora biti sama svoja obrnjena slika, torej je lahko tam le 0, 1 ali 8, ne pa 6 ali 9.

(*b*) Oglejmo si za začetek primer, ko je k sod, $k = 2m$. Zadnja števka n-ja mora biti enaka zasukani kopiji prve, predzadnja zasukani kopiji druge in tako naprej. Če si torej izberemo, kakšne števke bi imeli v zgornji polovici števila, bodo tiste iz spodnje polovice s tem že enolično določene. Vsako števko v zgornji polovici si lahko izberemo na 5 načinov (dovoljene števke so namreč 0, 1, 6, 8, 9; ostale pri zasuku za 180° postanejo neveljavne), le na prvem mestu ne sme biti ničla, saj hišnih številk načeloma ne pišemo z nepotrebnimi vodilnimi ničlami. Tako torej vidimo, da lahko dobimo $4 \cdot 5^{m-1}$ primernih hišnih številk.

Pri lihem k, recimo $k = 2m + 1$, je razmislek zelo podoben, le števko na sredini števila moramo obravnavati posebej. Ta števka se mora pri zasuku preslikati samo vase, zato sme stati tam le števka 0, 1 ali 8. Tako lahko torej sestavimo $3 \cdot 4 \cdot 5^{m-1}$ primernih hišnih številk. (Ta formula predpostavlja, da zgornja in spodnja polovica številke nista prazni, torej da je $m \geq 1$. Primer, ko iščemo enomestne hišne številke, torej $k = 1$ in $m = 0$, je najbolje obravnavati posebej. Takrat je edina števka hkrati srednja in prva, zato se omejitvi, da mora biti srednja števka ena od 0, 1 ali 8, pridruži še omejitev, da prva števka ne sme biti 0. Takrat obstajata torej le dva primerna n-ja.)

8. Lov na zaklad

Nalogo si lahko predstavljamo kot problem iskanja najkrajše poti v grafu, pri čemer

ima graf po eno točko za vsako prehodno polje našega zemljevida; od točke u do točke v naj obstaja usmerjena povezava natanko tedaj, če imata pripadajoči polji skupno stranico, dolžina take povezave pa naj bo čas premika s prvega polja na drugo. V tem grafu moramo zdaj poiskati najkrajšo pot od točke s, ki predstavlja naš začetni položaj, do točke z, ki predstavlja polje z zakladom.

To lahko naredimo z Dijkstrovim algoritmom. Ta med delom vzdržuje množico točk (recimo ji Q), do katerih že poznamo neko pot iz s, ki pa še ni nujno najkrajša; dolžino najkrajše doslej znane poti od s do u hranimo v $d[u]$.[11] Na vsakem koraku vzamemo iz Q tisto točko u, ki ima (med vsemi točkami v Q) najmanjšo vrednost $d[u]$; pokazati je mogoče, da je za to točko $d[u]$ dolžina najkrajše poti od s do u sploh. Zdaj si lahko za vsako u-jevo sosedo v mislimo pot, ki gre od s po najkrajši poti do u in nato od tam z enim korakom do v. Če je to najkrajša doslej znana pot od s do v, si jo zapomnimo v $d[v]$; če v še ni bila v množici Q, jo zdaj vanjo tudi dodajmo.

```
1 za vsako točko u: d[u] := ∞;
2 Q := {s}; d[s] := 0;
3 while Q ni prazna:
4   vzemi iz Q tisto točko u, ki ima najmanjšo vrednost d[u];
5   za vsako u-jevo sosedo v:
6     d' := d[u] + (dolžina povezave u → v);
7     if d' ≥ d[v] then continue;
8     if d[v] = ∞ then dodaj v v Q;
9     d[v] := d';
```

Na koncu tega postopka imamo v tabeli d dolžine najkrajših poti od s do vseh ostalih točk (razen tistih, ki jih iz s sploh ne moremo doseči; pri njih je $d[u] = \infty$). Ker nas pravzaprav ne zanimajo najkrajše poti do vseh polj, ampak le do tistega z zakladom, lahko glavno zanko ustavimo takoj, ko v vrstici 4 iz vrste vzamemo točko z.

Za učinkovito implementacijo tega postopka moramo razmisliti predvsem o tem, kako predstaviti množico Q, da bomo lahko v vrstici 4 čim hitreje poiskali točko u z najmanjšo vrednostjo $d[u]$. Običajno pri Dijkstrovem algoritmu v ta namen uporabimo dvojiško kopico (*heap*), pri kateri trajajo posamezne operacije (iskanje točke z najmanjšo $d[u]$, dodajanje točke v v Q, popravek kopice po zmanjšanju $d[v]$) $O(\log n)$ časa, če je n število točk v Q. Če imamo graf z n točkami in m povezavami, bo časovna zahtevnost celotnega postopka potem znašala $O((n+m)\log n)$. V našem primeru imamo največ $w \cdot h$ točk, vsaka točka pa ima največ štiri povezave, torej bo časovna zahtevnost $O(wh \log(wh))$.

Do še malo boljše rešitve pa pridemo, če upoštevamo dejstvo, da imajo v našem grafu le omejen možen nabor dolžin. Delimo v mislih vse dolžine povezav z 10; zdaj ima vsaka povezava dolžino 2, 3 ali 6. Pokazati je mogoče, da med izvajanjem Dijkstrovega algoritma v vsakem trenutku velja, da se minimum in maksimum vrednosti

[11] Natančneje povedano: točke grafa so v vsakem trenutku razdeljene na tri skupine: (1) točke, do katerih še ne poznamo nobene poti iz s (one imajo $d[u] = \infty$ in niso v Q); (2) točke, za katere že poznamo najkrajšo pot iz s (one imajo $d[u] < \infty$ in niso v Q); (3) množica Q: za točke iz nje pa je $d[u]$ najkrajša taka pot iz s do u, ki se ves čas giblje znotraj točk iz skupine (1), čisto na koncu pa naredi korak od ene take točke do u.

$d[u]$ po vseh točkah iz Q razlikujeta največ za toliko, kolikor je dolga najdaljša povezava v grafu. V našem primeru to pomeni, da je v vsakem trenutku za točke $u \in Q$ možnih le 7 različnih vrednosti $d[u]$. Množico Q lahko torej predstavimo s 7 seznami: za vsako možno dolžino l imamo seznam S_l, ki vsebuje vse tiste $u \in Q$, ki imajo $d[u] = l$. Sezname je koristno predstaviti kot dvojno povezane verige (*doubly linked lists*), da bomo lahko poceni in učinkovito brisali poljubno točko iz seznama in jo dodali v drug seznam (to pride prav, ko se v vrstici 9 kakšni točki spremeni dolžina $d[v]$). Točk z najmanjšo $d[u]$ zdaj ni treba posebej iskati, saj se moramo le sprehoditi po seznamu S_l (za najmanjši tak l, pri katerem seznam ni prazen). Posamezna operacija na množici Q zdaj vzame le $O(1)$ časa, zato je časovna zahtevnost celotnega postopka le še $O(wh)$.

Oglejmo si še implementacijo te rešitve v C-ju. Vsakemu kvadratku (x, y) našega zemljevida ustreza točka grafa s številko $y + wx$. Prvo točko seznama S_d hranimo v prva[d % M] (za $M = 7$), za vsako točko u pa hranimo prejšnjo in naslednjo točko njenega seznama v prej[u] in nasl[u], dolžino najkrajše doslej znane poti od s do u pa v dolz[u]. Vrstni red točk v posameznem seznamu ni pomemben.

```
#include <stdio.h>

#define MaxW 100
#define MaxH 100
#define M 7

int w, h, prva[M];
int prej[MaxH * MaxW], nasl[MaxH * MaxW], dolz[MaxH * MaxW];
char z[MaxH][MaxW + 2];
const int dx[4] = { 1, -1, 0, 0 }, dy[4] = { 0, 0, -1, 1 }, dt[4] = { 3, 3, 2, 6 };

int main()
{
  int x, y, xx, yy, zacetek, zaklad, d, dd, dp, stVSeznamih, u, v, i;
  /* Preberimo vhodne podatke. */
  FILE *f = fopen("zaklad.in", "rt"); fscanf(f, "%d %d\n", &h, &w);
  zacetek = -1; zaklad = -1;
  for (y = 0; y < h; y++)
  {
    fgets(z[y], w + 2, f);
    for (x = 0; x < w; x++) {
      u = y * w + x; dolz[u] = -1; prej[u] = -1; nasl[u] = -1;
      if (z[y][x] == '$') zaklad = u; else if (z[y][x] == '*') zacetek = u; }
  }
  fclose(f);
  /* Na začetku so vsi seznami prazni; nato dodajmo začetno polje v seznam 0. */
  for (d = 0; d < M; d++) prva[d] = -1;
  prva[0] = zacetek; dolz[zacetek] = 0;
  stVSeznamih = 1; d = 0;
  /* Pregledujmo graf, dokler ne izpraznimo vseh seznamov. */
  for (d = 0; stVSeznamih > 0; d++) while (prva[d % M] >= 0)
  {
    /* Vzemimo eno točko iz trenutnega seznama. */
    u = prva[d % M]; x = u % w; y = u / w; stVSeznamih--;
    if (u == zaklad) { stVSeznamih = 0; break; }
    prva[d % M] = nasl[u];
```

```
    /* Preglejmo njene sosede. */
    for (i = 0; i < 4; i++)
    {
      xx = x + dx[i]; yy = y + dy[i];
      /* Preverimo, če sosednje polje (xx, yy) res obstaja, je prehodno
         in do njega še ne poznamo enako dobre ali boljše poti kot skozi (x, y). */
      if (xx < 0 || xx >= w || yy < 0 || yy >= h) continue;
      if (z[yy][xx] == '#') continue;
      v = yy * w + xx; dp = dolz[v]; dd = d + dt[i];
      if (dp >= 0 && dd >= dp) continue;
      /* Našli smo novo najkrajšo pot (dolžine dd) do polja v = (xx, yy).
         Če je „v" že v nekem seznamu (za prejšnjo dolžino dp), ga vzemimo iz njega. */
      if (dp < 0) stVSeznamih++;
      else {
        if (prej[v] >= 0) nasl[prej[v]] = nasl[v]; else prva[dp % M] = nasl[v];
        if (nasl[v] >= 0) prej[nasl[v]] = prej[v]; }
      /* Postavimo „v" v seznam, ki ustreza novi dolžini dd. */
      prej[v] = -1; nasl[v] = prva[dd % M]; prva[dd % M] = v;
      dolz[v] = dd;
    }
  }
  /* Izpišimo rezultat. */
  f = fopen("zaklad.out", "wt"); fprintf(f, "%d\n", dolz[zaklad] * 10); fclose(f); return 0;
}
```

9. Minsko polje

Celicam, ki ne vsebujejo mine, bomo rekli *proste* celice. To, ali je pri nekem danem razporedu min mogoče priti preko travnika ali ne, lahko preverjamo na primer z iskanjem v širino. Preveriti moramo, ali je kakšna od prostih celic v najbolj spodnji vrstici mreže dosegljiva iz kakšne od prostih celic v najbolj zgornji vrstici mreže z zaporedjem vodoravnih in navpičnih premikov. Na začetku označimo vse proste celice v najbolj zgornji vrstici kot dosegljive in jih dodamo v vrsto Q; nato pa v vsakem koraku vzamemo eno celico iz vrste Q, označimo njene proste sosede kot dosegljive in dodamo še njih v vrsto (če pa so bile že označene kot dosegljive, jih tokrat preskočimo). Na ta način prej ali slej obiščemo vse celice, ki so dosegljive z zgornjega roba travnika.

algoritem JeTravnikPrehoden:
vhod: tabela *prosta*, ki pove, katere celice so proste (neminirane);

for $y := 1$ **to** h **do for** $x := 1$ **to** w **do** $dosegljiva[x, y] :=$ **false**;
$Q :=$ prazna vrsta;
for $x := 1$ **to** w:
  if $prosta[x, y]$ **then**
    $dosegljiva[x, 1] :=$ **false**; dodaj $(x, 1)$ v Q;
while Q ni prazna:
  vzemi prvi element, recimo (x, y), iz vrste Q;
  za vsako celico (x', y'), ki ima z (x, y) skupno stranico:
    if $prosta[x', y']$ **and not** $dosegljiva[x', y']$ **then**
      $dosegljiva[x', y'] :=$ **true**; dodaj (x', y') v Q;

for $x := 1$ **to** w **do if** *dosegljiva*$[x, h]$ **then return true**;
return false;

V resnici bi lahko glavno zanko (**while**) ustavili že prej, namreč čim za dosegljivo razglasi razglasi kakšno celico v najbolj spodnji vrstici ($y = h$). Ne glede na to izboljšavo pa ima (v najslabšem primeru) ta postopek časovno zahtevnost $O(wh)$.

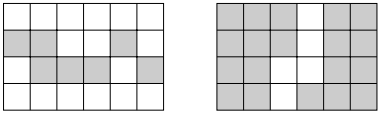
Nalogo lahko zdaj rešimo tako, da v zanki pregledujemo spisek celic, na katere naj bi postavili mine, in pri vsaki mini z iskanjem v širino preverimo, ali je travnik še vedno prehoden, če to mino dodamo nanj ali ne. Če se izkaže, da ni prehoden, mino spet pobrišemo in gremo na naslednjo po seznamu.

for $y := 1$ **to** h **do for** $x := 1$ **to** w **do** *prosta*$[x, y] :=$ **true**;
za vsako celico (x, y) s seznama celic, ki naj bi jih minirali:
  prosta$[x, y] :=$ **true**;
  (* *Z iskanjem v širino preverimo, če je še vedno mogoče prečkati travnik.* *)
  if JeTravnikPrehoden(*prosta*) **then continue**;
  prosta$[x, y] :=$ **false**;
izpiši tabelo *prosta*;

Na koncu tega postopka je v tabeli *prosta* končno stanje minskega polja in jo moramo le še izpisati, kot zahteva naloga.

Če je na poveljnikovem seznamu za miniranje L celic, je skupaj časovna zahtevnost tega postopka $O(Lwh)$. V najslabšem primeru (če je na seznamu večina celic) je $L = O(wh)$, torej je časovna zahtevnost naše rešitve skupaj $O(w^2h^2)$.

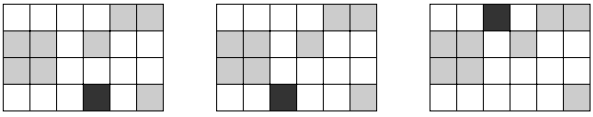
Možna pa je tudi malo bolj zapletena, vendar veliko učinkovitejša rešitev. Naloga zahteva, da preverjamo, ali obstaja pot po samih prostih celicah od zgornjega roba travnika do spodnjega, pri čemer so koraki na tej poti le vodoravni in navpični; taki poti recimo *prosta pot*. Razmišljamo lahko takole: če travnik ni prehoden (torej: če ne obstaja prosta pot), to pomeni, da mora obstajati nekakšna nepretrgana „pot“ iz samih miniranih celic od levega do desnega roba travnika; ta pot je tista, ki nam preprečuje, da bi po prostih celicah prišli od zgornjega roba do spodnjega. Taki poti iz samih miniranih celic bomo rekli *blokada*. Opozorimo pa na to, da lahko za razliko od proste poti, v kateri so bili dovoljeni le vodoravni in navpični koraki, pri blokadi nastopajo tudi diagonalni koraki: če imata dve celici skupno le eno oglišče, pa sta obe minirani, je že to dovolj, da med njima ne bomo mogli speljati naše proste poti. Primer kaže naslednja slika levo:



Po drugi strani, če prosta pot obstaja, potem gotovo ne obstaja blokada: če hočemo speljati pot po miniranih celicah od levega do desnega roba travnika, moramo nekako prečkati našo prosto pot, tega pa ne moremo, ker morata imeti na naši blokadi dve zaporedni minirani celici skupno vsaj eno oglišče, torej vmes ne moremo narediti skoka čez prosto pot. Primer vidimo na sliki zgoraj desno.

Tako torej vidimo, da obstaja prosta pot natanko tedaj, ko ne obstaja blokada. (Formalni dokaz te ugotovitve je malo bolj zapleten in si ga bomo ogledali na koncu

rešitve.) Lepo pri tem je, da je pri postopnem dodajanju min je lažje preverjati obstoj blokade kot obstoj proste poti. Že postavljene mine si lahko predstavljamo kot razdeljene na skupine, pri čemer v vsaki skupini velja, da je mogoče priti od poljubne mine do poljubne druge mine po samih miniranih celicah (in se pri tem na vsakem koraku premakniti od trenutne celice na eno od njenih sosed, ki imajo z njo skupno vsaj oglišče), ni pa mogoče na ta način priti do nobene mine v nobeni drugi skupini.[12] Ko pride na travnik nova mina, se zgodi ena od naslednjih treh stvari (glej spodnjo sliko; siva polja so obstoječe mine, črno polje je nova mina): (*a*) lahko nova mina tvori novo skupino sama zase; (*b*) lahko se priključi k eni od že obstoječih skupin; (*c*) lahko pa se zaradi nje dve skupini, ki sta bili prej ločeni, zlijeta v eno samo.



(*a*) (*b*) (*c*)

Za vsako skupino je koristno vzdrževati podatek o tem, ali vsebuje kakšno celico iz najbolj levega stolpca in ali vsebuje kakšno celico iz najbolj desnega stolpca. Takim bomo rekli *leve skupine* oz. *desne skupine*. Če je neka skupina leva in desna hkrati, to pomeni, da je mogoče v njej priti od levega do desnega roba travnika po samih miniranih celicah, torej obstaja blokada in travnik ni prehoden.

Preden na neko celico postavimo novo mino, moramo pregledati njenih osem sosed; nekatere od njih so lahko že minirane, pripadajo pa eni ali največ dvema različnima skupinama.[13] Če ena od miniranih sosed pripada neki levi skupini, druga pa neki desni skupini, potem nove mine ne smemo dodati, ker bi se s tem omenjeni dve skupini zlili v eno, v kateri bi bilo zdaj mogoče sestaviti blokado. Podobno razmišljamo v primeru, če bi naša nova mina sama stala v najbolj levem stolpcu in ima kakšno sosedo iz neke desne skupine (ali pa obratno).

Če pa do tega ne pride, lahko mino res postavimo (in nato ustrezno popravimo naše podatke o skupinah, izvedemo morebitno zlivanje skupin ipd.). Zapišimo dobljeni postopek s psevdokodo:

for $y := 1$ **to** h **do for** $x := 1$ **to** w **do** $prosta[x, y] :=$ **true**;
$S := \{\}$; (* *skupine min; na začetku ni nobene* *)
za vsako celico (x, y) s seznama celic, ki naj bi jih minirali:
  leva $:= (x = 1)$; *desna* $:= (x = w)$;
  za vsako minirano sosedo (x', y') celice (x, y):
    $s :=$ skupina (iz S), ki ji pripada celica (x', y');
    leva $:=$ *leva* **or** $s.leva$; *desna* $:=$ *desna* **or** $s.desna$;

[12] Z drugimi besedami, če si predstavljamo minirane celice kot točke grafa, v katerem sta dve točki sosednji, če imata pripadajoči celici skupno vsaj eno oglišče, potem so naše skupine min ravno povezane komponente tega grafa.

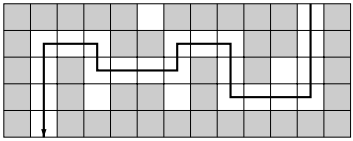
[13] Do tega, da bi imela celica, na katero poskušamo postaviti mino, minirane sosede iz treh ali več različnih skupin, ne more priti: sosedi iz različnih skupin ne smeta imeti skupnega niti oglišča, sicer bi se njuni skupini že prej morali združiti v eno; torej sta lahko sosedi celice (x, y) iz različnih skupin le tako, da sta to celici $(x \pm 1, y)$ ali pa celici $(x, y \pm 1)$. Zdaj za vsako oglišče celice (x, y) velja, da se dotika vsaj ene od teh dveh sosed; katerakoli tretja soseda celice (x, y) ima seveda z njo skupno vsaj eno oglišče, zato pa torej tudi z eno od prej omenjenih dveh sosed, torej mora pripadati eni od njunih skupin, ne pa kakšni tretji skupini.

if *leva* **and** *desna* **then continue**;
prosta$[x, y]$:= **false**;
if nismo našli nobene minirane sosede **then**
  dodaj v S skupino $\{(x, y)\}$ z atributoma *leva* in *desna*;
else if pripadajo vse minirane sosede isti skupini, recimo s **then**
  dodaj (x, y) v skupino s;
  $s.leva := s.leva$ **or** *leva*; $s.desna := s.desna$ **or** *desna*;
else
  minirane sosede pripadajo dvema skupinama, recimo jima s' in s'';
  v S združi skupini s' in s'' v eno samo, recimo ji s, in dodaj vanjo še (x, y);
  $s.leva :=$ *leva*; $s.desna :=$ *desna*;

Za učinkovito predstavitev skupin je koristno uporabiti gozd disjunktnih množic (*disjont-set forest*), podatkovno strukturo, pri kateri je časovna zahtevnost posamezne operacije (ugotavljanje, kateri skupini pripada neka minirana celica; dodajanje celice v skupino; združevanje dveh skupin) $O(\alpha(n))$, če je n število miniranih celic, pri tem pa funkcija α narašča tako počasi, da lahko v praksi to časovno zahtevnost obravnavamo kot $O(1)$.[14] Tako je zdaj časovna zahtevnost celotnega postopka le $O(wh)$.

Oglejmo si zdaj še dokaz prej omenjenih dveh trditev: če obstaja prehodna pot, potem ne obstaja blokada; če pa prehodna pot ne obstaja, potem blokada obstaja.

Recimo, da prehodna pot obstaja. Med vsemi prehodnimi potmi vzemimo najkrajšo od njih; to med drugim pomeni, da nobene celice ne obišče po večkrat in da v najbolj zgornji in najbolj spodnji vrstici obišče le eno celico. Za nadaljevanje našega razmisleka bo koristno, če si pot predstavljamo tudi kot krivuljo (lomljeno črto), ki teče po sredi celic, ki sestavljajo prehodno pot:



Postavimo se v poljubno minirano celico (x, y) in poglejmo, kaj počne naša krivulja v vrstici y levo od celice x. Mogoče je seveda, da krivulja sploh nikoli ne vstopi v ta del vrstice y. Če pa vanj nekoč vstopi, ga mora prej ali slej spet zapustiti; lahko je takih vstopov in izstopov tudi več. Pri vstopih in izstopih krivulja prečka mejo med vrstico y in eno od sosednjih vrstic, $y - 1$ in $y + 1$. Preštejmo, pri koliko vstopih se zgodi, da krivulja nato izstopi v drugo vrstico, kot je iz nje vstopila (temu bomo rekli, da je krivulja *prečkala* vrstico y). To število prečkanj označimo s $p(x, y)$; če je število prečkanj sodo, bomo rekli, da je minirana celica (x, y) tudi soda, sicer pa, da je liha.

Tako smo vpeljali pojem *parnosti* miniranih celic. (Lahko si ga predstavljamo tudi tako, da če se premikamo po krivulji od zgornjega roba travnika proti spodnjemu, so sode celice na naši desni, lihe pa na naši levi.)

Za poljubno minirano celico v najbolj levem stolpcu ($x = 1$) je očitno, da levo od nje v isti vrstici ni nobenih celic, torej krivulja tam vrstice tudi ne prečka, zato

[14] Za več o tej podatkovni strukturi gl. npr. Wikipedijo *s. v.* Disjoint-set data structure.

je $p(x, y) = 0$ in je taka celica zato soda.

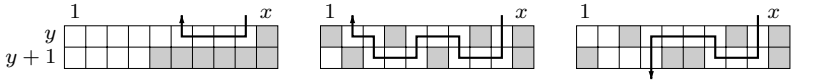
Za poljubno minirano celico v najbolj desnem stolpcu ($x = w$) pa vidimo, da so levo od nje v tej vrstici prav vsa prečkanja, ki jih krivulja izvede čez vrstico y; vsako prečkanje gre bodisi v smeri navzdol (krivulja vstopi v vrstico y iz $y - 1$ in kasneje izstopi v $y + 1$) bodisi navzgor (krivulja vstopi v y iz $y + 1$ in kasneje izstopi v $y - 1$). Ker je krivulja začela na zgornjem robu travnika, torej nad vrstico y, je moralo biti prvo prečkanje navzdol; po tistem je krivulja v vrstici $y + 1$ in ne more priti nazaj v vrstico $y - 1$ drugače kot tako, da prečka vrstico y v smeri navzgor: drugo prečkanje je torej v smeri navzgor; s podobnim razmislekom nadaljujemo in vidimo, da je vsako prečkanje v nasprotni smeri kot prejšnje. Ker pa se krivulja konča na spodnjem robu travnika, kar je pod vrstico y, mora biti zadnje prečkanje v smeri navzdol. Tako vidimo, da je število prečkanj v tem primeru zagotovo liho, torej so minirana polja v stolpcu $x = w$ liha.

Če hočemo sestaviti blokado, se mora ta začeti v stolpcu $x = 1$, torej na neki sodi celici, in končati v stolpcu $x = w$, torej na neki lihi celici. Torej bi morala blokada vsaj enkrat stopiti s sode celice na liho; spomnimo pa se, da imata dve zaporedni celici v blokadi skupno vsaj eno oglišče. Toda to je nemogoče, kajti izkaže se, da imata dve celici s skupnim ogliščem zagotovo enako parnost. Prepričajmo se zdaj o tem.

(1) Najlažji primer je, če se celici dotikata z navpično stranico: (x, y) in $(x+1, y)$. Tedaj pridejo pri ugotavljanju parnosti pri obeh celicah v poštev ista prečkanja, kajti ko gledamo območje levo od celice v isti vrstici, je to pri obeh celicah skoraj enako območje, le da je pri $(x+1, y)$ vanj vključena tudi celica (x, y); ker pa je ta minirana, krivulja ne gre skoznjo, torej na število prečkanj nič ne vpliva. Obe minirani celici imata torej enako število prečkanj in zato tudi enako parnost.

(2) Druga možnost je, da se celici dotikata z vodoravno stranico: (x, y) in $(x, y+1)$. Oglejmo si pravokotnik velikosti $(x - 1) \times 2$, ki ga sestavljajo vse celice levo od naših dveh miniranih celic. Vsakič ko krivulja vstopi v naš pravokotnik, ga mora nato prej ali slej spet zapustiti. Ko krivulja vstopi v ta pravokotnik, se to lahko zgodi na dva načina: na zgornjem robu (s premikom iz vrstice $y - 1$ v y) ali pa na spodnjem robu (s premikom iz $y + 2$ v $y + 1$). Podobno (le v obratni smeri) je tudi pri izstopih. Poglejmo, koliko prečkanj prispeva tak kos krivulje (od vstopa do izstopa) v $p(x, y)$ in $p(x, y + 1)$. (Glej primere na sliki spodaj.) (2.1) Recimo, da je krivulja vstopila in izstopila na zgornjem robu, torej na meji med vrsticama $y - 1$ in y. (2.1.1) Če je vmes (med vstopom in izstopom) ves čas ostala znotraj vrstice y, potem ni prečkala nobene od vrstic y in $y + 1$, tako da ta kos krivulje ne prispeva nobenih prečkanj. (2.1.2) Če pa se je vmes kdaj še premaknila v $y + 1$, je ob tem prečkala vrstico y (ker je vstopila vanjo iz $y-1$ in zdaj izstopila v $y+1$; toda potem se mora nekoč kasneje, preden lahko izstopi na zgornjem robu, premakniti iz $y + 1$ nazaj v y; po zadnjem takem koraku bo prišel izstop iz y v $y - 1$, kar šteje kot prečkanje vrstice y. Tako torej vidimo, da je ta kos krivulje prispeval dve prečkanji v $p(x, y)$ in nobenega v $p(x, y + 1)$. (2.2) Če krivulja vstopi in izstopi na spodnjem robu, razmišljamo podobno kot v (2.1). (2.3) Če krivulja vstopi na zgornjem robu, izstopi pa na spodnjem: po vstopu je v vrstici y, prej ali slej pa se mora prvič premakniti v $y + 1$, kar je eno prečkanje vrstice y. Nato se mogoče še večkrat premakne med $y + 1$ in y, kar ne povzroča prečkanj; prej ali slej pa se mora

iz $y+1$ premakniti v $y+2$ (da izstopi iz pravokotnika na spodnjem robu). Ker je v $y+1$ prišla iz y, šteje to kot prečkanje vrstice $y+1$. Ta kos krivulje je torej prispeval po eno prečkanje v $p(x,y)$ in v $p(x,y+1)$. (2.4) Če krivulja vstopi na spodnjem robu in izstopi na zgornjem, razmišljamo podobno kot v (2.3).



(2.1.1) ni prečkanj (2.1.2) dvakrat prečka y (2.3) enkrat prečka y, enkrat $y+1$

Vrednosti $p(x,y)$ in $p(x,y+1)$ lahko računamo tako, da na začetku obe postavimo na 0, nato pa gremo po vseh kosih krivulje znotraj našega pravokotnika (kos krivulje je del krivulje od vstopa v pravokotnik do prvega naslednjega izstopa iz njega) in pri vsakem kosu ustrezno povečamo vrednosti $p(x,y)$ in $p(x,y+1)$. Kot smo videli v prejšnjem odstavku, se lahko pri posameznem kosu povečata obe vrednosti za 1, lahko se poveča ena za 2 in druga za 0, lahko pa obe ostaneta nespremenjeni; v vsakem od teh primerov pa velja, da če sta bili pred to spremembo obe enake parnosti, sta tudi po njej še vedno obe enake parnosti. In ker sta na začetku bili obe vrednosti 0, torej enake parnosti, morata biti takšni tudi na koncu, ko dosežeta pravo vrednost $p(x,y)$ in $p(x,y+1)$. Tako torej vidimo, da sta minirani celici (x,y) in $(x,y+1)$ res enake parnosti.

(3) Mogoče je tudi, da se naši dve minirani celici dotikata le z enim ogliščem, ker je ena spodaj desno od druge, torej (x,y) in $(x+1,y+1)$. V tem primeru se lahko začasno pretvarjamo, da je na celici $(x+1,y)$ tudi mina; ta nič ne vpliva na izračun parnosti celic (x,y) in $(x+1,y+1)$, ker ne leži levo od njiju. Če je na $(x+1,y)$ mina, lahko po točki (1) sklepamo, da ima ta celica enako parnost kot (x,y), po točki (2) pa, da ima enako parnost kot $(x+1,y+1)$. Torej imata tudi (x,y) in $(x+1,y+1)$ enako parnost.

(4) Ostane še primer, da je ena od miniranih celic zgoraj levo od druge, torej $(x+1,y)$ in $(x,y+1)$. Tu razmišljamo podobno kot pri (3).

Tako torej vidimo, da se dve sosednji minirani celici (ki imata skupno vsaj eno oglišče) gotovo ujemata v parnosti, torej je nemogoče, da bi obstajal blokada, ki bi povezala kakšno od sodih celic na levem robu celice s kakšno od lihih celic na desnem robu mreže. □

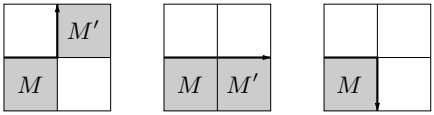
Zdaj moramo razmisliti še o drugem delu dokaza: če prehodne poti ni, potem zagotovo obstaja blokada. Trivialen primer je, da so vse celice v vrstici $y=1$ minirane; tedaj lahko blokado speljemo kar po njih. Drugače pa poženimo postopek iskanja v širino, kot smo ga videli na začetku te rešitve; dobimo neko množico dosegljivih celic, pri čemer je vsaj ena od njih v vrstici $y=1$, gotovo pa ni nobene v vrstici $y=h$ (sicer bi obstajala prehodna pot).

Da bo lažje, si mislimo, da ima travnik tudi vrstico $y=0$, v kateri ni nobene mine in so vse celice v tej vrstici dosegljive. Zdaj torej vemo, da ima vsak stolpec travnika vsaj eno prosto celico, pa tudi vsaj eno minirano (kajti če bi bile vse celice v stolpcu proste, bi ta stolpec že sam zase tvoril prosto pot čez travnik).

Naš postopek z iskanjem v širino postopoma širi dosegljivo območje tako, da vsakič doda vanj neko celico, ki ima skupno stranico z eno od že dosegljivih celic. Postopek se ustavi šele, ko to ni več mogoče, torej ko za vsako dosegljivo celico velja,

da s svojimi stranicami meji bodisi na zunanjost travnika bodisi na minirane celice bodisi na druge celice, ki smo jih tudi že prepoznali kot dosegljive. Postavimo se na najnižjo dosegljivo celico v stolpcu $x = 1$; njena spodnja soseda je gotovo minirana; sledimo zdaj robu dosegljivega območja tako, da bomo imeli na svoji levi ves čas dosegljivo celico, na svoji desni pa ves čas minirano celico. Spodnjega roba travnika na ta način gotovo ne bomo dosegli (ker bi to pomenilo, da je v najbolj spodnji vrstici neka celica dosegljiva, torej bi obstajala prehodna pot), tako da smo lahko prepričani, da celica na naši desni res vedno obstaja. Sčasoma s tem sprehodom dosežemo desni rob travnika in se ustavimo.

Oglejmo si zaporedje miniranih celic, ki smo jih imeli pri tem sprehodu na svoji desni. Recimo, da smo imeli pri nekem koraku na svoji levi minirano celico M, na naslednjem koraku pa M'. Če smo se vmes zasukali za 90° v levo, imata M in M' skupno oglišče; če smeri vmes nismo spremenili, imata M in M' skupno stranico; če pa smo se zasukali za 90° v desno, sta M in M' sploh ena in ista celica. Primer kaže naslednja slika.



Poleg tega tudi vidimo, da ker se je naš sprehod začel na levem robu travnika in končal na desnem, mora biti prva celica v našem zaporedju v stolpcu $x = 1$, zadnja pa v $x = w$. Vse to skupaj pa pomeni, da to zaporedje miniranih celic tvori blokado, prav to pa smo želeli dokazati. □

10. Speči agenti

Najprej določimo množico agentov, ki bodo sploh sodelovali v operaciji; recimo ji U. Lahko jo predstavimo kar s tabelo, v kateri za vsakega agenta piše, ali pripada množici U ali ne. Na začetku postavimo vse elemente tabele na **true**, nato pa gremo po vseh aretiranih agentih, pri vsakem od njih pa po vseh njegovih sosedih v grafu in postavljamo v tabeli U pripadajoče elemente na **false**.

za vsak $v \in V$: $U[v] :=$ **true**; *dosegljiv*$[v] :=$ **false**;
za vsak $v \in A$:
  $U[v] :=$ **false**;
  za vsakega v-jevega soseda w: $U[w] :=$ **false**;

Zdaj lahko izračunamo množico vseh ključev, ki jih ima vsaj kakšen od agentov iz U. Tudi to lahko predstavimo s tabelo logičnih vrednosti. Na začetku postavimo vse elemente na **false**, nato pa gremo z zanko po vseh agentih iz U, pri vsakem od njih pa po vseh njegovih ključih in postavljamo v tabeli pripadajoče elemente na **true**. Na koncu se sprehodim po tej tabeli in če vidimo, da je v njej še vedno kakšna vrednost **false**, lahko takoj zaključimo, da skupina U ne bo mogla izvesti tajne operacije, ker nima vseh ključev. Spotoma si v spremenljivki s še zapomnimo enega od agentov iz U (vseeno je, katerega).

za vsak $k \in K$: *imamoKlju*č$[k] :=$ **false**;
za vsak $v \in V$:

if not $U[v]$ **then continue**;
$s := u$;
za vsak ključ k agenta v: *imamoKljuč*$[k]$:= **true**;
(* *Če kakšnega ključa nima noben agent iz* U*, operacije ne bodo mogli izvesti.* *)
za vsak $k \in K$ **then return false**;

Nato moramo še preveriti, če je množica agentov U povezana. Začnimo pri agentu $s \in U$, ki smo si ga izbrali malo prej, in sledimo povezavam do njegovih sosedov, pa od teh do njihovih sosedov in tako naprej, pri čemer vedno upoštevamo le tiste sosede, ki so tudi sami v U. V tabeli *dosegljiv* si označujemo, katere agente smo na ta način že dosegli. Tiste agente, ki smo jih že dosegli, nismo pa še pregledali njihovih sosedov, hranimo v množici Q. Glavna zanka našega postopka vsakič vzame po enega agenta iz Q, pregleda njegove sosede in doda tudi njih v Q (če je iz tabele *dosegljiv* razvidno, da jih nismo videli že kdaj prej). Načeloma je vseeno, v kakšnem vrstnem redu jemljemo agente iz Q; pogost pristop je, da organiziramo Q kot vrsto, s čimer dobimo znani postopek preiskovanja v širino.

Q := prazna množica; dodaj s v Q; *dosegljiv*$[s]$:= **true**;
while Q ni prazna:
  v := poljuben element množice Q; pobriši v iz Q;
  za vsakega v-jevega soseda w:
    if *dosegljiv*$[w]$ **or not** $U[w]$ **then continue**;
    dodaj w v Q; *dosegljiv*$[w]$:= **true**;

Zdaj lahko pogledamo, če je kakšen od agentov iz U v tabeli *dosegljiv* še vedno označen kot nedosegljiv; tedaj vemo, da skupina U ni povezana in da operacije ne bo mogla izvesti, sicer pa jo bo lahko.

za vsak $v \in V$:
  if $U[v]$ **and not** *dosegljiv*$[v]$ **then return false**;
return true;

11. Načrtovanje tipkovnice

(*a*) Recimo, da imamo m tipk (ki jih oštevilčimo od 1 do t) in n črk. S p_i označimo, kolikokrat se v besedilu, ki ga hočemo natipkati, pojavi i-ta črka po abecedi. Nalogo lahko rešujemo z dinamičnim programiranjem. Zastavimo si podproblem: recimo, da bi radi prvih k črk (po abecedi) razporedili na prvih t tipk; kakšen je najboljši razpored za to oz. koliko pritiskov na tipke bo potrebnih, da natipkamo vse pojavitve teh prvih k črk v našem besedilu? To označimo s $f(k,t)$.

Na tipko t lahko razporedimo 2, 3 ali 4 črke. Če smo na t razporedili r črk, morajo biti to črke od $k-r+1$ do k, pri čemer bomo zdaj prvo od teh črk natipkali z enim pritiskom na tipko, drugo z dvema, itd., zadnjo (črko t) pa z r pritiski na tipko. Nato nam ostane še podproblem, kako prvih $k-s$ črk razporediti na prvih $t-1$ tipk. Med različnimi možnostmi za r uporabimo tisto, ki nam dá najmanjše skupno število pritiskov na tipke:

$$f(k,t) = \min_{2 \le r \le 4} \left(f(k-r, t-1) + \sum_{i=1}^{r} i \cdot p_{k-r+i} \right).$$

Robni primeri nastopijo pri $k < 2t$ ali $k > 4t$; takrat veljavnega razporeda sploh ni, ker imamo premalo črk za preveč tipk ali pa obratno, zato si lahko tam mislimo $f(k,t) = \infty$. Še en robni primer je $k = t = 0$, ko je problem trivialen in imamo $f(k,t) = 0$.

Funkcijo f lahko računamo naraščajoče po t in pri vsakem t naraščajoče po k. Že izračunane vrednosti funkcije hranimo v tabeli, tako da jih bomo imeli kasneje pri roki, ko jih bomo potrebovali za izračun f pri večjih k in t. Ob vsaki $f(k,t)$ pa hranimo še podatek o tem, pri katerem r smo jo dobili (torej koliko črk smo razporedili na tipko t pri najboljšem razporedu prvih k črk med prvih t tipk). S pomočjo teh podatkov lahko na koncu rekonstruiramo celoten najboljši razpored. Opisana rešitev porabi $O(nm)$ časa in pomnilnika.

(c) Pri tej različici naloge pride prav požrešna metoda. Spet recimo, da imamo n črk in m tipk, le da črke zdaj oštevilčimo padajoče po pogostosti in naj bo p_i število pojavitev i-te najpogostejše črke. Najpogostejših m črk razporedimo vsako na eno tipko, tako da se bodo te črke natipkale s po enim pritiskom na ustrezno tipko. Nato vzemimo naslednjih m črk in tudi njih razporedimo vsako na eno tipko, tako da se bodo te črke tipkale s po dvema pritiskoma na tipko. Tako nadaljujemo, dokler ne razporedimo vseh črk.

Mimogrede lahko opazimo, da s tem pride na posamezno tipko vsaj $\lfloor n/m \rfloor$ črk in kvečjemu $\lceil n/m \rceil$ črk; na primer, če imamo 8 tipk in 26 črk, bodo prišle na vsako tipko tri ali štiri črke. Tako smo torej rešili tudi podnalogo (b), ne le (c).

Prepričajmo se zdaj, da je tako dobljeni požrešni razpored (recimo mu R) črk med tipke res najboljši možni. Pa recimo, da je optimalen razpored (tak, ki zahteva najmanjše možno število pritiskov na tipke) nek R', ki je različen od R. Naj bo i najmanjša taka številka črke, pri kateri porabita R in R' različno število pritiskov na tipke. Za R že vemo, koliko pritiskov na tipke porabi za črko i, namreč $a_i := \lceil i/m \rceil$. Preden je R razporedil to črko na neko tipko, je že razporedil na vsako tipko vsaj $a_i - 1$ črk izmed črk $1, \ldots, i-1$; ker se v R' porabi za vsako od teh črk enako število pritiskov kot v R, to pomeni, da je tudi v R' že razporejenih vsaj $a_i - 1$ črk na vsako tipko. Če je torej R in R' pri črki i razlikujeta v številu pritiskov, se lahko to zgodi le tako, da je pri R' za črko i treba več pritiskov kot pri R; recimo, da jih je treba b_i (za nek $b_i > a_i$).

Če v R' obstaja kakšna tipka, na katero je razporejenih manj kot a_i črk, bi se dalo R' izboljšati tako, da bi na tisto tipko premaknili črko i, tako da bi se jo po novem tipkalo z a_i pritiski namesto b_i. Ker smo predpostavili, da je R' najboljši možni razpored, do tega gotovo ne more priti, torej vemo, da je v R' na vsako tipko razporejenih vsaj a_i črk. Niso pa to čisto iste črke kot pri R, saj smo malo prej videli, da se pri R natipka črko i z a_i pritiski, pri R' pa ne. Obstajati mora torej vsaj ena črka j, ki se pri R' tipka z a_i pritiski, pri R pa ne. Gotovo velja $j > i$, kajti za vse črke od 1 do $i-1$ že vemo, da se v obeh razporedih tipkajo z enakim številom pritiskov, za črko i pa vemo, da se pri R tipka za a_i pritiski.

Recimo zdaj, da v R' zamenjamo črki i in j; dobimo nov razpored R''. Zaradi te zamenjave pri črki i prihranimo $b_i - a_i$ pritiskov na tipke, pri črki j pa jih prav toliko izgubimo. Ker je $i < j$, je črka i vsaj tako pogosta kot j, torej pri tipkanju besedila kot celote s tem vsaj toliko pridobimo kot izgubimo. Razpored R'' torej ni nič slabši od R', pri tem pa se ujema z razporedom R še v enem indeksu več kot

razpored R'' (namreč črka i se pri R'' tipka z a_i pritiski, enako kot pri R). S tem razmišljanjem lahko nadaljujemo in vidimo, da lahko optimalni razpored postopoma predelamo v požrešnega, ne da bi se pri tem kdaj poslabšal; torej je tudi požrešni razpored enako dober kot najboljši možni.

12. Oddajnik

Recimo, da imamo n hiš s koordinatami (x_i, y_i) za $i = 1, \ldots, n$.

Preprosta, vendar neučinkovita rešitev te naloge je, da gremo s tremi gnezdenimi zankami po vseh možnih trojicah točk; za vsako trojico izračunamo središče in polmer krožnice skozi te tri točke, nato pa s še eno zanko preverimo, če vse ostale točke ležijo na tej krožnici ali znotraj nje. Med trojicami, pri katerih je ta pogoj izpolnjen, si zapomnimo tisto, pri kateri je polmer kroga največji. Zapišimo ta postopek s psevdokodo:

```
r* := −∞; (* največji radij doslej *)
for i := 3 to n:
  for j := 2 to i − 1:
    for k := 1 to j − 1:
      določi krožnico skozi točke (x_i, y_i), (x_j, y_j) in (x_k, y_k);
      naj bo (x_c, y_c) njeno središče, r pa njen polmer;
      if r ≤ r* then continue;
      l := 1; while l ≤ n:
        if (x_l − x_c)^2 + (y_l − y_c)^2 > r^2 then break else l := l + 1;
      if l > n then r* := r; x_c* := x_c; y_c* := y_c;
```

V spremenljivkah $x_c^\star, y_c^\star, r^\star$ torej hranimo podatke o največji doslej najdeni krožnici (izmed takih, ki pokrijejo vse hiše). Če naša nova krožnica (x_c, y_c, r) ni večja od največje doslej, se lahko z njo takoj nehamo ukvarjati, sicer pa z zanko po l preverimo, če kakšna hiša leži zunaj naše krožnice. Če take hiše ni, si krožnico zapomnimo kot novo najboljšo znano krožnico doslej.

Kako določimo središče in polmer krožnice skozi izbrane tri točke? Spomnimo se, da točka (x, y) leži na krožnici s središčem (x_c, y_c) in polmerom r natanko tedaj, ko velja $(x - x_c)^2 + (y - y_c)^2 = r^2$. V našem primeru mora to veljati za hiše i, j, k; tako dobimo:

$$\begin{aligned}(x_i - x_c)^2 + (y_i - y_c)^2 &= r^2\\ (x_j - x_c)^2 + (y_j - y_c)^2 &= r^2\\ (x_k - x_c)^2 + (y_k - y_c)^2 &= r^2.\end{aligned}$$

Če odštejemo drugo enačbo od prve in levo stran malo uredimo, dobimo

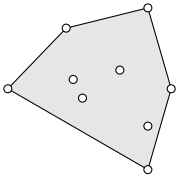
$$2(x_j - x_i)x_c + 2(y_j - y_i)y_c + (x_i^2 - x_j^2 + y_i^2 - y_j^2) = 0.$$

Podobno (le s k namesto j) dobimo tudi, če odštejemo tretjo enačbo od prve. Ker sta tu neznanki le x_c in y_c, imamo torej zdaj sistem dveh linearnih enačb z dvema neznankama, ki ga ni težko rešiti. Potem lahko vstavimo x_c in y_c v eno od prvotnih treh enačb in izračunamo še r.

Še en način, kako določiti središče krožnice, pa je tale: ker sta točki i in j enako oddaljeni od središča (namreč obe za r), to pomeni, da središče leži na simetrali

daljice med tema dvema točkama. Podoben razmislek pokaže tudi, da mora središče ležati na simetrali daljice med točkama i in k. Središče lahko torej določimo tako, da poiščemo presek obeh simetral (za to pa bomo morali tudi tu rešiti sistem dveh linearnih enačb z dvema neznankama).

Doslej opisani postopek je neučinkovit (in prepočasen za večje n), ker moramo pri njem pregledati kar $O(n^3)$ krožnic, pri nekaterih pa imamo potem še $O(n)$ dela z notranjo zanko po l. Pri razmišljanju o boljši rešitvi bo prišel prav pojem konveksne ovojnice. Za množico točk v ravnini rečemo, da je *konveksna*, če za vsak par točk iz te množice velja, da množica vsebuje tudi vse točke na daljici med tema dvema točkama. *Konveksna ovojnica* množice točk M pa je presek vseh tistih konveksnih množic, ki vsebujejo M kot podmnožico. V našem primeru bomo za M vzeli množico n točk, na katerih stojijo hiše. Ker je ta množica končna, se izkaže, da je njena konveksna ovojnica nek konveksen mnogokotnik, ki ima kot oglišča nekatere od naših hiš, ostale hiše pa ležijo v njegovi notranjosti. Primer kaže naslednja slika:



S podrobnostmi iskanja konveksne ovojnice se tu ne bomo ukvarjali; obstaja več dobro znanih algoritmov, ki lahko učinkovito poiščejo konveksno ovojnico naše množice M (z drugimi besedami: ki ugotovijo, katere hiše predstavljajo oglišča tega mnogokotnika in v kakšnem vrstnem redu).

Ena korist od konveksne ovojnice je na primer naslednja: ko preverjamo, ali neka krožnica zaobjame vse hiše, je dovolj, če ta pogoj preverimo le za hiše na robu ovojnice. Krog (torej krožnica skupaj s svojo notranjostjo) je namreč konveksna množica; če torej vsebuje hiše na robu ovojnice, vsebuje zato tudi vse točke iz notranjosti ovojnice, s tem pa torej tudi vse ostale hiše.

Še ena izboljšava pa izhaja iz naslednje lepe lastnosti konveksnih ovojnic: če imamo konveksno ovojnico neke množice M, potem poljubna točka t leži na robu te konveksne ovojnice natanko tedaj, ko obstaja neka taka premica skozi t, za katero velja, da vse točke iz M ležijo na eni strani te premice (ali pa na premici sami). (Dokaz prepustimo bralcu za vajo.)

Za našo nalogo je ta lastnost pomembna zato, ker nam pove, da si moramo tri hiše, skozi katere bomo speljali našo krožnico, nujno izbrati z roba konveksne ovojnice, ne pa iz njene notranjosti. O tem se prepričajmo takole: recimo, da si izberemo tri hiše A, B, C in speljemo skoznje krožnico, ki uspešno zaobjame vseh n hiš. Mislimo si tangento t, ki se dotika naše krožnice v točki A. Za tangento velja, da krožnica (in vse, kar je znotraj krožnice) leži na eni strani tangente; in če krožnica pokrije vse hiše, to pomeni, da tudi vse hiše ležijo na eni strani tangente (ali pa na njej — tam leži hiša A); torej po prej opisani lastnosti konveksne ovojnice leži A na robu ovojnice. Enak razmislek lahko seveda ponovimo tudi za B in C.

Tako torej ni treba pregledati vseh možnih trojic točk, ampak le tiste, pri katerih so vse tri točke z roba ovojnice. Obe izboljšavi skupaj torej povesta, da ko izračunamo konveksno ovojnico, lahko tiste hiše, ki ne ležijo na robu ovojnice, kar

pobrišemo, saj na rešitev nimajo nobenega vpliva; nato pa lahko rešujemo nalogo po enakem postopku, kot smo ga opisali na začetku.

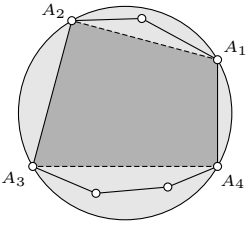
V praksi bi bila taka izboljšava že lahko koristna, v najslabšem primeru pa ni nujno, da bi bila od nje res kakšna korist; lahko se zgodi, da dobimo take vhodne podatke, v katerih vse hiše ležijo na robu konveksne ovojnice in ne moremo na podlagi dosedanjega razmisleka pobrisati nobene od njih.

Naslednja izboljšava pa je povezana s tem, da iščemo čim večjo krožnico. Neformalno lahko razmišljamo takole: če speljemo krožnico skozi točke A, B, C, bo njen polmer tem večji, čim večji je kót $\angle ABC$. Primer kaže naslednja slika levo:

Recimo, da smo točke A, B in C vzeli z roba naše konveksne ovojnice, vendar ne kot tri zaporedne točke; na primer, recimo, da je na konveksni ovojnici med A in B še neko oglišče D. Potem je kót $\angle DBC$ gotovo večji od $\angle ABC$ (saj je $\angle DBC = \angle DBA + \angle ABC$; glej desno sliko zgoraj), torej bi dobili večjo krožnico, če bi jo speljali skozi D, B in C namesto skozi A, B in C.

Tako nam lahko pride na misel, da bi namesto vseh možnih trojic točk (z roba ovojnice) gledali vedno le po tri zaporedne točke. Namesto $O(n^3)$ trojic bi morali tako pregledati le $O(n)$ trojic. Mislimo si poljubno krožnico, ki gre skozi vsaj tri točke z roba ovojnice in ki pokrije vse hiše (torej vse hiše ležijo na krožnici ali pa znotraj nje), pri tem pa krožnica ne gre skozi nobene tri zaporedne hiše z roba ovojnice. Pokazali bomo, da je mogoče vsaj tako veliko ali še večjo krožnico (ki tudi pokrije vse hiše) dobiti tudi s tremi zaporednimi hišami.

Na naši krožnici so gotovo vsaj tri hiše, lahko pa jih je tudi več; označimo jih po vrsti z $A_1, A_2, \ldots, A_k$. Daljice oblike $A_{i-1}A_i$ za $i = 1, \ldots, k$ (mislimo si še $A_0 = A_k$) so tetive te krožnice; notranjost krožnice nam razdelijo na k krožnih odsekov in na k-kotnik $A_1A_2 \ldots A_k$. Primer kaže naslednja slika:

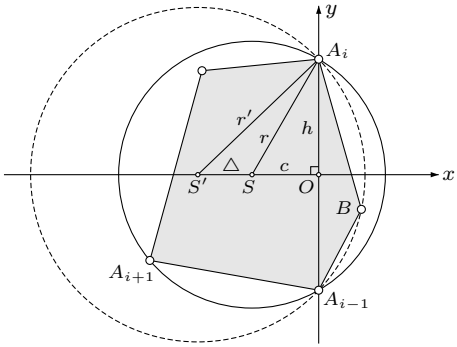


Na robu konveksne ovojnice (ki je narisan s polno črto) leži 7 točk, od tega na naši krožnici ležijo štiri: A_1, A_2, A_3, A_4. Nobene tri od njih niso zaporedne na robu ovojnice; med A_1 in A_2 je na robu ovojnice še ena točka, med A_3 in A_4 pa celo dve. Tetive $A_{i-1}A_i$ razrežejo naš krog na štiri krožne odseke (pobarvani so svetlo sivo) in štirikotnik $A_1A_2A_3A_4$ (pobarvan je temno sivo).

Gotovo se pri vsaj dveh različnih indeksih i zgodi, da točki A_{i-1} in A_i nista dve zaporedni točki na robu ovojnice (ampak je med njima na robu ovojnice še kakšna druga hiša). Kajti če se to ne bi zgodilo pri nobenem i, bi imeli na krožnici k zaporednih točk z roba ovojnice, mi pa smo predpostavili, da niso take niti tri, kaj šele k; če pa bi se to zgodilo pri enem i, recimo (brez izgube za splošnost) pri $i = 3$, bi to pomenilo, da sta A_1 in A_2 dve zaporedni točki na robu ovojnice, A_2 in A_3 pa tudi, torej imamo na krožnici že tri zaporedne točke z roba ovojnice, kar je po predpostavki spet nemogoče.

Središče našega kroga leži bodisi v mnogokotniku $A_1A_2\ldots A_k$ bodisi v enem od k odsekov, ki jih določajo tetive $A_{i-1}A_i$. Videli smo, da se pri vsaj dveh i zgodi, da A_{i-1} in A_i nista dve zaporedni točki z roba ovojnice; pri enem od njiju se mogoče izkaže, da središče kroga leži v odseku, ki ga določa tetiva $A_{i-1}A_i$, gotovo pa se to ne more zgoditi pri obeh. Izberimo torej tisti i, pri katerem A_{i-1} in A_i nista dve zaporedni točki z roba ovojnice in središče kroga ne leži v odseku, ki ga določa tetiva $A_{i-1}A_i$.

Zasukajmo sliko tako, da bo tetiva $A_{i-1}A_i$ navpična, njej pripadajoči odsek bo desno od nje, preostanek kroga (v katerem je, kot smo pravkar ugotovili, tudi njegovo središče) pa levo od nje. Vemo tudi, da je med A_{i-1} in A_i na robu ovojnice vsaj še ena točka; recimo ji B. Takih točk je lahko celo več, gotovo pa nobena od njih ne leži na krožnici. Primer kaže naslednja slika:



Razpolovišče daljice $A_{i-1}A_i$ označimo z O in postavimo vanj koordinatno izhodišče. Daljica zdaj leži ravno na y-osi, torej imata njeni krajišči koordinate oblike $(0,\pm h)$ za $h=\overline{A_iA_{i-1}}/2$. Simetrala daljice pa je zdaj naša x-os; središče krožnice (recimo mu S) seveda leži na tej simetrali, torej ima koordinate oblike $(-c,0)$ za $c=\overline{OS}$. Kaj se zgodi, če premaknemo središče po tej simetrali malo v levo, recimo na S'? (Pri tem premikanju si predstavljajmo, da je krožnica pripeta na A_{i-1} in A_i; polmer ji bomo torej ustrezno povečali, tako da bo tudi po premiku središča še vedno šla skozi ti dve točki. Primer take nove krožnice je na gornji sliki narisan s črtkano črto.) Dolžino premika označimo z $\triangle=\overline{SS'}$, torej ima S' koordinate $(-c-\triangle,0)$. Polmer se zaradi tega premika malo podaljša; če je pri starem polmeru veljalo $r^2=h^2+c^2$, imamo pri novem polmeru $(r')^2=h^2+(c+\triangle)^2=r^2+2c\triangle+\triangle^2$.

Krog zaradi tega premika na levi strani daljice pridobi nekaj površine, ničesar pa ne izgubi. O tem se lahko prepričamo takole: če je krog prej pokrival hišo $(-x,y)$ za nek $x>0$, to pomeni, da je veljalo $(x-c)^2+y^2\le r^2$. Če na levi strani te neenačbe odštejemo $2\triangle x$, bo še vedno manjša od desne strani, saj sta x in $\triangle$ oba pozitivna. Če nato na obeh straneh enačbe prišejemo $2c\triangle+\triangle^2$ in jo malo poenostavimo, dobimo $(x-c-\triangle)^2+y^2\le(r')^2$, torej krog res pokriva to hišo tudi po premiku.

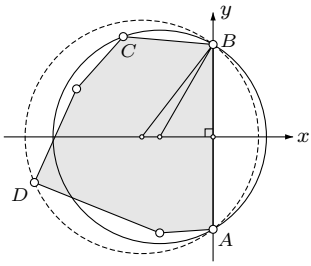
Na desni strani pa je krog nekaj površine sicer izgubil, toda če je premik $\triangle$ dovolj majhen, ni zaradi tega gotovo izpadla iz kroga nobena hiša: prav na krožnici ne leži med A_{i-1} in A_i nobena hiša; za hiše pa, ki ležijo v notranjosti kroga, lahko razmišljamo takole: recimo, da je taka hiša za d oddaljena od krožnice; iz nje lahko

izpade torej le, če se ji oddaljenost od središča poveča za več kot d (v resnici se mora povečati še malo bolj, ker se je povečal tudi polmer krožnice); če torej vzamemo za premik $\triangle$ nekaj, kar je $\leq d$, smo lahko prepričani, da naša hiša ne bo izpadla iz kroga.

Med A_{i-1} in A_i je na robu ovojnice vsaj ena hiša (malo prej smo ji rekli B), lahko jih je tudi več; vsaka od njih nam torej določa neko zgornjo mejo za $\triangle$; med temi mejami vzemimo najnižjo. Če premaknemo središče za toliko, bo vsaj ena od teh hiš (recimo brez izgube za splošnost, da bo to ravno B) med A_{i-1} in A_i prišla na krožnico, tako da bomo imeli spet krožnico skozi tri hiše (A_{i-1}, B in A_i).

Tako torej vidimo, da nas je predpostavka, da nobene tri zaporedne hiše z roba konveksne ovojnice ne ležijo na neki taki krožnici, ki pokriva vse hiše, pripeljala do zaključka, da mora obstajati neka še večja krožnica, ki pokriva vse hiše in gre skozi (vsaj) tri hiše z roba ovojnice. Torej pri iskanju največje primerne krožnice ne bomo ničesar pomembnega spregledali, če se omejimo le na krožnice skozi tri zaporedne hiše s konveksne ovojnice.

Namesto $O(n^3)$ trojic točk (in ustreznih krožnic) moramo zdaj pregledati le $O(n)$ krožnic. Spomnimo se, da je naš prvotni postopek vsakič, ko je našel večjo krožnico od dosedanje, tudi preveril, ali ta krožnica pokrije vse hiše. Izkaže pa se, da to zdaj ni več potrebno; če poiščemo največjo krožnico skozi tri zaporedne točke z roba ovojnice, bo ta krožnica zagotovo pokrila vse hiše. O tem se prepričamo takole. Tri zaporedne točke z roba ovojnice, skozi katere gre ta krožnica, označimo po vrsti z A, B, C; pa recimo, da vendarle obstaja na robu ovojnice še neka hiša D, ki leži zunaj te krožnice. Ker so A, B in C tri zaporedne hiše na robu ovojnice, to pomeni, da D ne leži niti med A in B niti med B in C, pač pa med C in A. Zasukajmo sliko tako, da je daljica AB navpična, preostanek konveksne ovojnice (med drugim tudi točki C in D) pa leži levo od nje; zdaj imamo zelo podoben prizor kot prej:



Če središče krožnice premaknemo v levo po simetrali daljice AB (in ustrezno povečamo polmer, tako da gre krožnica še vedno skozi A in B), se lahko s prav takim razmislekom kot prej prepričamo, da na levi strani krog le pridobiva ozemlje, na desni pa ga izgublja. Toda ob tem ne more na desni nobena hiša izpasti iz kroga, ker tam sploh ni nobene hiše (ker leži cela ovojnica z vsemi hišami vred levo od AB). Premaknimo torej središče krožnice tako daleč v levo, da poleg A in B leži na krožnici tudi D. Če je zunaj prvotne krožnice ležalo več hiš, vzemimo pri tem razmišljanju za D tisto med njimi, ki zahteva največji premik središča. Na koncu tega premikanja imamo torej krožnico, ki je večja od naše prvotne krožnice (skozi A, B in C), ob tem pa gre skozi tri točke z roba ovojnice (A, B in D) in pokrije

vse hiše. Za tako krožnico pa smo že prej videli, da mora obstajati tudi neka vsaj tolikšna krožnica skozi tri *zaporedne* točke z roba ovojnice, ki pokrije vse hiše. Tako smo prišli v protislovje s predpostavko, da je naša prvotna krožnica skozi A, B in C največja krožnica skozi tri zaporedne točke z roba ovojnice sploh.

Tako torej vidimo, da če poiščemo največjo krožnico skozi tri zaporedne točke z roba konveksne ovojnice, bo ta krožnica zagotovo pokrila vse hiše, tako da nam tega ni treba še posebej preverjati. Zapišimo dobljeni postopek še s psevdokodo:

```
izračunaj konveksno ovojnico množice vhodnih hiš;
za vsake tri zaporedne hiše na robu ovojnice:
  izračunaj središče in premer krožnice skozi te tri hiše;
  če je to največja krožnica doslej, si jo zapomni;
izpiši največjo dobljeno krožnico;
```

Z vsako trojico hiš imamo le $O(1)$ dela, takih trojic pa moramo zdaj pregledati le $O(m)$, če je m število hiš na robu ovojnice (seveda velja $m \leq n$). V časovni zahtevnosti tega postopka tako zdaj prevladuje iskanje konveksne ovojnice, kar je mogoče narediti na primer v $O(n \log n)$ ali $O(nm)$ časa, odvisno od tega, kateri algoritem uporabimo.

13. Kompleksnost vezij

(*a*) Vsaka od vhodnih spremenljivk ima 2 možni vrednosti, vseh vhodnih spremenljivk pa je n; obstaja 2^n različnih naborov vrednosti vseh n vhodnih spremenljivk. Na primer: pri $n = 3$ je takšnih naborov $2^3 = 8$ (to so 000, 001, 010, 011, 100, 101, 110 in 111).

Ko sestavljamo neko funkcijo oblike $\{0,1\}^n \to \{0,1\}$, si lahko pri vsakem od teh 2^n naborov vrednosti vhodnih spremenljivk izberemo vrednost funkcije na dva načina (funkcija lahko vrne 0 ali pa 1). Vseh možnih funkcij je zato 2^{2^n}. Pri $n = 3$ je na primer funkcij $2^8 = 256$, pri $n = 4$ pa že $2^{16} = 65536$.

(*b*) Za začetek speljimo vsako vhodno spremenljivko v vrata tipa NE, tako da bomo znali poleg vseh x_i računati tudi $\neg x_i$.

Za kateri koli nabor vrednosti vhodnih spremenljivk bi se zdaj dalo sestaviti takšna vrata tipa IN, ki bi vrnila vrednost 1 le, če je na vhodu v vezje točno ta nabor vrednosti spremenljivk, drugače pa bi vrnila vrednost 0. To dosežemo tako, da za vsak i od 1 do n speljemo na enega od vhodov v naša vrata IN bodisi spremenljivko x_i bodisi njeno negacijo $\neg x_i$ (torej izhod iz prej pripravljenih vrat NE), odvisno od tega, ali ima v našem izbranem naboru spremenljivka x_i vrednost 1 ali 0.

Pripravimo si po ena takšna vrata IN za vsak nabor vrednosti vhodnih spremenljivk, pri katerem ima naša funkcija f vrednost 1. Na koncu dodamo še vrata ALI, v katera speljemo izhode iz vseh vrat IN; rezultat je ravno funkcija f, ki smo jo hoteli izračunati.[15]

Vidimo, da je to vezje sestavljeno iz $n + m + 1$ vrat, če je m število naborov vrednosti vhodnih spremenljivk, pri katerih ima f vrednost 1; skupno število povezav

[15] Z drugimi besedami, funkcijo f lahko vedno izrazimo kot disjunkcijo (operator ALI) nič ali več konjunktivnih izrazov (operator IN), v katerih nastopajo le vhodne spremenljivke in njihove negacije. Takšni izražavi funcije se v logiki reče *disjunktivna normalna oblika*.

v vezju pa je $n + n \cdot m + m$. Za m velja $m \leq 2^n$, torej je kompleksnost tega vezja v najslabšem primeru $2^n + n + 1$ vrat in $2^n(n+1) + n$ povezav.

Če ima funkcija vrednost 1 v več kot polovici primerov, torej če je $m > 2^{n-1}$, se bolj splača najprej narediti (po zgoraj opisanem postopku) vezje za njeno negacijo in potem na koncu dodati še ena vrata tipa NE. S to izboljšavo je kompleksnost vezja v najslabšem primeru $2^{n-1} + n + 1$ vrat in $2^{n-1}(n+1) + n$ povezav. Pri $n = 3$ to na primer pomeni, da potrebujemo največ 8 vrat in 19 povezav.

(c) Vsa možna vezja do neke maksimalne kompleksnosti lahko preiščemo z rekurzijo: začnimo z vezjem, ki ne vsebuje nobenih vrat (le vhodne točke), nato pa na vsakem koraku na vse možne načine dodamo po ena vrata (pri vsakem tipu vrat imamo seveda tudi več možnosti glede tega, iz katerih točk speljemo povezave na vhode naših novih vrat) in nadaljujemo z rekurzivnim klicem. Iz rešitve v točki (b) smo že videli, da bi lahko katero koli funkcijo 3 spremenljivk računali z vezjem, ki ima največ 8 vrat in 19 povezav, tako da lahko rekurzijo ustavimo, če vezje doseže (ali preseže) to kompleksnost. Žal je ta rešitev prepočasna, ker je možnih vezij tudi v okviru teh omejitev veliko, poleg tega pa bo naša rekurzija do marsikaterega vezja prišla po večkrat (vsakič z malo drugačnim vrstnim redom dodajanja vrat v vezje).

V besedilu naloge smo rekli, da si v vezju izberemo neko izhodno točko in potem rečemo, da vezje računa funkcijo, ki jo predstavlja ta točka. Toda na vezje lahko pogledamo še malo drugače: namesto da bi si izbrali eno samo izhodno točko, si lahko predstavljamo, da vezje računa neko množico funkcij, namreč vseh tistih, ki jih predstavljajo točke vezja. Zdaj lahko definiramo kompleksnost množice funkcij kot kompleksnost najpreprostejšega vezja, ki računa natanko to množico funkcij. Hitro lahko vidimo, da potem kompleksnost posamezne funkcije f ni nič drugega kot kompleksnost najpreprostejše take množice funkcij, ki vsebuje f.

Če pogledamo več različnih vezij, ki računajo isto množico funkcij, vidimo, da med njimi pravzaprav za nas ni nobene bistvene razlike (razen v kompleksnosti). Recimo, da imamo neko vezje V za množico funkcij $A = \{f_1, f_2, \ldots, f_k\}$; to med drugim pomeni, da je v vezju neka točka, ki računa f_1, pa neka točka, ki računa f_2, in tako naprej. Recimo, da dodamo v to vezje nova vrata tipa IN in na njihove vhode speljemo izhoda točk, ki računata funkciji f_1 in f_2. Dobili smo vezje (recimo mu V') za množico funkcij $A' = A \cup \{g\}$, pri čemer je g definirana kot $g(x) = f_1(x) \wedge f_2(x)$. Toda če bi namesto vezja V začeli s poljubnim drugim vezjem U, ki računa isto množico funkcij A, bi lahko prav enako naredili tudi z njim: tudi v vezju U gotovo obstaja neka točka, ki računa f_1, in neka točka, ki računa f_2; in izhoda teh dveh točk bi lahko zdaj speljali na vhoda novih vrat IN, ki bi jih dodali v U. Dobili bi neko novo vezje U', ki je sicer mogoče drugačno od V', vendar računa isto množico funkcij, namreč A'. Enak razmislek velja seveda tudi za vrata tipov ALI in NE.

Ko torej razmišljamo o tem, kako je mogoče z dodajanjem novih vrat razširiti množico funkcij, ki jo neko vezje računa, lahko podrobnosti tega vezja popolnoma zanemarimo. Zato je dovolj že, če namesto vseh možnih vezij preiščemo vse možne množice funkcij (do neke maksimalne kompleksnosti). Množice funkcij bomo hranili kot ključe v razpršeni tabeli H, kot spremljevalni podatek pri vsaki množici pa še najmanjšo kompleksnost, s katero smo uspeli doslej to množico izračunati. V grobem bo torej naš postopek takšen:

vhod: število vhodnih spremenljivk n, maksimalna mestnost R,

maksimalna kompleksnost C;

1 naj bo H prazna razpršena tabela;
naj bo $A = \{g_1, \ldots, g_n\}$, pri čemer je $g_i(x_1, \ldots, x_n) = x_i$;
dodaj A v H (s pripadajočo kompleksnostjo 0 — to je namreč množica,
ki jo računa vezje brez vrat, le z vhodnimi točkami);
2 **for** $c := 0$ **to** $C - 1$:
3 za vsako $A \in H$, če je njena kompleksnost enaka c:
4 za vsako možno razširitev A' množice A:
5 naj bo c' kompleksnost množice A';
if $c' > C$ **then continue**;
6 če A' še ni v H ali pa ima tam kompleksnost, večjo od c',
potem vpiši A' v H s pripadajočo kompleksnostjo c';

Dodati moramo še nekaj podrobnosti. V koraku 4 nam „razširitev“ množice A pomeni vsako tako $A' = A \cup \{g\}$, ki jo lahko izračunamo, če v vezje za izračun množice A dodamo ena nova vrata, ki računajo funkcijo g. Če je na primer $A = \{f_1, \ldots, f_k\}$, jo lahko razširimo na naslednje načine: (1) za poljuben i od 1 do k lahko dodamo vrata tipa NE, torej bo $g(x_1, \ldots, x_n) = \neg f_i(x_1, \ldots, x_n)$; (2) za poljubno kombinacijo indeksov $i_1, \ldots, i_r$ (za $2 \leq r \leq R$) lahko dodamo vrata tipa IN, torej bo $g(x_1, \ldots, x_n) = \wedge_{j=1}^{r} f_{i_j}(x_1, \ldots, x_n)$; (3) enako kot (2), le z vrati tipa ALI (v definiciji funkcije g uporabimo $\vee$ namesto $\wedge$). Vsaka od teh treh možnosti nam torej potencialno dá veliko možnih razširitev; tiste pri točki (1) lahko preizkusimo z zanko, tiste v točkah (2) in (3) pa je najlažje zgenerirati z rekurzijo. Pri vsaki možni razširitvi tudi ni težko (v koraku 5) določiti kompleksnosti c' naše razširjene množice funkcij A'; če merimo kompleksnost s številom vrat, je $c' = c + 1$, če pa jo merimo s številom povezav, bomo v primeru razširitve tipa (1) vzeli $c' = c + 1$, pri razširitvah tipa (2) in (3) pa $c' = c + r$.

Na koncu nas bo v resnici bolj zanimala kompleksnost funkcij kot pa množic funkcij. Videli smo že, da je kompleksnost funkcije f preprosto minimum kompleksnosti vseh takih množic A, ki vsebujejo f. Lahko si torej pripravimo še eno tabelo T, v katero zapisujemo kompleksnost funkcij. Med vrstici 3 in 4 našega gornjega postopka pa dodajmo:

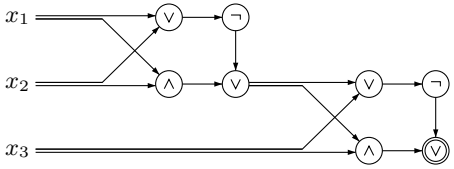
za vsako funkcijo $f \in A$:
če f še nimamo v tabeli T, jo tja vpišimo (s pripadajočo
kompleksnostjo c);

Čim opazimo, da imamo v tabeli T že vseh 2^{2^n} možnih funkcij, lahko celoten postopek prekinemo.

Za našo podnalogo (c) bomo pri gornjem postopku kot vhodne parametre vzeli $n = 3$ in maksimalno kompleksnost $C = 19$ (saj smo pri (b) videli, da več kot 19 povezav ne bomo potrebovali). V našem postopku smo kot parameter predvideli tudi R, največjo dovoljeno mestnost operatorjev IN in ALI (z drugimi besedami, to je največje dovoljeno število vhodov v vrata tega tipa). To bo prišlo prav pri podnalogah (d) in (e), kjer bomo postavili $R = 3$ oz. $R = 2$; kaj pa naj naredimo pri (c), kjer tovrstna omejitev ni predpisana? Razmišljamo lahko takole: recimo, da imamo neka vrata (tipa IN ali ALI) z R vhodi. Smiselno je, da vsi vhodi prihajajo iz različnih točk (drugače lahko odvečne povezave pobrišemo, pa na rezultat to ne bo

nič vplivalo); največ n od teh točk je lahko vhodnih, vsaj $R-n$ pa jih je torej vrat; vsaka od teh vrat prispevajo h kompleksnosti celega vezja vsaj 1, poleg tega pa naša vrata z R vhodi prispevajo h kompleksnosti vezja še R. Kompleksnost celega vezja bo torej vsaj $2R-n$. Ker nas zanimajo vezja s kompleksnostjo največ C, smo dobili mejo $2R-n \le C$, torej $R \le (C+n)/2$. V našem primeru imamo $C=19$, $n=3$, zato dobimo $R=11$.

Zdaj torej znamo izračunati kompleksnost vseh 256 možnih funkcij treh spremenljivk. Izkaže se, da je največja kompleksnost 14 povezav, pa še to potrebujemo le pri dveh funkcijah: ena je $\textsc{xor}(x_1,x_2,x_3) = (x_1+x_2+x_3) \bmod 2$, druga pa je njena negacija. (To funkcijo si lahko predstavljamo kot posplošitev operacije XOR na tri spremenljivke; če je prižganih liho mnogo vhodnih bitov, ima funkcija vrednost 1, sicer pa vrednost 0.) Naslednja slika kaže primer vezja z minimalno kompleksnostjo za funkcijo XOR (krog z dvojnim robom predstavlja izhodna vrata):



Naš gornji postopek nam sicer ne poišče vezja, pač pa neko množico funkcij, v kateri je med drugim tudi XOR (in še sedem drugih funkcij, ki jih računa ostalih sedem vrat gornjega vezja). V tej množici lahko zdaj gledamo, kako bi se dalo nekatere funkcije računati iz drugih, in tako z malo razmisleka rekonstruiramo vezje na gornji sliki. Lahko bi si celo napisali podprogram, ki bi na ta način iz množice rekonstruiral vezje. Možno pa bi bilo tudi dopolniti naš postopek za preiskovanje prostora množic funkcij tako, da bi v koraku 6, ko vpisuje A' v tabelo H, zraven tudi zabeležil, iz katere manjše množice A je prišel do A' in s kakšno razširitvijo (kakšna vrata je dodal in kaj je vzel za njihove vhode); vendar pa bi program zaradi tega porabil občutno več pomnilnika kot prej.

Kaj pa, če kompleksnost namesto s številom povezav definiramo s številom vrat (kot zahteva podnaloga (f))? Že v (b) smo videli, da bomo potrebovali največ 8 vrat. Izkaže se, da toliko vrat potrebujemo le pri funkciji XOR in njeni negaciji, za vsako drugo funkcijo pa je dovolj že 7 ali manj vrat.

(d) Če poženemo naš gornji postopek z $R=3$, se izkaže, da dobimo pri vseh funkcijah popolnoma enako kompleksnost kot pri (c). Z drugimi besedami, za vsako funkcijo treh spremenljivk je mogoče vezje z minimalno kompleksnostjo sestaviti tudi tako, da nikoli ne uporabimo vrat z več kot tremi vhodi.

(e) Če se omejimo na vrata z največ dvema vhodoma, se nekaterim funkcijam kompleksnost poveča (pri tem pa maksimum kompleksnosti po vseh 256 funkcijah ostane nespremenjen; to velja za obe definiciji kompleksnosti, tako tisto s številom povezav kot tisto s številom vrat). Natančneje povedano, pri 48 funkcijah se obe vrsti kompleksnosti povečata za 1; pri 6 funkcijah se poveča za 1 le kompleksnost, merjena s številom vrat, medtem ko kompleksnost, merjena s številom povezav, ostane nespremenjena; pri dveh funkcijah se obe vrsti kompleksnosti povečata za 2; pri ostalih funkcijah pa obe vrsti kompleksnosti ostaneta nespremenjeni. Funkciji,

pri katerih se obe vrsti kompleksnosti povečata za 2, sta

$$f(x_1, x_2, x_3) = \begin{cases} 1, & \text{če je } x_1 = x_2 = x_3 \\ 0, & \text{sicer} \end{cases}$$

in njena negacija. Maksimalno kompleksnost (8 vrat in 14 povezav) doseže zdaj kar osem funkcij, ne le dve kot prej (XOR in njena negacija).

14. Osebni rekord

Območju med dvema zaporednima meritvama v vhodnih podatkih recimo *segment*; dolžino i-tega segmenta označimo z $d_i := D_i - D_{i-1}$, njegovo trajanje pa s $t_i := T_i - T_{i-1}$.

(a) Pri tej različici naloge imamo en sam segment dolžine D, ki je večkratnik d; recimo, da je $D = kd$ (za neko celo število k). En možen potek teka, ki je vsekakor skladen z vhodnimi podatki, je tisti, pri katerem tekač ves čas teče s konstantno hitrostjo (torej D/T metrov na sekundo). V tem primeru za katerikoli d-metrski podinterval porabi T/k časa, torej tudi za najhitrejši d-metrski podinterval porabi T/k časa.

Ali obstaja kak drug potek (skladen z vhodnimi podatki) pri katerem bi za najhitrejši d-metrski podinterval porabili več kot T/k časa? Recimo, da bi tak potek obstajal; v njem torej za vsak d-metrski podinterval porabimo več kot T/k časa. Celoten tek lahko razdelimo na k intervalov dolžine d; pravkar smo videli, da za vsakega od njih porabimo več kot T/k časa; za celoten tek torej porabimo več kot $k \cdot (T/k) = T$ časa, to pa je nemogoče, saj tak potek ne bi bil skladen z vhodnimi podatki (ti namreč pravijo, da smo za celoten tek porabili natanko T časa).

Tako torej vidimo, da obstaja potek (skladen z vhodnimi podatki), pri katerem za najhitrejši d-metrski podinterval porabimo T/k časa, ne obstaja pa potek, pri katerem bi za najhitrejši d-metrski podinterval porabili več kot T/k časa. Odgovor, po katerem sprašuje naloga, je torej T/k.

(b) Razmišljamo lahko podobno kot pri (a). Predstavljamo si lahko potek P, v katerem vsak segment pretečemo s konstantno hitrostjo (za segment i je to hitrost d_i/t_i metrov na sekundo). Če zdaj v P pogledamo tak d-metrski interval, ki v celoti leži znotraj i-tega segmenta, vidimo, da smo za ta interval porabili $d/(d_i/t_i)$ časa; najhitrejši interval takšne oblike ima torej čas $\min_i d/(d_i/t_i)$. Kaj pa, če naš d-metrski interval leži na meji med dvema segmentoma? Del tega intervala torej pretečemo z eno hitrostjo, del pa z drugo; tak interval gotovo ni hitrejši, kot če bi v celoti ležal znotraj tistega od teh dveh segmentov, ki ima višjo hitrost. Torej lahko intervale, ki ležijo malo v enem segmentu in malo v drugem, pri določanju $f(P)$ ignoriramo. Tako torej vidimo, da je $f(P) = \min_i d/(d_i/t_i)$; zato pa je tudi $f^\star \geq \min_i d/(d_i/t_i)$.

Ali je mogoče, da bi bila ta neenakost stroga, torej da bi bilo $f^\star > \min_i d/(d_i/t_i)$? V tem primeru bi moral obstajati nek potek $\tilde{P}$, skladen z vhodnimi podatki, ki bi imel $f(\tilde{P}) > \min_i d/(d_i/t_i)$. Oglejmo si v tem poteku i-ti segment; po predpostavki je njegova dolžina, d_i, večkratnik d-ja; recimo, da je $d_i = k_i \cdot d$. Ta segment lahko razdelimo na k_i kosov dolžine d in če bi za vsakega od njih porabili več kot t_i/k_i časa, bi za celoten segment porabili več kot t_i časa, kar pa je nemogoče (saj vhodni

podatki pravijo, da za ta segment porabimo natanko t_i časa). Torej v i-tem segmentu gotovo obstaja nek d-metrski interval, ki smo ga pretekli v $\leq t_i/k_i$ časa; to pa je naprej enako $t_i/(d_i/d) = d/(d_i/t_i)$. Ker ta razmislek velja pri vsakem i, lahko zaključimo, da v $\tilde{P}$ prav gotovo obstaja nek d-metrski interval, ki smo ga pretekli v največ $\min_i d/(d_i/t_i)$ časa. Torej je $f(\tilde{P}) \leq \min_i d/(d_i/t_i)$. Tako smo prišli v protislovje, torej tak $\tilde{P}$ sploh ne more obstajati, torej $f^\star$ ne more biti strogo večji od $\min_i d/(d_i/t_i)$. Torej je $f^\star = \min_i d/(d_i/t_i)$.

(c) Najprej smo morebiti v skušnjavi, da bi začeli podobno kot pri (a) in sestavili potek, ki bi pretekel cel segment s konstantno hitrostjo. V našem primeru bi bila ta hitrost $D/T = 250/50 = 5$ metrov na skundo. Ker je $d = 100$, bi za kateri koli d-metrski podinterval porabili 20 sekund, torej je $f(P) = 20$. Ali je to že tudi $f^\star$?

Razmislimo, ali je mogoče sestaviti nek drugačen potek P, ki bi imel $f(P)$ večji od 20, recimo $f(P) = q$ za nek $q > 20$. Za kateri koli d-metrski podinterval moramo torej porabiti vsaj q sekund. Med drugim to pomeni, da za interval $[0, 100]$ porabimo vsaj q sekund; enako tudi za interval $[100, 200]$; in ker mora biti celoten tek dolg T sekund (da bo P skladen z vhodnimi podatki), nam za $[200, 250]$ ostane kvečjemu $a := T - 2q$ sekund. Ta čas seveda ne sme biti negativen ali enak 0, tako da smo dobili omejitev: $T - 2q > 0$, kar v našem primeru (ker je $T = 50$) pomeni $q < 25$.

Razmišljajmo zdaj vzvratno: tudi $[150, 250]$ je d-metrski interval, torej moramo tudi zanj porabiti vsaj q sekund; po drugi strani smo ravnokar videli, da za $[200, 250]$ porabimo največ a sekund; torej moramo za $[150, 200]$ porabiti vsaj $b := q - a$ sekund. Videli smo že, da hočemo za $[100, 200]$ porabiti vsaj q sekund; to se načeloma lahko lepo izide, če jih na $[100, 150]$ porabimo vsaj a, saj bomo tako skupaj z b sekundami na $[150, 200]$ dobili vsaj q sekund za interval $[100, 200]$. S tem razmišljanjem lahko nadaljujemo in si sčasoma sestavimo takšen scenarij:

	0	50	100	150	200	250 m
dolžina [m]	50	50	50	50	50	
čas [s]	a	b	a	b	a	

Konkreten potek P lahko torej sestavimo takole: prvih 50 m pretečemo s konstantno hitrostjo $50/a$, naslednjih 50 m s konstantno hitrostjo $50/b$, naslednjih 50 m spet s hitrostjo $50/a$ in tako naprej. Vidimo lahko, da kakorkoli si v tem poteku izberemo interval dolžine $d = 100$ (tudi če se ne začne pri večkratniku 50), bo 50 m tega intervala padlo na območja s hitrostjo $50/a$, ostalih 50 m pa na območja s hitrostjo $50/b$, tako da bo skupni čas tega intervala enak $50/(50/a) + 50/(50/b) = a + b = q$, torej je $f(P) = q$.

Prej smo videli omejitev $q < 25$. Torej lahko sestavimo poteke P, ki imajo $f(P)$ poljubno blizu 25, tako da je $f^\star \geq 25$. Naša prvotna domneva o $f^\star = 20$ je torej napačna.

Ali bi lahko pri kakšnem poteku P (skladnem z vhodnimi podatki) veljalo celo $f(P) > 25$? To bi pomenilo, da za vsak d-metrski podinterval porabimo več kot 25 sekund. Razdelimo v mislih naš potek na dva podintervala, $[0, 125]$ in $[125, 250]$. Ker je vsak od njiju daljši od d in ker za vsak d-metrski interval porabimo več kot 25 sekund, iz tega sledi, da za celoten tek porabimo več kot 50 sekund; to pa je

protislovje, saj vhodni podatki pravijo, da je celoten tek trajal le 50 sekund (in mi smo predpostavili, da je P skladen z njimi).

Torej lahko $f(P)$ pride poljubno blizu 25, ne more pa te meje preseči; zato je $f^\star = 25$.

(d) Zdaj moramo le posplošiti razmislek, ki smo ga opravili pri (c). Ker D ni nujno večkratnik d-ja, ga lahko zapišemo v obliki $D = k \cdot d + r$ za nek celoštevilski količnik k in za ostanek $r \in [0, d)$. Pišimo še $s = d - k$ in sestavimo potek P takole:

	0		d		$2d$		$(k-1)d$		kd	D
dolžina [m]	r	s	r	s	$\cdots$	r	s	r		
čas [s]	a	b	a	b	$\cdots$	a	b	a		

V našem poteku si torej izmenično sledijo kosi dolžine r (ki jih je $k+1$) in kosi dolžine s (ki jih je k). V vsakem kosu dolžine r imamo konstantno hitrost r/a in ga torej pretečemo v a sekundah, v vsakem kosu dolžine s pa imamo konstantno hitrost s/b in ga torej pretečemo v b sekundah. Ker je $d = s + r$, se lahko hitro prepričamo, da kakorkoli si tu izberemo interval dolžine d, bo gotovo r metrov tega intervala padlo v kose s hitrostjo r/a, preostalih s metrov tega intervala pa bo padlo v kose s hitrostjo s/b, tako da bomo za tak interval prav gotovo porabili $a + b$ časa. V našem poteku je torej $f(P) = a + b$.

Seveda nas zanimajo le poteki, ki so skladni z vhodnimi podatki. V našem primeru to pomeni, da mora imeti potek skupen čas T in dolžino D. Dolžina je že prava, za čas pa imamo $T = (k+1)a + kb$, kar nam dá zvezo $a = (T - kb)/(k+1)$. Če to vstavimo v $f(P)$, dobimo $f(P) = a + b = (T + b)/(k+1)$. Poleg tega naš tekač ne more teči neskončno hitro, torej imamo omejitev $a > 0$, torej $(T - kb)/(k+1) > 0$, torej $b < T/k$. Če to vstavimo v $f(P)$, dobimo $f(P) = (T + b)/(k+1) < T/(k+1) + T/(k(k+1)) = T/k$. Tako torej vidimo, da lahko sestavimo potek P, ki ima $f(P)$ poljubno blizu T/k; vzeti moramo le tak b, ki je dovolj blizu T/k (a še vedno manjši od njega, tako da bo potem tudi a, dobljen po formuli $a = (T - kb)/(k+1)$, pozitiven). Iz tega sledi, da je $f^\star \geq T/k$.

Ali bi lahko pri kakšnem poteku P (skladnem z vhodnimi podatki) veljalo celo $f(P) > T/k$? Ker je $D \geq kd$, lahko naš tek razdelimo na k enako dolgih podintervalov dolžine D/k in vemo, da je vsak od teh dolg vsaj d. Če je $f(P) > T/k$, mora vsak od teh podintevalov trajati več kot T/k časa, torej mora celoten tek trajati več kot $k(T/k) = T$ časa, to pa je protislovje (ker tak P ne bi bil skladen). Torej za vsak skladen P velja $f(P) \leq T/k$, tako da je tudi $f^\star \leq T/k$.

Vidimo torej, da lahko $f(P)$ pride poljubno blizu T/k, ne more pa te meje preseči. Zato je $f^\star = T/k$ oz. $f^\star = T/\lfloor D/d \rfloor$. Mimogrede, opazimo lahko tudi, da je ta rezultat pravzaprav posplošitev tistega iz podnaloge (a), le da moramo količnik D/d pred nadaljnjo uporabo zaokrožiti navzdol.

Za preizkus uporabimo našo formulo na primeru iz podnaloge (c): če vstavimo $T = 50$, $D = 250$, $d = 100$, res dobimo $f^\star = T/\lfloor D/d \rfloor = 50/\lfloor 250/100 \rfloor = 50/2 = 25$, kar je enak rezultat kot prej pri (c).

(e) Pri tej podnalogi imamo dva segmenta, vsak od njiju je dolg 19 m in je bil pretečen v 6 sekundah; zanima pa nas $d = 10$, torej najhitrejši 10-metrski podinterval.

Najprej nam mogoče pride na misel, da bi rešili problem za vsak segment posebej in potem vzeli minimum po vseh segmentih, tako kot smo to naredili pri (b). Če uporabimo rezultat iz (d) za vsak segment posebej, vidimo, da na takem segmentu lahko najhitreje pretečeni 10-metrski podinterval traja poljubno blizu $6/\lfloor 19/10 \rfloor = 6$ sekund, ne more pa najhitreje pretečeni 10-metrski podinterval trajati več kot 6 sekund. Če vzamemo minimum tega po obeh segmentih, imamo še vedno 6 sekund, saj sta oba segmenta enaka.

Toda ali je to pravi odgovor? Spomnimo se, s kakšnim potekom smo se pri (d) približali tej meji 6 sekund. Če tak potek uporabimo v vsakem od obeh segmentov, je celoten potek takšne oblike:

	0	9	10	19	28	29	38 m
dolžina [m]	9	1	9	9	1	9	
čas [s]	a	b	a	a	b	a	

Pri tem moramo vzeti nek a blizu 0 in b blizu 6 (seveda ob upoštevanju omejitve $2a + b = 6$). Toda vidimo lahko, da nam pri takšnem poteku okoli meje med segmentoma nastane 18-metrski interval, ki je bil pretečen v samo $2a$ sekundah. Če torej naš 10-metrski interval vzamemo na tem območju, ga bomo gotovo pretekli v največ $2a$ sekundah, zato ima naš potek $f(P) \leq 2a$, kar je precej manj kot 6.

Vzemimo poljuben potek P, ki je skladen z vhodnimi podatki in ima $f(P) = q$. Kaj lahko iz tega sklepamo o P (in o q)?

Vsak d-metrski (za $d = 10$) podinterval mora torej trajati vsaj q sekund; med drugim na primer $P[0, 10] \geq q$. Po drugi strani, ker je P skladen z vhodnimi podatki, velja $P[0, 19] = 6$. To dvoje skupaj nam dá $P[10, 19] \leq 6 - q$. Podobno zaradi $P[9, 19] \geq q$ dobimo $P[0, 9] \leq 6 - q$. Slednje nam skupaj s $P[0, 10] \geq q$ dá $P[9, 10] \geq q - (6 - q) = 2q - 6$.

V drugem segmentu nas podoben razmislek pripelje do $P[19, 28] \leq 6 - q$ in $P[29, 38] \geq 6 - q$, nato pa do $P[28, 29] \geq 2q - 6$.

Iz $P[10, 19] \leq 6 - q$ skupaj s $P[10, 20] \geq q$ dobimo $P[19, 20] \geq 2q - 6$. To nam skupaj s $P[19, 28] \leq 6 - q$ dá $P[20, 28] = P[19, 28] - P[19, 20] \leq (6 - q) - (2q - 6) = 12 - 3q$. To je koristen podatek; ker naš tekač ne more teči neskončno hitro, mora biti čas takega intervala pozitiven, torej imamo $12 - 3q > 0$ in zato $q < 4$. Torej za vsak skladen potek P velja $f(P) < 4$ in zato $f^\star \leq 4$.

Podobno iz $P[19, 28] \leq 6 - q$ skupaj s $P[18, 28] \geq q$ dobimo $P[18, 19] \geq 2q - 6$; to pa nam skupaj s $P[10, 19] \leq 6 - q$ dá $P[10, 18] \leq 12 - 3q$. Iz $P[0, 19] = 6$ lahko zdaj sklepamo na $P[0, 8] = P[0, 19] - P[8, 18] - P[18, 19] \leq 6 - q - (2q - 6) = 12 - 3q$. Nato dobimo še $P[8, 10] = P[0, 19] - P[0, 10] - P[10, 19] \geq 6 - (12 - 3q) - (6 - q) = 4q - 12$.

Podobno na desni dobimo $P[30, 38] \geq 12 - 3q$ in nato $P[28, 30] \geq 4q - 12$.

Če vse dosedanje ugotovitve združimo, vidimo, da ima vsak P, ki je skladen z vhodnimi podatki in ima $f(P) = q$, takšno obliko:

0	8	9	10	18	19	20	28	29	30	38 m
$\leq 12-3q$	$\geq 4q-12$	$\geq 2q-6$	$\leq 12-3q$	$\geq 2q-6$	$\geq 2q-6$	$\leq 12-3q$	$\geq 4q-12$	$\geq 2q-6$	$\leq 12-3q$	

Zdaj ni težko sestaviti konkretnega poteka z želenim q. Izberimo si poljuben $q < 4$ in izračunajmo $a = 12 - 3q$ in $b = 2q - 6$. Sestavimo takšen potek:

	0	8	9	10	18	19	20	28	29	30	38 m
dolžina [m]	8	1	1	8	1	1	8	1	1	8	
čas [s]	a	b	b	a	b	b	a	b	b	a	

Znotraj vsakega kosa dolžine 8 imejmo konstantno hitrost $8/a$, znotraj vsakega kosa dolžine 1 pa konstantno hitrost $1/b$. Če je q blizu 4, bo a blizu 0 in b blizu 2, zato si kose dolžine 8 lahko predstavljamo kot hitre, kose dolžine 1 pa kot počasne. Če pogledamo katerikoli d-metrski interval (za $d = 10$), vidimo, da 8 metrov tega intervala pade v hitra območja (in jih pretečemo v a sekundah), ostala 2 metra pa v počasna območja (in ju pretečemo v $2b$ sekundah), tako da je čas takega intervala v vsakem primeru $a + 2b$, kar je naprej enako $(12 - 3q) + 2(2q - 6) = q$. Torej je čas vsakega d-metrskega intervala q, zato je $f(P) = q$, točno to pa smo tudi želeli. Ta konstrukcija nam je tudi pokazala, da lahko s $f(P)$ pridemo poljubno blizu 4. Ker smo prej tudi videli, da $f(P)$ ne more preseči 4, lahko zaključimo, da je $f^\star = 4$.

(f) Poskusimo posplošiti razmislek, s katerim smo reševali podnalogo (e). Opazimo lahko, da se je naš razmislek tam začenjal s podintervali, ki so imeli eno od krajišč na začetku ali koncu kakšnega segmenta (torej na enem od D_i), drugo krajišče pa za d stran. Tudi v nadaljevanju smo se premikali za d levo ali desno, računali presek dveh takšnih podintervalov in podobno. Ves čas smo torej imeli opravka s podintervali, katerih krajišča so bila oblike $D_i \pm k \cdot d$ za celoštevilske k. Posumimo lahko, da bodo takšni podintervali zadoščali tudi v splošnem, ne le pri konkretnem primeru iz podnaloge (e).

Poiščimo torej vse tiste koordinate oblike $D_i \pm kd$, ki ležijo na $[0, D]$ in imajo celoštevilski k. Vse te koordinate uredimo naraščajoče in jih oštevilčimo: $x_0, x_1, \ldots, x_m$.

Predstavljajmo si poljuben potek P, ki je skladen z vhodnimi podatki. Da bo manj pisanja, pišimo $u_i = P[x_{i-1}, x_i]$ in $q = f(P)$. Ker je P skladen z vhodnimi podatki, za ta števila seveda velja naslednje:

- Za vsak segment j vemo, da je $P[D_{j-1}, D_j] = t_j$. Levo stran te enačbe lahko seveda zapišemo tudi kot vsoto nekaj zaporednih spremenljivk u_i; na desni strani pa imamo konstanto $t_j = T_j - T_{j-1}$.

- Za vsak $x \in [0, D - d]$ vemo, da je $P[x, x + d] \geq f(P)$. To velja torej tudi za $x = x_i$. V tem primeru lahko levo stran te neenačbe zapišemo tudi kot vsoto nekaj zaporednih spremenljivk u_i; na desni strani pa imamo q.

- Čas vožnje za posamezni interval oblike $[x_{i-1}, x_i]$ seveda ne more biti negativen, torej velja $u_i \geq 0$.

Tako smo dobili nek sistem linearnih enačb in neenačb v odvisnosti od neznank $u_1, \ldots, u_m, q$. Za vsak skladen potek P lahko torej sestavimo neko dopustno rešitev tega sistema in to tako, da ima v tej dopustni rešitvi spremenljivka q vrednost $f(P)$.

Sistem torej gotovo ima dopustne rešitve; poiščimo med njimi tisto z največjim q. (To bi se dalo narediti na primer s kakšnim od algoritmov za linearno programiranje.) Označimo jo z $(\mathbf{u}^\star, q^\star)$. Ker nam vsak skladen potek P dá neko dopustno rešitev

s $q = f(P)$ in ker za vsako dopustno rešitev $(\mathbf{u}, q)$ velja $q \le q^\star$, sledi, da za vsak skladen potek P velja $f(P) \le q^\star$, torej $f^\star = \sup\{f(P) : P \text{ je skladen}\} \le q^\star$.

Ali je mogoče, da bi tu veljala stroga neenakost, torej da bi bil $q^\star$ strogo večji od supremuma $f^\star$?

Oglejmo si pobliže rešitev $(\mathbf{u}^\star, q^\star)$. Spomnimo se, da imamo v našem sistemu (ne)enačb po eno enačbo za vsak segment iz vhodnih podatkov: ta enačba je oblike $u_j + u_{j+1} + \ldots + u_k = t$, pri čemer je t čas teka tistega segmenta. Ker ne bi bilo smiselno, da bi bil nek segment pretečen v času 0, so vhodni podatki gotovo taki, da je $t > 0$; torej mora biti vsaj eden od $u_j, u_{j+1}, \ldots, u_k$ večji od 0 (saj bi drugače leva stran ne mogla biti večja od 0, torej ne bi mogla biti enaka desni strani, ki je $t > 0$). Poglejmo zdaj v naši rešitvi $(\mathbf{u}^\star, q^\star)$, koliko izmed spremenljivk $u_j^\star, u_{j+1}^\star, \ldots, u_k$ je enakih 0 (gotovo ni nobena manjša od 0, saj potem rešitev ne bi bila dopustna); recimo, da jih je r (gotovo je $r < m$); povečajmo jih na ε (pri tem naj bo ε neko majhno pozitivno število; več o tem, kako si ga izbrati, bomo videli kasneje). Malo prej smo tudi videli, da je ena od spremenljivk $u_j^\star, u_{j+1}^\star, \ldots, u_k^\star$ gotovo že prej bila večja od 0; njo zdaj zmanjšajmo za $r\varepsilon$; če smo si izbrali dovolj majhen ε, bo ta spremenljivka še vedno večja od 0. Tako se na levi strani enačbe $u_j + u_{j+1} + \ldots + u_k = t$ nič ne spremeni (ena spremenljivka se za prav toliko zmanjša, za kolikor se vse ostale skupaj povečajo), zato je enačba še vedno izpolnjena.

Ta popravek izvedimo za vsak segment (torej pri vsaki enačbi našega sistema); popravljeni nabor u-jev označimo zdaj z $\tilde{\mathbf{u}}$ namesto $\mathbf{u}^\star$. Kaj pa se zgodi pri neenačbah? Nekatere spremenljivke na levi strani neenačb se povečajo za ε; nekatere se zmanjšajo, a za manj kot $m\varepsilon$; nekatere pa mogoče ostanejo nespremenjene. Ker je na levi strani neenačbe največ m spremenljivk, se torej leva stran zmanjša za največ $m^2\varepsilon$. Na desni strani neenačbe pa je $q^\star$; ker se je leva stran zmanjšala, je mogoče, da neenačba zdaj ne velja več; bo pa gotovo spet začela veljati, če za toliko zmanjšamo tudi $q^\star$. Če torej postavimo $\tilde{q} = q^\star - m^2\varepsilon$, vidimo, da je $(\tilde{\mathbf{u}}, \tilde{q})$ tudi dopustna rešitev, le malenkost slabša (za $m^2\varepsilon$) od optimalne $(\mathbf{u}^\star, q^\star)$. Opozorimo na to, da je $q^\star$ gotovo večja od 0, zato si vsekakor lahko izberemo dovolj majhen ε, da bo tudi $\tilde{q}$ večja od 0.

Lepo pri naši popravljeni dopustni rešitvi je, da so v njej vsi $\tilde{u}_i$ strogo večji od 0. Zato si lahko zdaj predstavljamo potek dirke $\tilde{P}$, v katerem interval $[x_{i-1}, x_i]$ pretečemo s konstantno hitrostjo $(x_i - x_{i-1})/\tilde{u}_i$ metrov na sekundo (ker so $\tilde{u}_i > 0$, bo to mogoče že pri neki končni hitrosti); za ta interval torej porabimo $\tilde{u}_i$ sekund. Ker je $\tilde{P}$ nastal iz dopustne rešitve $(\tilde{\mathbf{u}}, \tilde{q})$, je v njem gotovo vsak segment pretečen v predpisanem času; torej je $\tilde{P}$ skladen z vhodnimi podatki.

Kaj pa lahko povemo o $f(\tilde{P})$? Naj bo $[x, x + d]$ najhitreje pretečeni d-metrski podinterval v poteku $\tilde{P}$. Če je x enak enemu od x_i, potem je (zaradi načina, kako smo sestavili seznam koordinat $x_0, \ldots, x_m$) gotovo tudi $x + d$ enak nekemu x_j (za nek $j > i$) in v našem sistemu neenačb je bila tudi neenačba $u_i + \ldots + u_j \ge q$. Ta neenačba je veljala tudi v naši dopustni rešitvi $(\tilde{\mathbf{u}}, \tilde{q})$, torej je $\tilde{u}_i + \ldots + \tilde{u}_j \ge \tilde{q}$. Iz načina, kako smo iz $(\tilde{\mathbf{u}}, \tilde{q})$ dobili $\tilde{P}$, pa sledi, da je leva stran te neenačbe tudi enaka $\tilde{P}[x_i, x_j] = \tilde{P}[x, x + d] = f(\tilde{P})$. Tako torej vidimo, da je $f(\tilde{P}) \ge \tilde{q}$.

Kaj pa, če x ni enak nobenemu od x_i? Potem mora ležati med dvema zaporednima koordinatama, recimo med x_{i-1} in x_i. Zaradi načina, kako smo pripravili seznam koordinat $x_0, \ldots, x_m$, vemo, da zagotovo obstaja tudi koordinata z vred-

nostjo $x_{i-1}+d$ in ena z vrednostjo x_i+d; in med njima v našem seznamu gotovo ni nobene druge (kajti če bi bila tam vmes v našem seznamu neka koordinata x', potem bi morala med x_{i-1} in x_i ležati v našem seznamu tudi koordinata $x'-d$, to pa bi bilo v protislovju s predpostavko, da sta x_{i-1} in x_i dva zaporedna elementa našega seznama koordinat). Tako torej vidimo, da za nek j leži tudi $x+d$ nekje med $x_{j-1}=x_{i-1}+d$ in $x_j=x_i+d$. Na intervalu $[x_{i-1},x_i]$ ima naš potek $\tilde{P}$ konstantno hitrost, recimo v; tudi na intervalu $[x_{j-1},x_j]$ ima naš potek konstantno hitrost, recimo v'. Če je $v>v'$, bi lahko interval $[x,x+d]$ premaknili malo v desno, da bi iz njega nastal $[x_i,x_j]$; pri tem bi na levi izpadel iz njega interval $[x,x_i]$, na desni pa bi vanj prišel enako dolg interval $[x+d,x_j]$; ker je bil prvi prevožen z višjo hitrostjo (v) kot drugi (v'), iz tega sledi, da je čas intervala $[x_i,x_j]$ manjši kot čas intervala $[x,x+d]$. To pa je protislovje, saj smo predpostavili, da je interval $[x,x+d]$ tisti, ki smo ga prevozili v najkrajšem času. Torej ni mogoče, da bi bila $v>v'$; na enak način pa bi se prepričali tudi, da ne more biti $v<v'$ (v tem primeru bi lahko interval $[x,x+d]$ izboljšali, če bi ga premaknili na $[x_{i-1},x_{j-1}]$). Ostane le še možnost, da je $v=v'$; tedaj lahko interval $[x,x+d]$ premaknemo bodisi v $[x_i,x_j]$ ali pa v $[x_{i-1},x_{j-1}]$, pa njegovo trajanje ostane nespremenjeno. V tem primeru torej dobimo, da je $f(\tilde{P})=\tilde{P}[x,x+d]=\tilde{P}[x_i,x_j]$, za slednjega pa smo že prej videli, da je $\geq\tilde{q}$.

V vsakem primeru torej za naš potek $\tilde{P}$ velja, da je $f(\tilde{P})\geq\tilde{q}\geq q^\star-m^2\varepsilon$. Torej mora biti tudi $f^\star=\sup\{f(P):P\text{ je skladen}\}\geq q^\star-m^2\varepsilon$. Prej smo se vprašali, ali je mogoče, da bi bil $q^\star>f^\star$. V tem primeru bi za nek $\varepsilon'>0$ veljalo $q^\star=f^\star+\varepsilon'$. Če to združimo s pravkar dobljenim $f^\star\geq q^\star-m^2\varepsilon$, dobimo naprej $q^\star=f^\star+\varepsilon'\geq q^\star+\varepsilon'-m^2\varepsilon$. Če vzamemo dovolj majhen (a še vedno pozitiven) ε, bo $\varepsilon'-m^2\varepsilon>0$ in takrat bomo dobili $q^\star>q^\star$, kar je protislovje. Torej res ni mogoče, da bi bil $q^\star>f^\star$. Ker pa smo pred tem že dokazali, da je $q^\star\geq f^\star$, nam ostane le še možnost, da je $q^\star=f^\star$. Optimalna rešitev našega optimizacijskega problema je torej ravno odgovor, po katerem je spraševala naloga — najmanjša zgornja meja za čas najhitreje prevoženega d-metrskega podintervala dirke. □

Naloge so sestavili: hišna številka, speči agenti, načrtovanje tipkovnice — Nino Bašić; 3-d tiskalnik — Boris Gašperin; Ganttov diagram, digitalna ura — Matija Grabnar; kontrolne naloge, minsko polje, — Tomaž Hočevar; funkciji — Matjaž Leonardis; Cezar — Mark Martinec; lov na zaklad, oddajnik — Jure Slak; osebni rekord — Marjan Šterk; kompleksnost vezij — Janez Brank.

NASVETI ZA MENTORJE O IZVEDBI ŠOLSKEGA TEKMOVANJA IN OCENJEVANJU NA NJEM

[Naslednje nasvete in navodila smo poslali mentorjem, ki so na posameznih šolah skrbeli za izvedbo in ocenjevanje šolskega tekmovanja. Njihov glavni namen je bil zagotoviti, da bi tekmovanje potekalo na vseh šolah na približno enak način in da bi ocenjevanje tudi na šolskem tekmovanju potekalo v približno enakem duhu kot na državnem.—*Op. ur.*]

Tekmovalci naj pišejo svoje odgovore na papir ali pa jih natipkajo z računalnikom; ocenjevanje teh odgovorov poteka v vsakem primeru tako, da jih pregleda in oceni mentor (in ne npr. tako, da bi se poskušalo izvorno kodo, ki so jo tekmovalci napisali v svojih odgovorih, prevesti na računalniku in pognati na kakšnih testnih podatkih). Pri reševanju si lahko tekmovalci pomagajo tudi z literaturo in/ali zapiski, ni pa mišljeno, da bi imeli med reševanjem dostop do interneta ali do kakšnih datotek, ki bi si jih pred tekmovanjem pripravili sami. Čas reševanja je omejen na 180 minut.

Nekatere naloge kot odgovor zahtevajo program ali podprogram v kakšnem konkretnem programskem jeziku, nekatere naloge pa so tipa „opiši postopek“. Pri slednjih je načeloma vseeno, v kakšni obliki je postopek opisan (naravni jezik, psevdokoda, diagram poteka, izvorna koda v kakšnem programskem jeziku, ipd.), samo da je ta opis dovolj jasen in podroben in je iz njega razvidno, da tekmovalec razume rešitev problema.

Glede tega, katere programske jezike tekmovalci uporabljajo, naše tekmovanje ne postavlja posebnih omejitev, niti pri nalogah, pri katerih je rešitev v nekaterih jezikih znatno krajša in enostavnejša kot v drugih (npr. uporaba perla ali pythona pri problemih na temo obdelave nizov).

Kjer se v tekmovalčevem odgovoru pojavlja izvorna koda, naj bo pri ocenjevanju poudarek predvsem na vsebinski pravilnosti, ne pa na sintaktični. Pri ocenjevanju na državnem tekmovanju zaradi manjkajočih podpičij in podobnih sintaktičnih napak odbijemo mogoče kvečjemu eno točko od dvajsetih; glavno vprašanje pri izvorni kodi je, ali se v njej skriva pravilen postopek za rešitev problema. Ravno tako ni nič hudega, če npr. tekmovalec v rešitvi v C-ju pozabi na začetku **#include**ati kakšnega od standardnih headerjev, ki bi jih sicer njegov program potreboval; ali pa če podprogram main() napiše tako, da vrača **void** namesto **int**.

Pri vsaki nalogi je možno doseči od 0 do 20 točk. Od rešitve pričakujemo predvsem to, da je pravilna (= da predlagani postopek ali podprogram vrača pravilne rezultate), poleg tega pa je zaželeno tudi, da je učinkovita (manj učinkovite rešitve dobijo manj točk).

Če tekmovalec pri neki nalogi ni uspel sestaviti cele rešitve, pač pa je prehodil vsaj del poti do nje in so v njegovem odgovoru razvidne vsaj nekatere od idej, ki jih rešitev tiste naloge potrebuje, naj vendarle dobi delež točk, ki je približno v skladu s tem, kolikšen delež rešitve je našel.

Če v besedilu naloge ni drugače navedeno, lahko tekmovalčeva rešitev vedno predpostavi, da so vhodni podatki, s katerimi dela, podani v takšni obliki in v okviru takšnih omejitev, kot jih zagotavlja naloga. Tekmovalcem torej načeloma ni treba pisati rešitev, ki bi bile odporne na razne napake v vhodnih podatkih.

V nadaljevanju podajamo še nekaj nasvetov za ocenjevanje pri posameznih nalogah.

1. Nadležne besede

- Rešitev sme predpostaviti, da je v vhodnih podatkih vsaj ena beseda (torej da seznam vhodnih besed ni prazen).
- Naš primer rešitve bere vhodne podatke do EOF, enako dobre pa so tudi rešitve, ki predpostavijo, da je konec podatkov označen kako drugače, na primer s prazno vrstico, ali pa da je na začetku vhoda podano najprej število besed.
- Rešitvam, ki po nepotrebnem preberejo celoten seznam vhodnih besed v pomnilnik (namesto da bi jih brale in obdelovale sproti), naj se zaradi tega odšteje tri točke.
- Rešitve smejo predpostaviti, da nobena vhodna beseda ni prazna. Z drugimi besedami, vsaka vhodna beseda je dolga vsaj en znak.
- Če ima rešitev pravilno zastavljen mehanizem za ugotavljanje, s katero tipko se tipka določeno črko, vendar ima kakšno napako v s tem povezanih konstantah (kot je npr. tabela tipka v naši rešitvi), naj se ji zaradi tega odšteje največ dve točki.

2. Prepisovanje

- Naša rešitev pri tej nalogi sproti bere in pozablja števila iz vhodne datoteke; za enako dobro pa naj velja tudi rešitev, ki bi vsa števila prebrala v tabelo v pomnilniku in jih šele potem začela obdelovati.
- Mogoče je, da so osumljeni trije (ali več) zaporedni učenci. Rešitvam, ki pri štetju osumljenih v takem primeru srednjega od teh učencev štejejo dvojno, naj se zaradi tega odšteje sedem točk.
- Naloga pravi, da učence osumimo prepisovanja, če se dve sosednji števili razlikujeta za t ali manj. Če rešitev pomotoma sumi na prepisovanje le, če se števili razlikujeta za strogo manj kot t, ne pa tudi takrat, ko se razlikujeta za natanko t, naj se ji zaradi tega odšteje dve točki.
- Če poskuša rešitev pomotoma primerjati s prejšnjim učencem tudi prvega (ki prejšnjega učenca sploh nima), naj se ji zaradi tega odšteje tri točke.
- V naši rešitvi za posebno obravnavo prvega učenca poskrbimo tako, da preverjamo, če je indeks i že večji od 0 ali še ne. Enako dobra možnost bi bila tudi, da bi spremenljivko prejsnji pred glavno zanko inicializirali na neko vrednost, ki je gotovo za več kot t različna od katerega koli veljavnega števila (na primer na $10^9 + t + 1$).
- Če bi slučajno kakšna rešitev imela časovno zahtevnost $O(n^2)$ namesto le $O(n)$, naj dobi največ 13 točk (če je drugače pravilna).

3. Riziko

- Pri tej rešitvi podrobnosti branja vhodnih podatkov in izpisa rezultatov niso pomembne, tako da je vseeno, kako program bere vhodne podatke, ali pri tem tudi še kaj izpiše (npr. vprašanja uporabniku, ki naj bi vnašal podatke prek konzole), kako točno je formatiran izpis rezultatov ipd. Enako dobra bi bila tudi rešitev, ki bi predpostavila, da so podatki že na voljo v nekih spremenljivkah oz. kot parametri podprograma, pomembno je le, da ne predpostavi, da so podatki že urejeni.

- Če rešitev uredi kocke posameznega igralca naraščajoče namesto padajoče, naj se ji zaradi tega odšteje dve točki. Če na urejanje popolnoma pozabi, naj se ji odšteje pet točk.

- Naša rešitev ureja kocke vsakega od igralcev sama, enako dobro pa je tudi, če rešitev za urejanje uporabi kakšne funkcije iz standardne knjižnice svojega programskega jezika.

- Če rešitev sama implementira urejanje, je vseeno, po kakšnem postopku ureja (če je le rezultat pravilen). Naš primer rešitve uporablja bubble sort, možni pa so seveda še mnogi drugi postopki.

4. Eksplozija

- Poudarek pri tej nalogi je na ideji rešitve, ne na podrobnostih implementacije (in še posebej ne na obravnavanju morebitnih zaokrožitvenih napak pri delu s števili s plavajočo vejico). Pomembno pa je, da tekmovalčev odgovor vsebuje nekakšno utemeljitev, iz katere je razvidno, da je tekmovalec razumel, *zakaj* je njegov postopek pravilen, in ne le, da ima pač občutek, da je pravilen.

- Poleg postopka, ki je opisan v naših rešitvah, so seveda možne še drugačne rešitve, ki tudi lahko dosežejo vse točke, če dajejo pravilne rezultate (in so dobro utemeljene). Na primer, ena možnost je, da točke (po eksploziji) uredimo po y, jih v tem vrstnem redu razdelimo na skupine po 5 točk in nato vsako skupino uredimo še po x. Pokazati je mogoče, da tako dobimo enak vrstni red točk, kot če bi enako urejanje izvedli pred eksplozijo — najprej imamo torej koordinate predmeta, ki je bil pred eksplozijo na $(0, 0)$, nato tistega, ki je bil pred eksplozijo na $(0, 1)$ in tako naprej. Zdaj lahko za vsak predmet potegnemo premico skozi njegov prvotni položaj in njegov položaj po eksploziji ter poiščemo presečišče vseh tako dobljenih premic; tam je prišlo do eksplozije.

- Opazimo lahko, da je pri tej nalogi mogočih pravzaprav le 16 položajev eksplozije. Možna rešitev je torej tudi ta, da za vsako od teh 16 možnosti izračuna položaje vseh točk po eksploziji, nato pa vhodni nabor 25 točk primerja z vsemi 16 razporedi, da vidi, s katerim se ujema (pri tem mora paziti tudi na dejstvo, da so točke v vhodnih podatkih lahko poljubno premešane). Tudi taka rešitev lahko dobi vse točke, če je pravilno zamišljena in utemeljena. Če je

primerjanje vhodnega razporeda z ostalimi 16 razporedi izvedeno na zelo neučinkovit način (na primer tako, da vhodni razpored premešamo na 25! možnih načinov), naj taka rešitev dobi največ 10 točk (če je drugače pravilna).

- Naloga pravi, da so predmeti v naših vhodnih podatkih lahko poljubno premešani. Rešitev, ki predpostavi, da so koordinate predmetov po eksploziji navedene v nekem konkretnem (in koristnem) vrstnem redu (npr. tako, da najprej pride položaj predmeta, ki je bil pred eksplozijo na $(0, 0)$, nato položaj predmeta, ki je bil pred eksplozijo na $(0, 1)$ itd.), naj dobi največ 13 točk (če je drugače pravilna).

5. Barvanje plošče

- Za vsak primer poudarimo, da naloga ne zahteva, naj rešitev izpiše ali kako drugače zgenerira konkretno zaporedje operacij, ki bi ustrezno pobarvalo ploščo — rešitev mora le izračunati najmanjše potrebno število operacij.

- Postopek, kot smo ga opisali v C-ju na koncu naše rešitve, ima časovno zahtevnost $O(n)$. Z malo manj pazljivosti pri implementaciji zanke, ki mora pregledati vse b in pri vsakem najti primeren a, si lahko predstavljamo rešitve s časovno zahtevnostjo $O(n^2)$ ali celo $O(n^3)$. Take rešitve naj pri tej nalogi dobijo največ 18 točk (če so drugače pravilne).

- Glede utemeljitve pravilnosti rešitve pričakujemo predvsem nekakšen razmislek o tem, da nam smeri obračanja plošče ni treba spremeniti več kot enkrat.

- Ekstremno neučinkovite rešitve — npr. če bi nekdo generiral vsa možna zaporedja $2n$ ali manj operacij in preverjal, če pravilno pobarvajo ploščo (in med takimi izpisal dolžino najkrajšega) — naj dobijo največ 8 točk (če so drugače pravilne).

- Če rešitev ne deluje za poljuben z, ampak le za eno konkretno vrednost z-ja (na primer $z = 0$) in pri tem ne razloži, kako lahko vhodne primere z drugačnim z predelamo na to konkretno vrednost, naj se ji odšteje dve točki.

Težavnost nalog

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

Naloga	Kam bi sodila po težavnosti na državnem tekmovanju ACM
1. Nadležne besede	lažja do srednja naloga v prvi skupini
2. Prepisovanje	srednja v prvi ali lahka naloga v drugi skupini
3. Riziko	srednja v prvi ali lažja naloga v drugi skupini
4. Eksplozija	srednja do težja naloga v drugi skupini
5. Barvanje plošče	težja v drugi ali srednja v tretji skupini

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

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

REZULTATI

Tabele na naslednjih straneh prikazujejo vrstni red vseh tekmovalcev, ki so sodelovali na letošnjem tekmovanju. Poleg skupnega števila doseženih točk je za vsakega tekmovalca navedeno tudi število točk, ki jih je dosegel pri posamezni nalogi. V prvi in drugi skupini je mogoče pri vsaki nalogi doseči največ 20 točk, v tretji skupini pa največ 100 točk.

Načeloma se v vsaki skupini podeli dve prvi, dve drugi in dve tretji nagradi, letos pa so se rezultati izšli tako, da smo v prvi skupini izjemoma podelili tri prve in štiri tretje nagrade. Poleg nagrad na državnem tekmovanju v skladu s pravilnikom podeljujemo tudi zlata in srebrna priznanja. Število zlatih priznanj je omejeno na eno priznanje na vsakih 25 udeležencev šolskega tekmovanja (teh je bilo letos 310) in smo jih letos podelili enajst. Srebrna priznanja pa se podeljujejo po podobnih kriterijih kot pred leti pohvale; prejmejo jih tekmovalci, ki ustrezajo naslednjim trem pogojem: (1) tekmovalec ni dobil zlatega priznanja; (2) je boljši od vsaj polovice tekmovalcev v svoji skupini; in (3) je tekmoval v prvi ali drugi skupini in dobil vsaj 20 točk ali pa je tekmoval v tretji skupini in dobil vsaj 80 točk. Namen srebrnih priznanj je, da izkažemo priznanje in spodbudo vsem, ki se po rezultatu prebijejo v zgornjo polovico svoje skupine. Podobno prakso poznajo tudi na nekaterih mednarodnih tekmovanjih; na primer, na mednarodni računalniški olimpijadi (IOI) prejme medalje kar polovica vseh udeležencev. Poleg zlatih in srebrnih priznanj obstajajo tudi bronasta, ta pa so dobili najboljši tekmovalci v okviru šolskih tekmovanj (letos smo podelili 144 bronastih priznanj).

V tabelah na naslednjih straneh so prejemniki nagrad označeni z „1“, „2“ in „3“ v prvem stolpcu, prejemniki priznanj pa z „Z“ (zlato) in „S“ (srebrno).

PRVA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Tomaž Martinčič	4	STPŠ Trbovlje	16	20	19	19	20	94
1Z	2	Matej Bevec	3	Gimnazija Bežigrad	20	20	14	18	20	92
1Z		Gregor Kikelj	2	ŠC Novo mesto, SEŠTG	20	20	16	19	17	92
2Z	4	Aljaž Kolar	2	ŠC Kranj, Str. gim.	20	20	14	19	17	90
2Z	5	Timi Korda Mlakar	3	SERŠ Maribor	18	20	15	19	17	89
3S	6	Bor Grošelj Simić	1	Gimnazija Vič in ZRI	20	20	5	20	20	85
3S		Martin Dagarin	2	Vegova Ljubljana	20	20	14	19	12	85
3S		Rok Strah	2	Vegova Ljubljana	18	20	12	20	15	85
3S	9	Luka Jevšenak	2	Gimnazija Velenje	19	20	8	20	17	84
S	10	Matic Gačar	4	SERŠ Maribor	18	20	10	18	17	83
S	11	Martin Prelog	2	ŠC Kranj, STŠ	20	20	5	20	17	82
S		Jernej Grlj	2	Škof. klas. gimn. Lj.	15	20	8	19	20	82
S	13	Januš Likozar	4	ŠC Kranj, Str. gim.	20	10	11	19	19	79
S		Zen Lednik	4	ŠC Celje, SŠ za KER	20	20	5	17	17	79
S	15	Domen Antlej	4	ŠC Celje, Gimn. Lava	20	20	1	20	17	78
S		Luka Železnik	1	II. gimnazija Maribor	16	20	11	14	17	78
S		Jernej Leskovšek	3	STPŠ Trbovlje	15	20	7	19	17	78
S	18	Nejc Jezeršek	1	Vegova Ljubljana	15	19	8	17	17	76
S		Janko Knez	4	ŠC Celje, Gimn. Lava	19	20	1	20	16	76
S		Katja Logar	4	Škof. klas. gimn. Lj.	18	15	15	11	17	76
S	21	Uroš Šmajdek	3	ŠC Novo mesto, SEŠTG	20	20	5	18	12	75
S		Marko Kužner	3	SERŠ Maribor	14	20	9	15	17	75
S	23	Simon Klemenčič	3	ŠC Kranj, STŠ	16	20	3	14	20	73
S		Mihael Trajbarič	4	Škof. klas. gimn. Lj.	20	20	1	20	12	73
S	25	Gregor Štefanič	4	ŠC Novo mesto, SEŠTG	15	20	6	19	12	72
S	26	Jakob Zmrzlikar	2	Gimnazija Vič	18	11	4	19	19	71
S	27	Jakob Jesenko	2	Škof. klas. gimn. Lj.	14	20	2	17	17	70
S		Urh Primožič	1	Škof. klas. gimn. Lj.	20	20	4	16	10	70
S	29	Gašper Golob	2	Vegova Ljubljana	20	2	15	20	12	69
S		Jon Mikoš	1	Gimnazija Vič	13	12	14	15	15	69
S	31	Marcel Mumel	3	SERŠ Maribor	20	3	13	15	17	68
S	32	Miha Marinko	1	Gimnazija Vič	16	17	9	12	12	66
S		Marko Hostnik	2	Gimnazija Bežigrad	17	7	10	20	12	66
S	34	Kevin Šarlah	2	ŠC Celje, SŠ za KER	17	8	9	19	12	65
S	35	Miha Frangež	1	SERŠ Maribor	13	7	8	19	17	64
S	36	Jan Uršič	4	Škof. klas. gimn. Lj.	19	19	1	14	10	63
S		Aljaž Zakošek	3	I. gimnazija v Celju	6	20	6	19	12	63
S		Domen Rostohar	3	ŠC Novo mesto, SEŠTG	18	6	13	16	10	63
S		Samo Debeljak	1	Gimnazija Vič in ZRI	17	18	2	9	17	63

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
S	40	David Grabnar	2	Vegova Ljubljana	8	10	7	19	17	61
S		Gregor Štefanič	4	SERŠ Maribor	20	2	5	17	17	61
S		Matic Dokl	3	II. gimnazija Maribor	6	20	5	14	16	61
S	43	Vid Jerovšek	4	ŠC Novo mesto, SEŠTG	4	20	1	18	17	60
S		Mark Podlinšek	1	ZRI	12	7	8	17	16	60
S	45	Žan Bizjak	3	Vegova Ljubljana	19	0	6	13	20	58
S	46	Matjaž Ciglič	3	Gimnazija Vič	10	16	0	13	17	56
S		Matej Pevec	1	Vegova Ljubljana	1	9	10	19	17	56
S		Tomaž Jerman	4	STPŠ Trbovlje	4	7	14	19	12	56
S	49	Miha Bogataj	4	ŠC Kranj, Str. gim.	14	3	7	14	17	55
S	50	Matic Conradi	1	I. gimnazija v Celju	16	1	8	17	12	54
S		Lucian Semprimožnik	1	I. gimnazija v Celju	17	6	1	13	17	54
S	52	Nejc Kozjek	3	ŠC Kranj, STŠ	2	7	7	19	17	52
S		David Murko	3	ŠC Ptuj, ERŠ	13	7	5	10	17	52
	54	Alja Kolenc	4	ŠC Celje, Gimn. Lava	10	4	5	15	17	51
		Domen Ramšak	2	ŠC Velenje, ERŠ	20	0	1	13	17	51
	56	Mark Valentin Vovk	3	Gimnazija Poljane	15	1	7	15	11	49
	57	Miha Pompe	1	Gimnazija Vič	12	5	8	11	12	48
	58	David Lunar	3	ŠC Kranj, STŠ	2	7	5	15	17	46
		Gregor Brantuša	2	II. gimnazija Maribor	3	0	10	13	20	46
		Gašper Jalen	2	Škof. klas. gimn. Lj.	0	10	1	18	17	46
	61	Boris Pirečnik	2	ŠC Velenje, ERŠ	6	0	12	18	9	45
		Gašper Močnik	3	ŠC Novo mesto, SEŠTG	3	5	6	15	16	45
	63	Aljaž Žel	2	II. gimnazija Maribor	2	20	3	17	2	44
		Miha Krajnc	1	STPŠ Trbovlje	15	6	6	16	1	44
	65	Luka Miklavčič	3	Gimnazija Vič	12	6	6	3	15	42
		Blaž Košir	1	Gimnazija Vič	15	0	7	10	10	42
		Andreja Kernc	3	ŠC Novo mesto, SEŠTG	2	5	9	18	8	42
	68	Rok Šerak	2	STPŠ Trbovlje	14	1	4	15	6	40
	69	Žiga Deutchbauer	2	ŠC Velenje, ERŠ	0	3	5	12	17	37
	70	Lenart Kovač	3	I. gimnazija v Celju	0	6	4	14	12	36
		Domen Dolanc	4	STPŠ Trbovlje	10	0	6	13	7	36
		Janez Turnšek	4	ŠC Celje, Gimn. Lava	17	7	0	12	0	36
	73	Andreja Merše	4	Škof. klas. gimn. Lj.	2	5	6	12	10	35
	74	Zoran Uran	1	II. gimnazija Maribor	0	0	6	11	17	34
	75	Andraž Čeh	2	ŠC Ptuj, ERŠ	17	7	2	5	2	33
		Luka Laharnar	2	STPŠ Trbovlje	2	2	3	14	12	33
	77	Mitja Brezovnik	4	ŠC Celje, SŠ za KER	0	0	1	17	14	32
	78	Jan Ivkovič	3	Gimnazija Bežigrad	0	20	3	5	3	31
		Aljoša Kalacanović	1	Gimnazija Vič	14	5	0	11	1	31

(*nadaljevanje na naslednji strani*)

PRVA SKUPINA (*nadaljevanje*)

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
	80	Jani Kaukler	3	SERŠ Maribor	3	2	5	10	8	28
	81	Matic Hrastelj	2	STPŠ Trbovlje	1	0	1	12	13	27
		Matej Zečiri	1	STPŠ Trbovlje	2	1	3	12	9	27
	83	Urban Matko	3	SŠ Domžale	0	0	5	4	17	26
		Urška Jagarinec	3	Gimnazija Vič	2	2	1	14	7	26
		Jernej Jerebic	1	SPTŠ Murska Sobota	2	1	1	12	10	26
	86	Niko Kolar	4	ŠC Celje, SŠ za KER	0	3	0	12	7	22
	87	Boštjan Planko	2	ŠC Celje, SŠ za KER	17	3	0	1	0	21
	88	Vid Tilen Ratajec	1	ZRI	7	0	0	3	10	20
	89	Gregor Rant	3	ŠC Kranj, STŠ	2	3	4	2	8	19
		David Konc	4	Gimnazija Kranj	1	1	1	3	13	19
	91	Miha Gošte	2	STPŠ Trbovlje	1	0	1	5	11	18
		Severin Živic	1	ZRI	3	0	0	4	11	18
	93	Miha Zidar	4	STPŠ Trbovlje	2	2	0	10	3	17
	94	Žak Paradižnik	2	ŠC Celje, SŠ za KER	2	0	0	10	4	16
	95	Vasja Drnovšček	3	ŠC Nova Gorica	0	2	4	4	5	15
		David Trafela	2	II. gimnazija Maribor	0	0	0	0	15	15
	97	Žan Rogan	2	SPTŠ Murska Sobota	1	0	0	1	12	14
	98	Timotej Petrovčič	1	Gimnazija Vič	2	4	1	4	1	12
		David Bejek	2	ŠC Velenje, ERŠ	1	0	0	3	8	12
	100	Erik Toplak	2	SPTŠ Murska Sobota	0	0	2	1	8	11
		Julijana Djordjević	3	SŠ Domžale	0	0	5	5	1	11
	102	Tian Breznik	3	II. gimnazija Maribor	10	0	0	0	0	10
	103	Tomaž Klopčič	4	SŠ Domžale	6	0	0	0	1	7
		Matej Volkar	1	SŠ Domžale	2	0	0	0	5	7
		Miha Pavlič	3	SŠ Domžale	2	5	0	0	0	7
	106	Matic Jeseničnik	2	ŠC Velenje, ERŠ	0	2	0	0	0	2

DRUGA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Žiga Vene	4	Gimnazija Krško	18	19	15	19	14	85
1Z	2	Mitja Žalik	3	II. gimnazija Maribor	16	18	13	20	17	84
2Z	3	Matej Marinko	3	Gimnazija Vič	11	15	18	20	19	83
2S	4	Marko Rus	4	Škof. klas. gimn. Lj.	16	20	10	20	5	71
3S	5	David Popović	3	Gimnazija Bežigrad	20	5	20	7	18	70
3S	6	Urban Duh	2	II. gimnazija Maribor	17	14	12	17	8	68
S	7	Rok Cej	4	Gimnazija Bežigrad	18	13	18	7	11	67
S	8	Matic Rašl	3	ŠC Ptuj, ERŠ	0	18	20	15	13	66
S	9	Timotej Knez	4	Škof. klas. gimn. Lj.	3	19	18	20	5	65
S	10	Andraž Juvan	3	Vegova Ljubljana	14	12	20	15	3	64
S		Jure Bevc	4	Gimnazija Krško	4	17	17	11	15	64
S	12	Žiga Klemenčič	3	Vegova Ljubljana	3	18	13	20	5	59
S	13	Blaž Zupančič	2	Škof. klas. gimn. Lj.	3	7	20	18	9	57
S	14	Matija Lazić	4	Vegova Ljubljana	0	20	10	18	5	53
S		David Pintarič	4	SPTŠ Murska Sobota	3	14	15	16	5	53
S	16	Oton Pavlič	4	Škof. klas. gimn. Lj.	10	15	13	10	4	52
S	17	Tomaž Hribernik	3	Gimnazija Kranj	7	15	13	16	0	51
S	18	Leonard Logarič	3	Gimnazija Bežigrad	12	0	13	7	18	50
S	19	Žiga Patačko Koderman	3	Gimnazija Vič	5	7	12	19	4	47
	20	Blaž Novak	2	SŠ Ravne na Koroškem	0	18	10	0	15	43
	21	Jon Kuhar	2	Gimnazija Kranj	5	10	13	14	0	42
	22	Jure Hudoklin	4	Gimnazija Bežigrad	3	18	13	1	5	40
		Matic Šincek	4	ŠC Velenje, ERŠ	15	15	0	5	5	40
	24	Vid Drobnič	3	Gimnazija Vič	4	10	13	7	4	38
	25	Blaž Rojc	4	Gimnazija Nova Gorica	5	7	20	0	2	34
	26	Peter Bohanec	4	Gimn. Bežigrad in ZRI	3	0	17	8	5	33
	27	Jaka Jenko	4	ŠC Velenje, ERŠ	0	15	16	0	0	31
	28	Vid Klopčič	3	Gimnazija Vič	4	0	2	18	4	28
		Primož Fabiani	3	ZRI	0	18	10	0	0	28
	30	Luka Brezavšček	4	ŠC Nova Gorica	3	14	5	0	0	22
	31	Miloš Kostadinovski	4	ŠC Nova Gorica	4	7	10	0	0	21
		Jaka Basej	2	ZRI	2	0	10	7	2	21
	33	Miha Breznik	2	SŠ Ravne na Koroškem	5	0	11	0	2	18
	34	Denis Krajnc	3	SŠ Ravne na Koroškem	1	8	8	0	0	17
	35	Rok Mori	4	SŠ Ravne na Koroškem	5	3	8	0	0	16
	36	Matevž Gros	2	ZRI	0	10	2	1	0	13
		Jure Pavlin	4	SŠ Domžale	2	0	11	0	0	13
	38	David Rozman	2	ZRI	1	0	0	1	0	2

TRETJA SKUPINA

Nagrada	Mesto	Ime	Letnik	Šola	Točke (po nalogah in skupaj) 1	2	3	4	5	$\sum$
1Z	1	Mihail Denkovski	3	II. gimnazija Maribor	51	67	40	77	42	277
1Z	2	Aleksej Jurca	3	Gimnazija Bežigrad	100		60	90		250
2Z	3	Žiga Željko	3	Gimn. Bežigrad in ZRI	100		30	40	55	225
2S	4	Tim Poštuvan	2	ZRI	94	0	8	34	40	176
3S	5	Rok Krumpak	4	ŠC Celje, Gimn. Lava	80		20	0	65	165
3S	6	Martin Peterlin	3	Vegova Lj. in ZRI	77		37		50	164
S	7	Bor Brecelj	4	Gimnazija Vič in ZRI	100		20	0	40	160
S	8	Rok Kos	4	Gimnazija Vič in ZRI	47		20	55	10	132
	9	Tadej Gašparovič	9	OŠ J. Krajca Rakek	90	0	0	0	30	120
	10	Aljaž Eržen	4	Vegova Lj. in ZRI	84	0	0	0	28	112
	11	Tadej Šinko	3	SPTŠ Murska Sobota	77		20			97
	12	Luka Govedič	2	II. gimnazija Maribor	94		0		0	94
	13	Benjamin Benčina	4	Gimn. Bežigrad in ZRI	48				24	72
	14	Tevž Murkovič	4	Vegova Ljubljana	37		0	0	10	47
	15	Miha Mitič	2	Gimnazija Kranj	10	0	0		30	40
	16	Mark Marinček	4	Gimnazija Poljane			0			0
		David Ocepek	4	Gimnazija Bežigrad						0

NAGRADE

Za nagrado so najboljši tekmovalci vsake skupine prejeli naslednjo strojno opremo in knjižne nagrade:

Skupina	Nagrada	Nagrajenec	Nagrade
1	1	Tomaž Martinčič	tablični računalnik Acer Iconia One 10
1	1	Matej Bevec	tablični računalnik Acer Iconia One 10
1	1	Gregor Kikelj	3 TB zunanji disk
1	2	Aljaž Kolar	3 TB zunanji disk
1	2	Timi Korda Mlakar	2 TB zunanji disk
1	3	Bor Grošelj Simić	2 TB zunanji disk
1	3	Martin Dagarin	miška Razer DeathAdder Chroma
1	3	Rok Strah	miška Razer DeathAdder Chroma
1	3	Luka Jevšenak	miška Razer DeathAdder Chroma
2	1	Žiga Vene	tablični računalnik Acer Iconia One 10 Raspberry Pi 3 model B Dasgupta *et al.: Algorithms*
2	1	Mitja Žalik	tablični računalnik Lark Evolution x2 Dasgupta *et al.: Algorithms*
2	2	Matej Marinko	2 TB zunanji disk Dasgupta *et al.: Algorithms*
2	2	Marko Rus	2 TB zunanji disk
2	3	David Popović	miška Razer DeathAdder Chroma
2	3	Urban Duh	miška Razer DeathAdder Chroma
3	1	Mihail Denkovski	tablični računalnik Acer Iconia One 10 Raspberry Pi 3 model B Halim: *Competitive Programming 3*
3	1	Aleksej Jurca	tablični računalnik Lark Evolution x2 Raspberry Pi 3 model B Halim: *Competitive Programming 3*
3	2	Žiga Željko	2 TB zunanji disk Halim: *Competitive Programming 3*
3	2	Tim Poštuvan	2 TB zunanji disk
3	3	Rok Krumpak	64 GB USB ključ
3	3	Martin Peterlin	64 GB USB ključ
Off-line naloga — Volilna območja			
	2	Matjaž Leonardis	Raspberry Pi 3 model B
	3	Žan Ninin	Raspberry Pi 3 model B

SODELUJOČE ŠOLE IN MENTORJI

II. gimnazija Maribor	Mirko Pešec
Gimnazija Bežigrad	Gregor Anželj, Andrej Šuštaršič
Gimnazija Kranj	Zdenka Vrbinc
Gimnazija Nova Gorica	Jurij Knez
Gimnazija Poljane	Janez Malovrh, Boštjan Žnidaršič
Gimnazija Vič	Klemen Bajec, Andrej Brodnik, Valentin Kragelj, Marina Trost
Osnovna šola Jožeta Krajca Rakek	Lidija Anzeljc
I. gimnazija v Celju	Matej Zdovc
Srednja elektro-računalniška šola Maribor (SERŠ)	Vida Motaln, Branko Potisk, Manja Sovič Potisk
Srednja poklicna in tehniška šola Murska Sobota (SPTŠ)	Igor Kutoš, Karel Maček, Boris Ribaš
Srednja šola Domžale	Tadej Trinko
Srednja šola Ravne	Gorazd Geč, Zdravko Pavleković
Srednja tehniška in poklicna šola Trbovlje (STPŠ)	Uroš Ocepek
Škofijska klasična gimnazija Šentvid	Helena Medvešek, Matej Tomc
Šolski center Celje, Gimnazija Lava	Karmen Kotnik, Tomislav Viher
Šolski center Celje, Srednja šola za kemijo, elektrotehniko in računalništvo (KER)	Gorazd Breznik, Dušan Fugina
Šolski center Kranj, Srednja tehniška šola	Aleš Hvasti
Šolski center Kranj, Strokovna gimnazija	Gašper Strniša
Šolski center Krško-Sevnica, Gimnazija Krško	Andrej Peklar
Šolski center Nova Gorica, Elektrotehniška in računalniška šola (ERŠ)	Tomaž Mavri, Boštjan Vouk
Šolski center Nova Gorica, Gimnazija in zdravstvena šola (GZŠ)	Barbara Pušnar
Šolski center Novo mesto, Srednja elektro šola in tehniška gimnazija (SEŠTG)	Albert Zorko, Simon Vovko

Šolski center Ptuj, Elektro in računalniška šola (ERŠ)	Marjan Čeh, Franc Vrbančič
Šolski center Velenje, Gimnazija Velenje	Miran Zevnik
Šolski center Velenje, Elektro in računalniška šola (ERŠ)	Miran Zevnik
Vegova Ljubljana	Marko Kastelic, Melita Kompolšek, Nataša Makarovič, Darjan Toth

Zavod za računalniško izobraževanje (ZRI), Ljubljana

OFF-LINE NALOGA — VOLILNA OBMOČJA

Na računalniških tekmovanjih, kot je naše, je čas reševanja nalog precej omejen in tekmovalci imajo za eno nalogo v povprečju le slabo uro časa. To med drugim pomeni, da je marsikak zanimiv problem s področja računalništva težko zastaviti v obliki, ki bi bila primerna za nalogo na tekmovanju; pa tudi tekmovalec si ne more privoščiti, da bi se v nalogo poglobil tako temeljito, kot bi se mogoče lahko, saj mu za to preprosto zmanjka časa.

Off-line naloga je poskus, da se tovrstnim omejitvam malo izognemo: besedilo naloge in testni primeri zanjo so objavljeni več mesecev vnaprej, tekmovalci pa ne oddajajo programa, ki rešuje nalogo, pač pa oddajajo rešitve tistih vnaprej objavljenih testnih primerov. Pri tem imajo torej veliko časa in priložnosti, da dobro razmislijo o nalogi, preizkusijo več možnih pristopov k reševanju, počasi izboljšujejo svojo rešitev in podobno. Opis naloge in testne primere smo objavili novembra 2015 skupaj z razpisom za tekmovanje v znanju; tekmovalci so imeli čas do 18. marca 2016 (dan pred tekmovanjem), da pošljejo svoje rešitve.

Opis naloge

Dana je karirasta mreža, ki predstavlja površino neke države. Za vsako celico (kvadratek) mreže je znano, koliko prebivalcev živi na njej. Naloga je razdeliti mrežo na določeno število volilnih območij tako, da so si volilna območja čim bolj podobna po številu prebivalcev. Natančneje povedano, radi bi, da bi bila razlika v številu prebivalcev med območjem z največ prebivalci in območjem z najmanj prebivalci čim manjša.

Primer: recimo, da imamo naslednjo mrežo s 4×3 celicami (števila v posameznih celicah pomenijo število prebivalcev).

3	3	2	1
2	2	7	0
3	4	1	1

Recimo, da moramo mrežo razdeliti na 3 volilna območja. Ena možnost je naslednja (debele črte označujejo meje med območji):

3	3	2	1
2	2	7	0
3	4	1	1

Pri tej razdelitvi imamo eno območje (zgornje) s $3 + 3 + 2 + 1 = 9$ prebivalci, eno območje (srednje) z $2 + 2 + 7 + 0 = 11$ prebivalci in eno območje (spodnje) s $3+4+1+1 = 9$ prebivalci. Razlika med območjem z največ in območjem z najmanj prebivalci je torej $11 - 9 = 2$.

Isto mrežo lahko razdelimo na območja tudi takole:

3	3	2	1
2	2	7	0
3	4	1	1

Pri tej razdelitvi ima zgornje levo območje 10 prebivalcev, spodnje levo 9 prebivalcev, desno območje pa 10 prebivalcev. Razlika med območjem z največ in območjem z najmanj prebivalci je tako $10 - 9 = 1$, zato je ta rešitev boljša od prejšnje (pri kateri je bila ta razlika 2).

Testni primeri

Pripravili smo 300 testnih primerov, pri vsakem od njih pa velja omejitev, da je velikost mreže največ 300×300. Število prebivalcev v posamezni celici mreže je vsaj 0 in največ 10^6, v celi mreži pa je skupaj naveč $2 \cdot 10^9$ prebivalcev. Število testnih primerov je veliko zato, ker smo hoteli odvrniti ljudi od oddajanja ročno sestavljenih razporedov.

Za razporejanje prebivalcev smo uporabili postopek po zgledu preferential attachmenta, ki se uporablja med drugim v modeliranju socialnih omrežij.[16] Prebivalce dodajamo na mrežo enega po enega; z verjetnostjo p se novi prebivalec naseli v bližino kakšnega od že obstoječih prebivalcev (to naredimo tako, da si naključno izberemo enega od že obstoječih prebivalcev in nato novega naselimo v isto celico ali pa v eno od njenih 8 sosed, pri čemer imajo vse te celice enako verjetnost, da bodo izbrane), z verjetnostjo $1 - p$ pa se ne ozira na obstoječe prebivalce in se naseli v naključno izbrano celico (pri čemer imajo vse celice enako verjetnost, da bodo izbrane).

Učinek tega postopka je, da če je p velik, se bodo prebivalci večinoma zgostili v nekaj predelih mreže (kot da bi bila tam velika mesta, drugod pa redko poseljeno podeželje); če pa je p majhen, bodo bolj enakomerno razporejeni po celi mreži. Različni testni primeri so imeli različne p-je, tako da so pokrili širok razpon možnosti od bolj zgoščene do bolj razpršene poselitve.

Rezultati

Sistem točkovanja je bil tak kot pri off-line nalogah v prejšnjih letih. Pri vsakem testnem primeru smo razvrstili tekmovalce po obsegu njihovega razporeda likov, nato pa je prvi tekmovalec (tisti z najmanjšim obsegom) dobil 10 točk, drugi 8, tretji 7 in tako naprej po eno točko manj za vsako naslednje mesto (osmi dobi dve točki, vsi nadaljnji pa po eno). Na koncu smo za vsakega tekmovalca sešteli njegove točke po vseh tristo testnih primerih.

Letos je svoje rešitve pri off-line nalogi poslalo kar osem tekmovalcev. Končna razvrstitev je naslednja:

[16] Gl. npr. Wikipedijo *s. v.* Preferential attachment.

Rok Kralj	2795
Matjaž Leonardis (U. v Oxfordu)	2423
Žan Ninin (ŠC Nova Gorica)	2292
Amon Stopinšek (FRI)	1795
Žiga Željko (Gim. Bežigrad)	937
Dean Cerin (FAMNIT)	476
Simon Gorše	51
Nejc Kadivnik (FRI)	4

Na spletni strani tekmovanja so objavljene tudi vizualizacije vseh prejetih rešitev vseh tekmovalcev (`http://rtk.ijs.si/2016/gerry/rtk2016-gerry-vse.pdf`).

Rešitev

Oglejmo si opis rešitve, ki ga je prispeval Tomaž Hočevar (tekmovanja se sicer ni uradno udeležil, če pa bi se ga, bi prepričljivo zmagal; njegove rešitve so bile pri 260 od 300 testnih primerov boljše od rešitev vseh ostalih tekmovalcev). Razdelitev volilnih območij poteka v dveh fazah. V prvi fazi zgradimo začetno razdelitev, ki jo nato v drugi fazi izboljšujemo. Oba koraka vsebujeta element naključnosti, zato celotno rešitev večkrat ponovimo in izberemo najboljši rezultat. V nekaterih primerih je kvaliteta rešitve zelo odvisna od rezultata prve faze. Ob neugodni začetni razdelitvi tudi z izboljšavami ne pridemo daleč.

Za izdelavo začetne razdelitve območij izberemo za vsako območje naključno celico, ki predstavlja zametek tega območja. Nato območja postopoma rastejo, dokler niso zasedene vse celice v mreži. Na vsakem koraku te rasti razširimo območje s trenutno najmanjšim številom prebivalcev. To storimo tako, da mu dodelimo naključno izmed prostih celic, na katere to območje trenutno meji. Tako ves čas ohranjamo povezana območja.

Nato razdelitev območij izboljšujemo z naključno izmenjavo sosednjih celic, ki pripadata različnim območjem. Pri tem moramo seveda paziti, da katerega od območij ne razdelimo na več nepovezanih komponent. Če s tako menjavo izboljšamo rezultat, zamenjavo obdržimo, sicer jo razveljavimo.

Pri tem ne optimiziramo neposredno rezultata (razlike med največjim in najmanjšim številom prebivalcev v območju), ker zelo malo potez vodi do take izboljšave. Namesto tega optimiziramo funkcijo $f(\mathbf{p}) = (\max_i p_i - \min_i p_i)^2 + \sum_i (\bar{\mathbf{p}} - p_i)^2$, kjer je $\mathbf{p} = (p_1, \ldots, p_d)$ vektor s števili prebivalcev v posameznih območjih, p_i je število prebivalcev v i-tem območju, $\bar{\mathbf{p}} = (\sum_i p_i)/d$ pa je povprečno število prebivalcev na območje. Ta funkcija torej teži k čim bolj enakomerni razporeditvi prebivalcev po območjih (čim manjše odstopanje od povprečja), kar posredno izboljšuje tudi razliko med najbolj in najmanj poseljenim območjem.

Včasih je nemogoče z eno samo zamenjavo doseči izboljšanje rezultata (dosežemo lokalni optimum), zato lahko preizkušamo tudi daljša zaporedja zamenjav. Zamenjamo lahko npr. sosednji celici med območjema A in B ter celici med območjema B in C. Izkazalo se je, da so take verižne zamenjave smiselne vse tja do dolžine 5. Na začetku optimizacije se osredotočimo na krajše verige, kasneje pa na vedno daljše.

Iz lokalnih optimumov se rešujemo še s t.i. simuliranim ohlajanjem (*simulated annealing*). Gre pravzaprav za izboljšavo prej opisanega postopka inkrementalnih izboljšav (*hill-climbing*) z vpeljavo naključnosti. Pri simuliranem ohlajanju včasih

sprejmemo tudi spremembo, ki vodi do znižanje optimizacijske funkcije. Verjetnost take poteze je odvisna od tega, koliko smo poslabšali optimizacijsko funkcijo oz. rezultat in od časa, ki smo ga že porabili za optimizacijo. Večje ko je poslabšanje in dlje ko že teče optimizacija, nižja je verjetnost sprejetja take spremembe. Dejanska formulacija te verjetnosti je odvisna od primera uporabe in implementacije rešitve in nemalokrat vključuje kakšne čarobne konstante, do katerih se avtor dokoplje z eksperimentiranjem. Najboljša rešitev je npr. uporabljala funkcijo $p = 1 - (0{,}9 + 0{,}1 \cdot (t^{0{,}1} + f^{0{,}01}))$, kjer je t pretečeni čas in f relativno poslabšanje rezultata.

UNIVERZITETNI PROGRAMERSKI MARATON

Društvo ACM Slovenija sodeluje tudi pri pripravi študenskih tekmovanj v programiranju, ki v zadnjih letih potekajo pod imenom Univerzitetni programerski maraton (UPM, `www.upm.si`) in so odskočna deska za udeležbo na ACMovih mednarodnih študentskih tekmovanjih v programiranju (International Collegiate Programming Contest, ICPC). Ker UPM ne izdaja samostojnega biltena, bomo na tem mestu na kratko predstavili to tekmovanje in njegove letošnje rezultate.

Na študentskih tekmovanjih ACM v programiranju tekmovalci ne nastopajo kot posamezniki, pač pa kot ekipe, ki jih sestavljajo po največ trije člani. Vsaka ekipa ima med tekmovanjem na voljo samo en računalnik. Naloge so podobne tistim iz tretje skupine našega srednješolskega tekmovanja, le da so včasih malo težje oz. predvsem predpostavljajo, da imajo reševalci že nekaj več znanja matematike in algoritmov, ker so to stvari, ki so jih večinoma slišali v prvem letu ali dveh študija. Časa za tekmovanje je pet ur, nalog pa je praviloma 6 do 8, kar je več, kot jih je običajna ekipa zmožna v tem času rešiti. Za razliko od našega srednješolskega tekmovanja pri študentskem tekmovanju niso priznane delno rešene naloge; naloga velja za rešeno šele, če program pravilno reši vse njene testne primere. Ekipe se razvrsti po številu rešenih nalog, če pa jih ima več enako število rešenih nalog, se jih razvrsti po času oddaje. Za vsako uspešno rešeno nalogo se šteje čas od začetka tekmovanja do uspešne oddaje pri tej nalogi, prišteje pa se še po 20 minut za vsako neuspešno oddajo pri tej nalogi. Tako dobljeni časi se seštejejo po vseh uspešno rešenih nalogah in ekipe z istim številom rešenih nalog se potem razvrsti po skupnem času (manjši ko je skupni čas, boljša je uvrstitev).

UPM poteka v štirih krogih (dva spomladi in dva jeseni), pri čemer se za končno razvrstitev pri vsaki ekipi zavrže najslabši rezultat iz prvih treh krogov, četrti (finalni) krog pa se šteje dvojno. Najboljše ekipe se uvrstijo na srednjeevropsko regijsko tekmovanje (CERC, ki je bilo letos 18.–20. novembra 2016 v Zagrebu), najboljše ekipe s tega pa na zaključno svetovno tekmovanje (ki bo 20.–25. maja 2017 v Rapid Cityu, ZDA).

Na letošnjem UPM je sodelovalo 62 ekip s skupno 180 tekmovalci, ki so prišli s treh slovenskih univerz, nekaj pa je bilo celo srednješolcev. Tabela na naslednjih dveh straneh prikazuje vse ekipe, ki so se pojavile na vsaj enem krogu tekmovanja.

	Ekipa	Št. rešenih nalog*	Čas
1	Vid Kocijan, Patrik Zajec (FMF + FRI), Jasna Urbančič (FRI)	19	24:36:18
2	Žiga Željko (Gim. Bežigrad), Aljaž Eržen (Vegova Lj.), Žan Knafelc (FDV)	16	19:08:45
3	Jure Slak, Maks Kolman, Žiga Gosar (FMF)	16	24:21:52
4	Sven Cerk, Martin Šušterič (FMF + FRI), Miha Eleršič (FRI)	15	23:43:45
5	Daniel Silađi, Vladan Jovičić, Marko Palangetić (FAMNIT)	14	18:28:20
6	Alexei Drake, Andraž Dobnikar (FRI)	13	23:00:01
7	Marko Ljubotina, Mateja Hrast (FMF)	13	24:24:53
8	Anja Petković, Žiga Lukšič, Vesna Iršič (FMF)	12	18:05:59
9	Žan Skamljič, Domen Vidovič, Dominik Korošec (FERI)	11	15:13:27
10	Filip Koprivec (FMF), Filip Peter Lebar (FE), Luka Kolar (FRI)	11	17:32:29
11	Erik Langerholc (FMF), Tadej Ciglarič, Domen Lušina (FRI)	11	21:47:53
12	Nejc Kadivnik, Miha Zadravec (FRI)	10	11:33:23
13	Žiga Zupančič, Juš Kosmač, Eva Zmazek (FMF)	9	13:44:31
14	Vid Drobnič, Žiga Patačko Koderman, Matej Marinko (Gim. Vič)	9	14:19:05
15	Robert Koprivnik, Jan Maleš, Matej Drobnič (FERI)	9	15:29:18
16	Lara Jerman (FKKT), Klara Nosan (FRI), Lidija Magdevska (FMF + FRI)	9	16:46:16
17	Martin Turk, Jan Geršak, Jernej Katanec (FRI + FMF)	9	19:52:03
18	Sandi Mikuš, Žiga Lesar, Sara Kužni (FRI)	7	8:30:52
19	Rok Kos, Bor Brecelj (Gim. Vič), Benjamin Benčina (Gim. Bežigrad)	7	9:33:35
20	Petra Čotar, Jan Rozman, Rok Venturini (FMF)	7	10:31:11
21	Gal Meznarič, Jan Mikolič, Aljaž Jeromel (FERI)	7	12:01:03
22	Dejan Skledar, Klemen Forstnerič, Jan Pomer (FERI)	6	11:26:57
23	Dan Toškan, Deni Cerovac, Alex Smuk (STŠ Koper)	6	11:33:58
24	Leo Gombač, Jan Grbac, Mirt Hlaj (FAMNIT)	5	3:55:33
25	Tadej Novak (FMF), Mitja Rozman (FMF + FRI), Dejan Krejić (Pedag. f.)	5	7:53:22
26	Jure Kolenko, Milutin Spasić, Ernest Beličič (FRI)	5	8:28:44
27	Aljoša Mrak, Jan Markočič, Marko Čavdek (FRI)	5	9:35:05
28	Izak Glasenčnik, Leon Pahole, Robi Novak (FERI)	5	10:14:32
29	Primož Godec, Manca Žerovnik, Ožbolt Menegatti (FRI)	5	11:43:22
30	Rok Fortuna, Urban Marovt (FRI)	5	12:07:51
31	Andrej Dolenc, Peter Lazar (FMF + FRI), Peter Us (FRI)	4	3:22:10
32	Tim Poštuvan, Bor Grošelj Simić (Gim. Vič), Martin Peterlin (Vegova Lj.)	4	4:45:49
33	Žiga Šmelcer (FE), Žiga Gradišar (FMF), Lojze Žust (FRI)	4	5:45:23
34	Luka Avbreht, Samo Kralj, Živa Urbančič (FMF)	4	7:06:44
35	Ines Meršak, Ana Borovac, Matic Oskar Hajšen (FMF)	4	7:44:06
36	Matej Tomc (FE), Blaž Zupančič, Marko Rus (Šk. klas. gimn. Lj.)	4	8:23:05
37	Matej Logar, Miha Rot, Gašper Domen Romih (FMF)	4	9:55:37
38	Jure Grabnar, Klemen Gantar, Matej Vehar (FRI)	4	10:25:02
39	Simon Cof, Rok Hudobivnik, Žiga Kern (FRI)	4	13:03:14

* Opomba: naloge z najslabšega od prvih treh krogov se ne štejejo, naloge z zadnjega kroga pa se štejejo dvojno. Enako je tudi pri času, le da se čas zadnjega kroga ne šteje dvojno.

(*nadaljevanje na naslednji strani*)

	Ekipa	Št. rešenih nalog*	Čas
40	Miha Štravs, Andraž Jelenc, Sandi Režonja (FMF + FRI)	3	1:02:17
41	Blaž Soboč an, Miha Garafolj, Matic Zupančič (FMF)	3	3:46:25
42	Gregor Cimerman, Blaž Divjak, Gašper Kojek (FRI)	3	4:31:19
43	Denis Selčan, Manja Tement, Mojca Orgulan (FERI)	3	5:26:19
44	Kristjan Sešek, Klemen Kozjek, Primož Črnigoj (FRI)	3	7:19:20
45	Karen Frlic, Matevž Fabančič, Gašper Jelovčan (FRI)	3	8:14:38
46	Domen Balantič, Andrej Rus, Špela Čopi (FRI + FMF)	3	8:46:13
47	Simon Weiss, Marcel Čampa (FMF), Roman Komac (FRI)	3	9:14:38
48	Jure Taslak (FRI + FMF), Aleš Zavec (Filoz. f.), Mihael Švigelj (FRI)	3	9:51:41
49	Matej Kramberger, Luka Urbanc, Boštjan Budna (FERI)	3	10:15:30
50	Matevž Špacapan, Luka Stopar, Amon Stopinšek (FRI)	2	1:59:38
51	Peter Bernad, Teja Kac, Jan Fekonja (FNM)	2	3:18:23
52	Matej Slemenik, Vid Mahovič (FRI + F. za upravo), Maša Vinter (FMF)	2	4:34:17
53	Luka Lajovic, Jakob Turk, Anže Štular (FMF)	2	7:52:46
54	Špela Zakrajšek, Delfina Bariša (FRI + FMF), Maja Stojanova (FRI)	2	10:28:12
55	Goran Tubić, Tilen Jesenko, Gašper Moderc (FAMNIT)	1	0:38:33
56	Robert Barachini (FRI)	1	1:08:56
57	Klemen Turšič, Grega Mežič, Uroš Prosenik (FRI)	1	3:34:00
58	Alen Verk, Andrej Marsel, Jure Žerak (FERI)	1	5:12:05
59	Tristan Višnar, Miha Vreš, Kevin Šuc (FERI)	0	0:00:00
	Žiga Podgrajšek, Andrej Kostič, Grega Premuša (FERI)	0	0:00:00
	Urban Rajter, Nikola Klipa, Tine Tetičkovič (FNM))	0	0:00:00
	Larisa Carli (FMF), Uroš Vaupotič, Blaž Milar (FE + FRI)	0	0:00:00

* Opomba: naloge z najslabšega od prvih treh krogov se ne štejejo, naloge z zadnjega kroga pa se štejejo dvojno. Enako je tudi pri času, le da se čas zadnjega kroga ne šteje dvojno.

Na srednjeevropskem tekmovanju so nastopile ekipe 1, 3 in 4 kot predstavnice Univerze v Ljubljani, 9 kot predstavnica Univerze v Mariboru in malo spremenjena ekipa 5 kot predstavnica Univerze na Primorskem. V konkurenci 67 ekip z 28 univerz iz 6 držav so slovenske ekipe dosegle naslednje rezultate:

Mesto	Ekipa	Št. rešenih nalog	Čas
26	Žiga Gosar, Maks Kolman, Jure Slak	4	10:52
30	Marko Palangetić, Daniel Silađi, Anton Uramer	4	14:28
32	Sven Cerk, Miha Eleršič, Martin Šušterič	3	5:07
37	Vid Kocijan, Jasna Urbančič, Patrk Zajec	3	7:52
52	Dominik Korošec, Žan Skamljič, Domen Vidovič	2	6:57

Na srednjeevropskem tekmovanju je bilo 12 nalog, od tega jih je zmagovalna ekipa rešila deset.

ANKETA

Tekmovalcem vseh treh skupin smo na tekmovanju skupaj z nalogami razdelili tudi naslednjo anketo. Rezultati ankete so predstavljeni na str. 137–144.

Letnik: ☐ 8. r. OŠ ☐ 9. r. OŠ ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Kako si izvedel(a) za tekmovanje?
☐ od mentorja ☐ na spletni strani (kateri? _____)
☐ od prijatelja/sošolca ☐ drugače (kako? _____)
Kolikokrat si se že udeležil(a) kakšnega tekmovanja iz računalništva pred tem tekmovanjem? _____
Katerega leta si se udeležil(a) prvega tekmovanja iz računalništva? _____
Najboljša dosedanja uvrstitev na tekmovanjih iz računalništva (kje in kdaj)? _____

Koliko časa že programiraš? _____
Kje si se naučil(a)? ☐ sam(a) ☐ v šoli pri pouku ☐ na krožkih ☐ na tečajih
☐ poletna šola ☐ drugje: _____

Za programske jezike, ki jih obvladaš, napiši (začni s tistimi, ki jih obvladaš najbolje):

Jezik: _____
Koliko programov si že napisal(a) v tem jeziku: ☐ do 10 ☐ od 11 do 50 ☐ nad 50
Dolžina najdaljšega programa v tem jeziku:
☐ do 20 vrstic ☐ od 21 do 100 vrstic ☐ nad 100
[Gornje rubrike za opis izkušenj v posameznem programskem jeziku so se nato še dvakrat ponovile, tako da lahko reševalec opiše do tri jezike.]

Ali si programiral(a) še v katerem programskem jeziku poleg zgoraj navedenih? V katerih?

Kako vpliva tvoje znanje matematike na programiranje in učenje računalništva?
☐ zadošča mojim potrebam
☐ občutim pomanjkljivosti, a se znajdem
☐ je preskromno, da bi koristilo

Kako vpliva tvoje znanje angleščine na programiranje in učenje računalništva?
☐ zadošča mojim potrebam
☐ občutim pomanjkljivosti, a se znajdem
☐ je preskromno, da bi koristilo

Ali bi znal(a) v programu uporabiti naslednje podatkovne strukture:

Drevo	☐ da	☐ ne
Hash tabela (razpršena / asociativna tabela)	☐ da	☐ ne
S kazalci povezan seznam (linked list)	☐ da	☐ ne
Sklad (stack)	☐ da	☐ ne
Vrsta (queue)	☐ da	☐ ne

Ali bi znal(a) v programu uporabiti naslednje algoritme:

Evklidov algoritem (za največji skupni delitelj)	☐ da	☐ ne
Eratostenovo rešeto (za iskanje praštevil)	☐ da	☐ ne
Poznaš formulo za vektorski produkt	☐ da	☐ ne
Rekurzivni sestop	☐ da	☐ ne
Iskanje v širino (po grafu)	☐ da	☐ ne
Dinamično programiranje	☐ da	☐ ne
[če misliš, da to pomeni uporabo new, GetMem, malloc ipd., potem obkroži „ne“]		
Katerega od algoritmov za urejanje	☐ da	☐ ne

Katere(ga)? ☐ bubble sort (urejanje z mehurčki)
☐ insertion sort (urejanje z vstavljanjem)
☐ selection sort (urejanje z izbiranjem)
☐ quicksort
☐ kakšnega drugega: _____

Ali poznaš zapis z velikim O za časovno zahtevnost algoritmov?
[npr. $O(n^2)$, $O(n \log n)$ ipd.] ☐ da ☐ ne

[Le pri 1. in 2. skupini.] V besedilu nalog trenutno objavljamo deklaracije tipov in podprogramov v pascalu, C/C++, C#, pythonu in javi.
— Ali razumeš kakšnega od teh jezikov dovolj dobro, da razumeš te deklaracije v besedilu naših nalog? ☐ da ☐ ne
— So ti prišle deklaracije v pythonu kaj prav? ☐ da ☐ ne
— Ali bi raje videl(a), da bi objavljali deklaracije (tudi) v kakšnem drugem programskem jeziku? Če da, v katerem? _____

V rešitvah nalog trenutno objavljamo izvorno kodo v C-ju.
— Ali razumeš C dovolj dobro, da si lahko kaj pomagaš z izvorno kodo v naših rešitvah? ☐ da ☐ ne
— Ali bi raje videl(a), da bi izvorno kodo rešitev pisali v kakšnem drugem jeziku? Če da, v katerem? _____

[Le pri 1. in 2. skupini.] Kakšno je tvoje mnenje o sistemu za oddajanje odgovorov prek računalnika? _____

_____

[Le pri 3. skupini.] Letos v tretji skupini podpiramo reševanje nalog v pascalu, C, C++, C#, javi in VB.NET. Bi rad uporabljal kakšen drug programski jezik? Če da, katerega? _

_____

Katere od naslednjih jezikovnih konstruktov in programerskih prijemov znaš uporabljati?	ne poznam	da, slabo	da, dobro
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz standardnega vhoda ali pa ju zapisati na standardni izhod?	☐	☐	☐
Ali bi znal(a) prebrati kakšno celo število in kakšen niz iz datoteke ali pa ju zapisati v datoteko?	☐	☐	☐
Tabele (**array**):			
— enodimenzionalne	☐	☐	☐
— dvodimenzionalne	☐	☐	☐
— večdimenzionalne	☐	☐	☐
Znaš napisati svoj podprogram (**procedure**, **function**)	☐	☐	☐

Poznaš rekurzijo	☐	☐	☐
Kazalce, dinamično alokacijo pomnilnika (New/Dispose, GetMem/FreeMem, malloc/free, **new**/**delete**, ...)	☐	☐	☐
Zanka **for**	☐	☐	☐
Zanka **while**	☐	☐	☐
Gnezdenje zank (ena zanka znotraj druge)	☐	☐	☐
Naštevni tipi (*enumerated types* — **type** ImeTipa = (Ena, Dve, Tri) v pascalu, **typedef enum** v C/C++)	☐	☐	☐
Strukture (**record** v pascalu, **struct**/**class** v C/C++)	☐	☐	☐
and, **or**, **xor**, **not** kot aritmetični operatorji (nad biti celoštevilskih operandov namesto nad logičnimi vrednostmi tipa boolean) (v C/C++/C#/javi: &, |, ^, ~)	☐	☐	☐
Operatorja **shl** in **shr** (v C/C++/C#/javi: <<, >>)	☐	☐	☐
Znaš uporabiti kakšnega od naslednjih razredov iz standardnih knjižnic:			
hash_map, hash_set, unordered_map, unordered_set (v C++), Hashtable, HashSet (v javi/C#), Dictionary (v C#), dict, set (v pythonu)	☐	☐	☐
map, set (v C++), TreeMap, TreeSet (v javi), SortedDictionary (v C#)	☐	☐	☐
priority_queue (v C++), PriorityQueue (v javi), heapq (v pythonu)	☐	☐	☐

[Naslednja skupina vprašanj se je ponovila za vsako nalogo po enkrat.]
Zahtevnost naloge: ☐ prelahka ☐ lahka ☐ primerna ☐ težka ☐ pretežka ☐ ne vem
Naloga je (ali: bi) vzela preveč časa: ☐ da ☐ ne ☐ ne vem
Mnenje o besedilu naloge:
— dolžina besedila: ☐ prekratko ☐ primerno ☐ predolgo
— razumljivost besedila: ☐ razumljivo ☐ težko razumljivo ☐ nerazumljivo
Naloga je bila: ☐ zanimiva ☐ dolgočasna ☐ že znana ☐ povprečna
Si jo rešil(a)?
- ☐ nisem rešil(a), ker mi je zmanjkalo časa za reševanje
- ☐ nisem rešil(a), ker mi je zmanjkalo volje za reševanje
- ☐ nisem rešil(a), ker mi je zmanjkalo znanja za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo časa za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo volje za reševanje
- ☐ rešil(a) sem jo le delno, ker mi je zmanjkalo znanja za reševanje
- ☐ rešil(a) sem celo

Ostali komentarji o tej nalogi: _____

_____

Katera naloga ti je bila najbolj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

_____

Katera naloga ti je bila najmanj všeč? ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
Zakaj? _____

_____

Na letošnjem tekmovanju ste imeli tri ure / pet ur časa za pet nalog.
Bi imel(a) raje: ☐ več časa ☐ manj časa ☐ časa je bilo ravno prav
Bi imel(a) raje: ☐ več nalog ☐ manj nalog ☐ nalog je bilo ravno prav

Kakršne koli druge pripombe in predlogi. Kaj bi spremenil(a), popravil(a), odpravil(a), ipd., da bi postalo tekmovanje zanimivejše in bolj privlačno? _____

_____

Kaj ti je bilo pri tekmovanju všeč? _____

_____

Kaj te je najbolj motilo? _____

_____

Če imaš kaj vrstnikov, ki se tudi zanimajo za programiranje, pa se tega tekmovanja niso udeležili, kaj bi bilo po tvojem mnenju treba spremeniti, da bi jih prepričali k udeležbi?

_____

_____

Poleg tekmovanja bi radi tudi v preostalem delu leta organizirali razne aktivnosti, ki bi vas zanimale, spodbujale in usmerjale pri odkrivanju računalništva. Prosimo, da nam pomagate izbrati aktivnosti, ki vas zanimajo in bi se jih zelo verjetno udeležili.

Udeležil bi se oz. z veseljem bi spremljal:

- ☐ izlet v kak raziskovalni laboratorij v Evropi (po možnosti za dva dni)
- ☐ poletna šola računalništva (1 teden na IJS, spanje v dijaškem domu)
- ☐ poletna praksa na IJS
- ☐ predstavitve novih tehnologij (.NET, mobilni portali, programiranje „vgrajenih računalnikov“, strojno učenje, itd.) (1× mesečno)
- ☐ predavanja o algoritmih in drugih temah, ki pridejo prav na tekmovanju (1× mesečno)
- ☐ reševanje tekmovalnih nalog (naloge se rešuje doma in bi bile delno povezane s temo, predstavljeno na predavanju; rešitve se preveri na strežniku) (1× mesečno)
- ☐ tvoji predlogi: _____

  _____

Vesel(a) bi bil pomoči pri:

- ☐ iskanju štipendije
- ☐ iskanju podjetij, ki dijakom ponujajo njim prilagojene poletne prakse in druge projekte, kjer se ob mentorstvu lahko veliko naučijo.

Ali si pri izpolnjevanju ankete prišel/la do sem? ☐ da ☐ ne

Hvala za sodelovanje in lep pozdrav! Tekmovalna komisija

REZULTATI ANKETE

Anketo je izpolnilo 83 tekmovalcev prve skupine, 16 tekmovalcev druge skupine in 10 tekmovalcev tretje skupine. Vprašanja so bila pri letošnji anketi enaka kot lani.

Mnenje tekmovalcev o nalogah

Tekmovalce smo spraševali: kako zahtevna se jim zdi posamezna naloga; ali se jim zdi, da jim vzame preveč časa; ali je besedilo primerno dolgo in razumljivo; ali se jim zdi naloga zanimiva; ali so jo rešili (oz. zakaj ne); in katera naloga jim je bila najbolj/najmanj všeč.

Rezultate vprašanj o zahtevnosti nalog kažejo grafi na str. 138. Tam so tudi podatki o povprečnem številu točk, doseženem pri posamezni nalogi, tako da lahko primerjamo mnenje tekmovalcev o zahtevnosti naloge in to, kako dobro so jo zares reševali.

V povprečju so se zdele tekmovalcem v vseh skupinah naloge še kar težke, vendar so številke podobne kot v prejšnjih letih. Če pri vsaki nalogi pogledamo povprečje mnenj o zahtevnosti te naloge (1 = prelahka, 3 = primerna, 5 = pretežka) in vzamemo povprečje tega po vseh petih nalogah, dobimo: 3,31 v prvi skupini (v prejšnjih letih 3,41, 3,40, 3,28, 3,39, 3,56), 3,65 v drugi skupini (prejšnja leta 3,33, 3,44, 3,35, 3,50, 3,39) in 3,43 v tretji skupini (prejšnja leta 3,61, 3,19, 3,40, 3,21, 3,57).

Med tem, kako težka se je naloga zdela tekmovalcem, in tem, kako dobro so jo zares reševali (npr. merjeno s povprečnim številom točk pri tej nalogi), je ponavadi (šibka) negativna korelacija; letos je bila še kar močna ($R^2 = 0{,}39$; v prejšnjih letih 0,56, 0,14, 0,52, 0,21, 0,11, pred tem okoli 0,4).

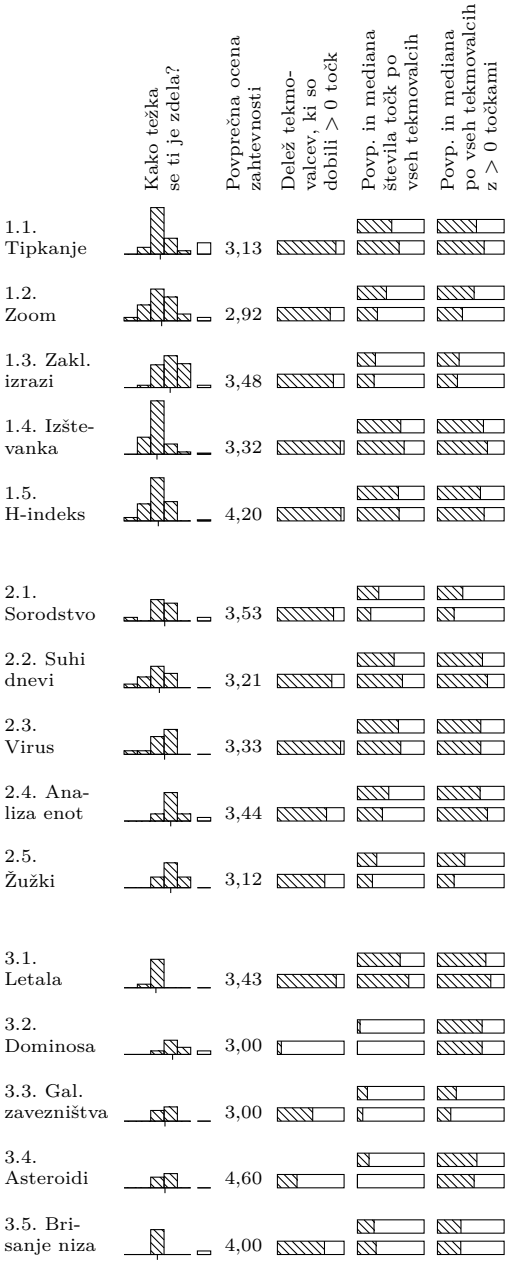
Težke so se tekmovalcem zdele predvsem naloge 1.3 (zaklepajski izrazi), 2.4 (analiza enot) in 2.5 (za žužke gre). V tretji skupini se jim je zdela daleč najtežja naloga 3.2 (dominosa), kar je rahlo presenetljivo, saj za reševanje te naloge ne potrebujemo kakšne posebne zvitosti, pač pa le grobo silo (ta naloga je takorekoč šolski primer rekurzivnega sestopa).

Kot najlažjo so tekmovalci v prvi skupini ocenili nalogo 1.4 (izštevanka), v drugi nalogo 2.2 (suhi dnevi), v tretji skupini pa 3.1 (letala). Pri nalogah 1.4 in 1.2 (zoom) je bilo celo nekaj pripomb, da sta prelahki.

Rezultate ostalih vprašanj o nalogah pa kažejo grafi na str. 139. Nad razumljivostjo besedil ni veliko pripomb (še malo manj kot prejšnja leta); kot težje razumljiva izstopa v prvi skupini naloga 1.2 (zoom), v drugi pa 2.4 (virus) in 2.5 (za žužke gre). Slednji dve nalogi sta malo bolj nestandardnega tipa in take je pogosto težko razložiti tako, da bo opis naloge preprost in razumljiv, vendar hkrati tudi dovolj natančen in nedvoumen. V tretji skupini je bilo letos zelo malo pripomb glede razumljivosti besedil.

Tudi z dolžino besedil so tekmovalci pri skoraj vseh nalogah zadovoljni, približno enako kot v prejšnjih letih oz. celo še malo bolj. Edina naloga, pri kateri je bilo veliko pripomb, da je predolga, je bila 3.3 (galaktična zavezništva). Pri večini ostalih nalog se je besedilo več ljudem zdelo prekratko kot predolgo, še posebej v prvi skupini.

Naloge se jim večinoma zdijo zanimive; ocene so pri tem vprašanju podobne kot prejšnja leta, vendar je letos malo več pripomb, češ da je neka naloga že znana. Razlike v oceni zanimivosti med nalogami so večinoma majhne, kot manj zanimivi



Mnenje tekmovalcev o zahtevnosti nalog in število doseženih točk

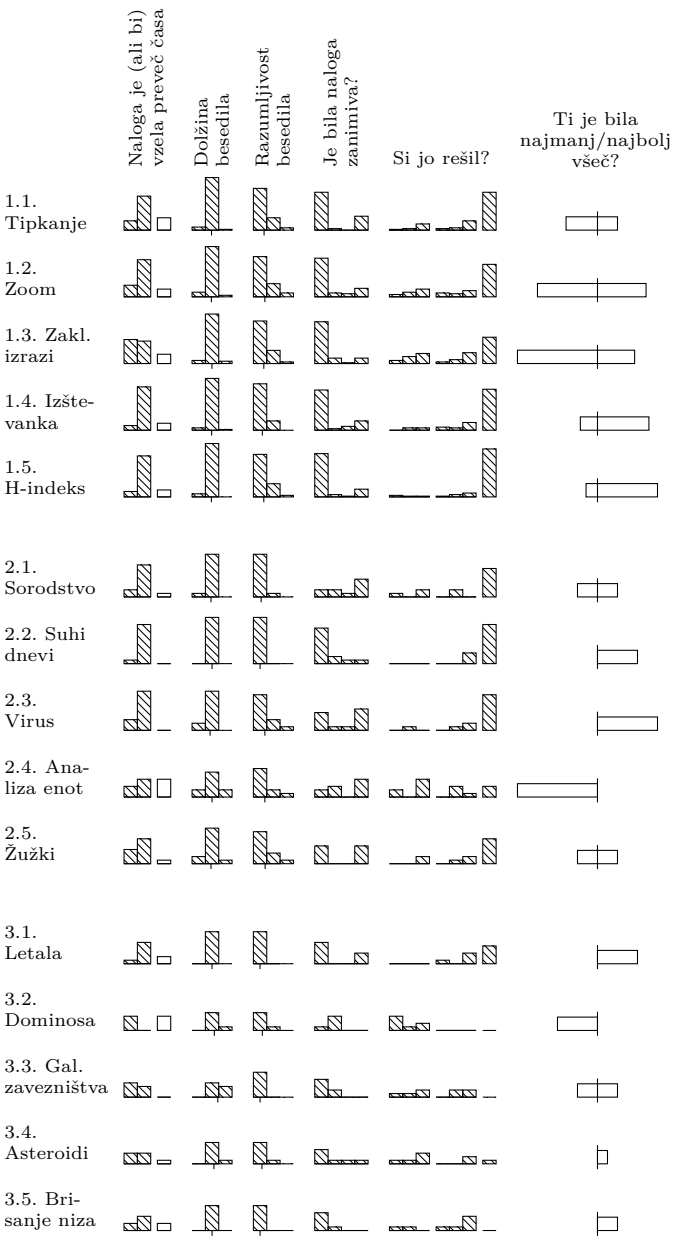
Pomen stolpcev v vsaki vrstici:

Na levi je skupina šestih stolpcev, ki kažejo, kako so tekmovalci v anketi odgovarjali na vprašanje o zahtevnosti naloge. Stolpci po vrsti pomenijo odgovore „prelahka“, „lahka“, „primerna“, „težka“, „pretežka“ in „ne vem“. Višina stolpca pove, koliko tekmovalcev je izrazilo takšno mnenje o zahtevnosti naloge. Desno od teh stolpcev je povprečna ocena zahtevnosti (1 = prelahka, 3 = primerna, 5 = pretežka). Povprečno oceno kaže tudi črtica pod to skupino stolpcev.

Sledi stolpec, ki pokaže, kolikšen delež tekmovalcev je pri tej nalogi dobil več kot 0 točk. Naslednji par stolpcev pokaže povprečje (zgornji stolpec) in mediano (spodnji stolpec) števila točk pri vsej nalogi. Zadnji par stolpcev pa kaže povprečje in mediano števila točk, gledano le pri tistih tekmovalcih, ki so dobili pri tisti nalogi več kot nič točk.

Mnenje tekmovalcev o nalogah

Višina stolpcev pove, koliko tekmovalcev je dalo določen odgovor na neko vprašanje.



Stolpci se od leve proti desni nanašajo na naslednja vprašanja in možne odgovore:

Naloga je (ali: bi) vzela preveč časa:

- da
- ne
- ne vem

Dolžina besedila:

- prekratko
- primerno
- predolgo

Razumljivost besedila:

- razumljivo
- težko razumljivo
- nerazumljivo

Naloga je bila:

- zanimiva
- dolgočasna
- že znana
- povprečna

Si jo rešil?

- nisem, zmanjkalo časa
- nisem, zmanjkalo volje
- nisem, zmanjkalo znanja
- delno, zmanjkalo časa
- delno, zmanjkalo volje
- delno, zmanjkalo znanja
- rešil sem celo

Katera naloga ti je bila najmanj všeč? Katera ti je bila najbolj všeč?

izstopata predvsem nalogi 2.4 (analiza enot) in 3.2 (dominosa), kot bolj zanimive pa 2.2 (suhi dnevi) in 3.5 (brisanje niza).

Pripomb, da bi naloga vzela preveč časa, je bilo malo, podobno kot prejšnja leta, le v tretji skupini malo več. Največ takih pripomb je bilo pri nalogi 1.3 (zaklepajski izrazi), 3.2 (dominosa), 3.3 (galaktična zavezništva) in 3.4 (asteroidi). Najbrž to mnenje izvira iz dejstva, da so te naloge malo težje, saj drugače vsaj 1.3 in 3.2 sami po sebi nista tako zamudni za reševanje.

Pri glasovih o tem, katera naloga je tekmovalcu najbolj in katera najmanj všeč, je bila v prvi skupini najbolj priljubljena naloga 1.5 (H-indeks), najmanj pa 1.3 (zaklepajski izrazi). V drugi skupini kot izrazito nepriljubljena izstopa 2.4 (analiza enot), najbolj všeč pa jim je bila 2.3 (virus). V tretji skupini jim je bila najbolj všeč naloga 3.1 (letala), najmanj pa 3.2 (dominosa).

Programersko znanje, algoritmi in podatkovne strukture

Ko sestavljamo naloge, še posebej tiste za prvo skupino, nas pogosto skrbi, če tekmovalci poznajo ta ali oni jezikovni konstrukt, programerski prijem, algoritem ali podatkovno strukturo. Zato jih v anketah zadnjih nekaj let sprašujemo, če te reči poznajo in bi jih znali uporabiti v svojih programih.

	Prva skupina	Druga skupina	Tretja skupina
priority_queue v C++ ipd.	4 %	23 %	30 %
map v C++ ipd.	6 %	23 %	40 %
unordered_map v C++ ipd.	9 %	23 %	50 %
zamikanje s **shl**, **shr**	18 %	17 %	60 %
operatorji na bitih	43 %	54 %	70 %
strukture	41 %	54 %	90 %
naštevni tipi	26 %	54 %	50 %
gnezdenje zank	90 %	100 %	90 %
zanka **while**	95 %	93 %	100 %
zanka **for**	96 %	93 %	100 %
kazalci	13 %	29 %	30 %
rekurzija	35 %	50 %	80 %
podprogrami	70 %	86 %	100 %
več-d tabele (**array**)	47 %	50 %	100 %
2-d tabele (**array**)	62 %	71 %	100 %
1-d tabele (**array**)	78 %	86 %	100 %
delo z datotekami	58 %	93 %	100 %
std. vhod/izhod	89 %	100 %	100 %

Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo in bi znali uporabiti določen konstrukt ali prijem: „da, dobro“ (poševne črte), „da, slabo“ (vodoravne črte) ali „ne“ (nešrafirani del stolpca). Ob vsakem stolpcu je še delež odgovorov „da, dobro“ v odstotkih.

Rezultati pri vprašanjih o programerskem znanju so podobni tistim iz prejšnjih let. V tretji skupini (pravijo, da) znajo malo manj kot lani, vendar je zaradi majhnega števila anket v tretji skupini vprašljivo, koliko se lahko zanesemo na te podatke. Stvari, ki jih tekmovalci poznajo slabše, so na splošno približno iste kot prejšnja leta: rekurzija, kazalci, naštevni tipi in operatorji na bitih, v prvi in drugi skupini tudi strukture.

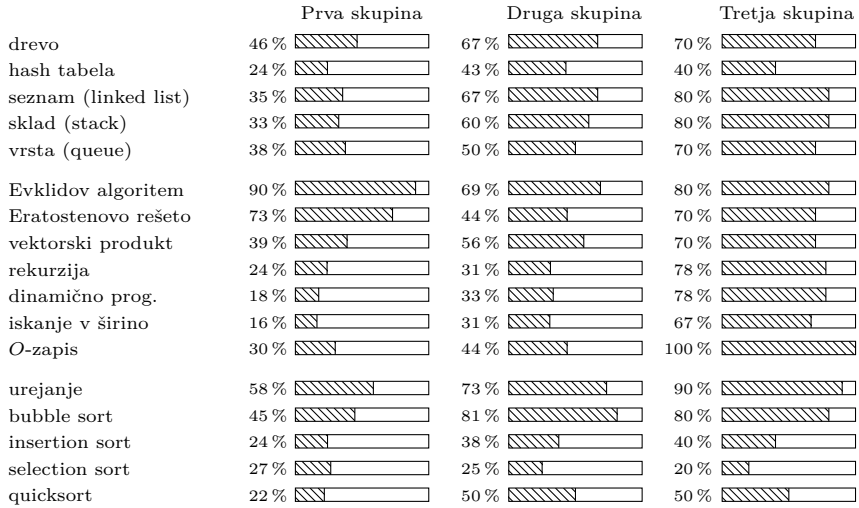


Tabela kaže, kako so tekmovalci odgovarjali na vprašanje, ali poznajo nekatere algoritme in podatkovne strukture. Ob vsakem stolpcu je še odstotek pritrdilnih odgovorov.

Uporaba programskih jezikov

Največ tekmovalcev tudi letos uporablja C/C++ (podobno kot zadnja leta je čisti C razmeroma redek), je pa njegova prednost pred drugimi jeziki manjša. V prvi skupini sta letos najpogostejša jezika python in C++, ki sta približno izenačena, malo manj ljudi je uporabljalo javo, še manj pa C#. V drugi skupini je najpogostejši python, kar je novost, ker je bil prejšnja leta python praviloma v drugi skupini uporabljan veliko redkeje kot v prvi. Z nekaj zaostanka mu sledi C++, še malo za njim pa sta java in C#. Razmerja med temi štirimi jeziki sicer iz leta v leto po malem nihajo brez kakšnega očitnega trenda. V tretji skupini je C++ najpogostejši, sledi pa mu java. Od podprtih jezikov letos v tretji skupini nihče ni uporabljal pascala (tako je bilo tudi lani), pa tudi basica ne.

Podobno kot prejšnja leta se je tudi letos pojavilo nekaj tekmovalcev (in tekmovalk), ki oddajajo le rešitve v psevdokodi ali pa celo naravnem jeziku, tudi tam, kjer naloga sicer zahteva izvorno kodo v kakšnem konkretnem programskem jeziku. Iz tega bi človek mogoče sklepal, da bi bilo dobro dati več nalog tipa „opiši postopek“ (namesto „napiši podprogram“), vendar se v praksi običajno izkaže, da so takšne naloge med tekmovalci precej manj priljubljene in da si večinoma ne predstavljajo preveč dobro, kako bi opisali postopek (pogosto v resnici oddajo dolgovezne opise izvorne kode v stilu „nato bi s stavkom **if** preveril, ali je spremenljivka `x` večja od spremenljivke `y`“). Podobno kot lani smo tudi letos pri nalogah tipa „opiši postopek“ pripisali „ali napiši podprogram (kar ti je lažje)“.

Podobno kot v prejšnjih letih je v anketi še kar nekaj tekmovalcev napisalo, da dobro poznajo tudi PHP in/ali javascript, vendar na tekmovanju letos PHPja ni uporabljal nihče, javascript pa le en tekmovalec.

Podrobno število tekmovalcev, ki so uporabljali posamezne jezike, kaže tabela na str. 142. Glede štetja C in C++ v tej tabeli je treba pripomniti, da je razlika med

	Leto in skupina																	
	2016			2015			2014			2013			2012			2011		
Jezik	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
pascal	$\frac{1}{3}$	3		5	2		$2\frac{1}{2}$	2	1	1		1	6	1	4	3	4	3
C	$4\frac{1}{3}$	1	2	3		1	$3\frac{1}{2}$	6		2	7		7	2	1	7	2	
C++	28	8	9	27	9	$9\frac{1}{2}$	19	$4\frac{1}{2}$	$10\frac{1}{2}$	17	$12\frac{1}{2}$	7	26	16	9	$23\frac{1}{2}$	19	8
java	24	6	5	22	6	$3\frac{1}{2}$	23	2	$1\frac{1}{2}$	12	8	1	17	$6\frac{1}{2}$	1	6	5	3
PHP			–	3		–	2		–	1	$\frac{1}{2}$	–		1	–	$\frac{1}{2}$		–
basic						1		1	–	1		–			–			–
C#	12	5	1	16	5		12	$1\frac{1}{2}$	2	18	$\frac{1}{2}$		17	1	3	4	2	3
python	$29\frac{1}{3}$	12		26	1	–	16	6	–	16	8	–	25	5	–	20	6	–
NewtonScript			–			–			–		$\frac{1}{2}$	–		$\frac{1}{2}$	–			–
javascript	1		–	1		–	1		–			–			–			–
batch			–			–	1		–			–			–			–
psevdokoda	5		–	6	1	–	10		–	6		–	3		–	6		–
nič	2	3		4	1		4	2		2			2			1	1	

Število tekmovalcev, ki so uporabljali posamezni programski jezik.
Nekateri uporabljajo po več različnih jezikov (pri različnih nalogah) in se štejejo delno k vsakemu jeziku. (V letu 2016 je bil tak primer en sam, in sicer tekmovalec, ki je uporabljal pascal, C in python.) „Nič“ pomeni, da tekmovalec ni napisal nič izvorne kode. Znak „–“ označuje jezike, ki se jih tisto leto v tretji skupini ni dalo uporabljati. Psevdokoda šteje tekmovalce, ki so pisali le psevdokodo, tudi pri nalogah tipa „napiši (pod)program“.

njima majhna in včasih pri kakšnem krajšem kosu izvorne kode že težko rečemo, za katerega od obeh jezikov gre. Je pa po drugi strani videti, da se raba stvari, po katerih se C++ loči od C-ja, sčasoma povečuje; vse več tekmovalcev na primer uporablja `string` namesto **`char *`** in tip `vector` namesto tradicionalnih tabel (*arrays*). Tako kot lani smo tudi letos pri enem tekmovalcu opazili rešitve v C++11/14 (med drugim je uporabljal **`auto`** v njegovem novem pomenu).

Pri pythonu zdaj že velika večina ljudi uporablja python 3 in ne python 2; je pa res, da je pri tako preprostih programih, s kakršnimi se srečujemo na našem tekmovanju, razlika večinoma le v tem, ali `print` uporabljajo kot stavek ali kot funkcijo.

V besedilu nalog za 1. in 2. skupino objavljamo deklaracije tipov, spremenljivk, podprogramov ipd. v pascalu, C/C++, C#, pythonu in javi. Delež tekmovalcev, ki pravijo, da deklaracije razumejo, je letos v prvi skupini višji kot običajno (75/79), tudi v drugi je še kar visok (14/16). Nenavadno je, da so pri vprašanju, ali bi želeli deklaracije še v kakšnem jeziku, nekateri tekmovalci navedli jezike, v katerih deklaracije že imamo, na primer javo, C++ in C#; od originalnih predlogov je bil najpogostejši javascript. Tako se človek vpraša, koliko se smemo na odgovore pri teh vprašanjih sploh zanašati. V vsakem primeru pa se poskušamo zadnja leta v besedilih nalog izogibati deklaracijam v konkretnih programskih jezikih in jih zapisati bolj na splošno, na primer „napiši funkcijo `foo(x, y)`“ namesto „napiši funkcijo **`bool`** `foo(`**`int`** `x, `**`int`** `y)`“.

V rešitvah nalog zadnja leta objavljamo izvorno kodo le v C-ju; tekmovalce smo v anketi vprašali, če razumejo C dovolj, da si lahko kaj pomagajo s to izvorno kodo, in če bi radi videli izvorno kodo rešitev še v kakšnem drugem jeziku. Večina je s C-jem sicer zadovoljna (42/77 v prvi skupini, 14/15 v drugi, 9/10 v tretji), pravzaprav so ti deleži še višji kot lani, vendar je letos v prvi skupini še vedno precej takih,

ki pravijo, da izvorne kode v rešitvah ne razumejo. Zanimivo vprašanje je, ali bi s kakšnim drugim jezikom dosegli večji delež tekmovalcev (koliko tekmovalcev ne bi razumelo rešitev v javi? ali v pythonu?). Med jeziki, ki bi jih radi videli namesto (ali poleg) C-ja, jih največ omenja python in javo, malo manj je glasov za C++, zlasti v prvi skupini pa jih nekaj želi C#.

Letnik

Po pričakovanjih so tekmovalci zahtevnejših skupin v povprečju v višjih letnikih kot tisti iz lažjih skupin. Razmerja so podobna kot prejšnja leta, v povprečju pa so se tekmovalci letos spet rahlo pomladili. Letos je nastopil en osnovnošolec, in sicer v tretji skupini.

	Št. tekmovalcev po letnikih					Povprečni
Skupina	OŠ	1	2	3	4	letnik
prva		23	31	29	23	2,5
druga			8	13	17	3,2
tretja	1		3	5	8	3,3

Druga vprašanja

Podobno kot prejšnja leta je velikanska večina tekmovalcev za tekmovanje izvedela prek svojih mentorjev (hvala mentorjem!). V smislu širitve zanimanja za tekmovanje in večanja števila tekmovalcev se zelo dobro obnese šolsko tekmovanje, ki ga izvajamo zadnjih nekaj let, saj se odtlej v tekmovanje vključuje tudi nekaj šol, ki prej na našem državnem tekmovanju niso sodelovale. Nekaj ljudi je za naše tekmovanje slišalo na računalniškem tekmovanju Bober, ki ga tudi organizira društvo ACM Slovenija.

Pri vprašanju, kje so se naučili programirati, sta podobno kot prejšnja leta najpogostejši odgovor, da so se naučili programirati sami ali pa v šoli (ti dve skupini sta letos približno izenačeni); malo manj pa je takih, ki so se naučili programirati na krožkih ali tečajih.

Pri času reševanja in številu nalog je največ takih, ki so s sedanjo ureditvijo zadovoljni. Med tistimi, ki niso, so mnenja precej razdeljena, vendar je podobno kot lani najpogostejša kombinacija „več časa, enako nalog“. V tretji skupini si nekoliko več ljudi kot ponavadi želi manj nalog.

Iz odgovorov na vprašanje, kakšne potekmovalne dejavnosti bi jih zanimale, je težko zaključiti kaj posebej konkretnega.

Z organizacijo tekmovanja je drugače velika večina tekmovalcev zadovoljna in nimajo posebnih pripomb. Od 2009 imajo tekmovalci v prvi in drugi skupini možnost pisati svoje odgovore na računalniku namesto na papir (kar so si prej v anketah že večkrat želeli). Velika večina jih je res oddajala odgovore na računalniku, nekaj pa jih je vseeno reševalo na papir. Pri oddajanju odgovorov na računalniku se stanje izboljšuje, letos skoraj ni bilo težav s shranjevanjem, je pa še vedno nekaj pripomb glede počasnosti sistema.

Podobno kot prejšnja leta si je veliko tekmovalcev tudi želelo, da bi imeli v prvi in drugi skupini na računalnikih prevajalnike in podobna razvojna orodja. Razlog, zakaj se v teh dveh skupinah izogibamo prevajalnikom, je predvsem ta, da hočemo s tem obdržati poudarek tekmovanja na snovanju algoritmov, ne pa toliko na lovljenju drobnih napak; in radi bi tekmovalce tudi spodbudili k temu, da se lotijo vseh

	Kje si izvedel za tekmovanje				Kje si se naučil programirati					Čas reše-vanja			Število nalog			Potekmovalne dejavnosti							
Skupina	od mentorja	na spletni strani	od prijatelja/sošolca	drugače	sam	pri pouku	na krožkih	na tečajih	poletna šola	hočem več časa	hočem manj časa	je že v redu	hočem več nalog	hočem manj nalog	je že v redu	izlet v tuji laboratorij	poletna šola	praksa na IJS	predstavitve tehnologij	predavanja o algoritmih	reševanje nalog	iskanje štipendije	iskanje podjetij
I	74	3	5	3	49	47	16	10	5	19	8	48	4	10	60	26	32	31	35	38	24	27	41
II	15	0	1	0	9	9	1	2	1	2	4	8	0	3	10	4	2	4	4	2	2	4	6
III	9	0	1	0	8	3	5	0	2	1	3	4	0	5	3	2	4	4	1	6	5	3	7

nalog, ne pa da se zakopljejo v eno ali dve najlažji in potem večino časa porabijo za testiranje in odpravljanje napak v svojih rešitvah pri tistih dveh nalogah. Je pa res, da bi pri nekaterih programskih jezikih prišlo prav vsaj kakšno primerno razvojno okolje (IDE), ki človeku pomaga hitreje najti oz. napisati imena razredov in funkcij iz standardne knjižnice ipd.

CVETKE

V tem razdelku je zbranih nekaj zabavnih odlomkov iz rešitev, ki so jih napisali tekmovalci. V oklepajih pred vsakim odlomkom sta skupina in številka naloge.

(1.1) Že res, da smo v navodilih rekli, naj tekmovalci izvorno kodo v svojih odgovorih dobro komentirajo, nekateri pa s tem vseeno malo pretiravajo:

```
x = 0              # x = 0
y = 0              # y = 0
seznam = []        # naredimo prazen seznam
```

(1.1) Nekaj tekmovalcev se je lotilo te naloge na čisto zgrešen način in so poskusili napisati program, ki od uporabnika zahteva, da natipka besede, program pa bi pri tem le štel pritiske na tipke:

```
for (int i = 0; i < stevilo_besed; i++)
{
   preberi vnos_besede[i];
   izpiši vnos_besede[i];
   stevec zabeleži vsak pritisk tipke na tipkovnici;
}
izpiši "Pripadajoci izhod: " << stevec << endl;
```

Še en podoben primer:

```
if (beseda == "enter"): # Preveri, če je vnešen znak „enter“.
   # šteje se, kolikokrat je uporabnik pritisnil backspace
   # če uporabnik pritisne še enkrat enter
   # se v spremenljivko stCrkBesede prišteje še, kolikokrat je uporabnik pritisnil backspace
```

Tale ekstremno lena rešitev pa prevali štetje pritiskov kar na uporabnika:

```
a = input("Vnesi stevilo besed, ki bi jih rad vnesel")
SteviloBackspacou = input("Kolikokrat ste pritisnili Backspace?")
SteviloPritiskovTipk = 1 + i + a + SteviloBackspacou
```

(1.1) Rešitev z nezaupanjem do direktive **using**:

```
using namespace std; // najavim std → bolj optimalno bi bilo, če bi pri ukazih
          // najavljal std::, ampak ker gre za manjši program, ne vpliva precej
```

(1.1) Zanimiva razširitev operatorja sizeof, ki naj bi tukaj vračal dolžino niza:

```
string line = ""; // vnesena beseda
string prevB = ""; // prejšnja beseda
⋮
if (sizeof(line) < sizeof(prevB)) {
```

(1.1) Eden od tekmovalcev je napisal rešitev v javascriptu in jo celo vključil v dokument v HTMLju s preprostim grafičnim uporabniškim vmesnikom:

```
⟨html⟩
⋮
⟨body⟩
  ⟨textarea id="text"⟩⟨/textarea⟩
  ⟨button onclick="submit()"⟩preveri⟨/button⟩
  ⟨script⟩
    function submit() {
      var in = document.getElementById('text').innerHTML;
```

(1.1) Pri brisanju znakov iz nizov pridejo ljudem pogosto na misel razne zanimive sintaktične inovacije; en tak primer imamo tudi letos:

```
y = y − y[len(y) − 1] # Odrežemo zadnji znak v besedilu.
```

(1.1) Rešitev, ki prevali delo na „neko knjižnico“:

```
c = besedilo.match(besedilo[i − 1], besedilo[i]); # uporabim match iz neke knjižnice,
  # funkcija mi vrne chare, ki se v obeh besedah ujemajo;
```

Isti tekmovalec pri 3. nalogi:

```
ns.replace(nasprotna().zvezdica, par.oklepaj); # te funkcije so izmišljene,
                                               # saj se mi svojih ne da pisati
```

(1.1) Tale rešitev v pythonu poskuša prebrati vhodno datoteko tako, da jo kar uvozi kot pythonovski modul:

> Najprej naložimo v program datoteko `besede.txt`
> (**import** besede.txt **as** besede)

(1.2) Pri tej nalogi smo dobili najdaljšo rešitev letos: dolga je kar 854 (nepraznih) vrstic. Temelji na neupravičeni predpostavki, da je vhodni niz vedno dolg tri znake (ker je tako dolg niz v primeru v besedilu naloge), torej je možnih le $4 \cdot 4 \cdot 4 = 64$ različnih vhodnih nizov in rešitev lahko s kupom stavkov **if** preveri, katera od teh 64 možnosti je na vhodu, ter izpiše ustrezen rezultat. Podobne (vendar malo krajše) rešitve je oddalo še nekaj drugih tekmovalcev.

(1.2) Rešitev za ljubitelje šemljenja:

> Najprej pa moramo kostumizirati koordinate znotraj četrtine, za vsako četrtino v prvem delu posebej.

(1.2) Rešitev, ki prevali delo na knjižnico za grafični uporabniški vmesnik:

```
# polje lahko definiramo s funkcijo grid, če imamo število
# manjših polj (pixlov) v večjem polju
# za funkcijo potrebujemo import programskega dodatka tkinter
# za uporabo pojavnega okna in „grid“-anja tega okna.
```

(1.2) Tale tekmovalec je očitno pozabil, da če kvadrat razdelimo na četrtine, so njihove stranice le pol manjše od stranic prvotnega kvadrata, ne štirikrat manjše:

```
double unit = x2 − x1; // unit od x1 do x2
⋮
unit = unit / 4; // vedno je zoomiran kvadrat 4-krat manjši,
                 // zato je tudi enota 4-krat manjša
```

(1.2) Koristne operacije:

```
else if (C == 2) { x2 += 0; y2 += 0; }
```

(1.2) Rešitev z veliko kompliciranja:

> najprej sem želel narediti **void** brez **if** stavkov, s pomočjo mapa → ta bi pomnožil x1 s stoi(map[nekaj][1]). Toda notri bi bilo preveč operacij (stoi, množenje), da bi se stvar po mojem okusu sploh splačala. Zato naredim s 4 **if** stavki. [...] Imel sem tudi dve ideji, eno z 2d tabelami ter drugo brez **for** zanke in s kompliciranim sistemom množenja...

(1.2) Tale tekmovalec nekako ni razumel, da dobi vhodni niz kot parameter, zato se je namesto z njim raje ukvarjal z zaporedjem naključnih premikov:

```
Random r = new Random();
// ni določeno, kolikokrat ponovimo cikel, zato vzamemo naključno od 1–5
rand = r.nextInt(4) + 1;
do {
   ⋮
   // izberemo si naključno četrtino
   int cet = r.nextInt(4);
   ⋮
} while (i < rand);
```

(1.3) Letošnjo nagrado za najglobljo indentacijo dobi naslednja mojstrovina (da prihranimo prostor, smo izpustili komentarje in nekatere dele pogojev v stavkih if):

```
void Dopolni(char s[n]) {
   ⋮
   if (strlen(s) % 2 == 0) {
      if (!(s[strlen(s) − 1] == '(' || ...) {
         ⋮
         if (zv % 2 == 0 && zv >= 0) {
            for (int i = 1; i < strlen(s) − 1; i++) {
               if (s[i] == ')' || s[i] == '>' || s[i] == ']' || s[i] == '}') {
                  for (intj = 0; j <= i; j++) {
                     if ((s[i] == ')' && s[j] == '(') || ...) {
                        ⋮
                     else {
                        for (k = (j + i)/ 2; k > 0; k−−) {
                           if (s[k] == '(' || s[k] == '<' || s[k] == '{' || s[k] == '[') {
                              ⋮
                           else {
                              switch (s[k]) {
                              case '(':
                                 s[k + 1] == ')';
```

(1.3) Rešitev z veliko samozavesti:

```
System.out.println(charArrayToString(oklepaji)); // ko je konec dodajanja,
                                                 //dobljeno izpišemo in dobim 20 točk.
```

(1.3) Ena od najbolj posrečenih tipkarskih napak letos: zanklepaj!

```
if (s[i] == zaklepaj) // če je znak zaklepaj koncaj nastavi pozicijo
                      // na ta zanklepaj in vrni string
```

Lahko si predstavljamo zaklepaj, ki se pojavlja v zanki `:)`

(1.3) Rešitev s sklicevanjem na nadnaravne sile:

```
// Nato uporabi čarovnijo in naj bi pravilno dopolnil oklepajski izraz.
```

(1.3) Pri tej nalogi so mnoge rešitve najprej preverile, če je niz lihe dolžine, saj iz takega niza gotovo ni mogoče dobiti veljavnega oklepajskega izraza. Naslednji tekmovalec se je tega lotil na domiseln način:

```
double a;
int b;
a = s.length() / 2;
b = s.length() / 2;
if (a = b) {
```

Z implementacijo sta tu sicer dve težavi: pri izrazu `s.length() / 2` se izvede celoštevilsko deljenje ne glede na to, ali rezultat potem priredimo spremenljivki tipa **double** ali tipa **int**; in v stavku **if** se izvede prireditev namesto primerjave.

(1.3) Rešitev z nezaupanjem do operatorja „<“:

```
s.Replace('*', kotn < 0 && kotn != 0 ? '<' : '*');
```

(1.3) Pazljiva izbira oklepaja:

> postavil bi se v simetralo in se pomikal v levo in v desno ter zvezdice zamenjal s oklepajem, ki je zrcalen tistemu, ki je njemu zrcalen.

(1.3) Rešitev za ljubitelje nepotrebnih optimizacij: deklariral je `map<string, string>` (kar je v praksi ponavadi rdeče-črno drevo ali kaj podobnega), s katerim lahko preslika zaklepaj v pripadajoči oklepaj:

> Ko ga najde, pogleda v slovar (`map`), s čim mora nadomestiti zvezdico. Tako prihranim $O(4)$ časa reševanje, saj namesto 4 **if** stavkov dobim takojšen odgovor.

(1.3) Letošnji prispevek na področju neobičajnih sintaktičnih inovacij:

```
if (okrogli == 0; oglati == 0; zaviti == 0; kotni == 0;) {
  System.out.println("Problem je nerešljiv");
```

Videti je torej, da je podpičje uporabil kot operator `&&`, kar je še toliko bolj čudno zato, ker drugod v programu operator `&&` čisto normalno uporablja...

(1.3) Tale tekmovalec si je očitno navodilo, naj podprogram „izpiše, da je problem nerešljiv“, razlagal zelo dobesedno:

```
if (len(s) % 2 !=0) or abc != 0:
  print(", da je problem nerešljiv.")
```

(1.4) Nagrado za najdaljša imena podprogramov dobi:

```
if (forceTheChildToLiftTheirLeg(anotherMethodToFilterThoseChilds(getNextChild())))
```

(1.4) Še ena lepa sintaktična operacija: operator „ni manjši“, ki seveda pomeni isto kot bolj znani „večji ali enak“.

```
while (calcMest !< fStotrok)
  calcMest −= fStotrok;
```

(1.4) Slikoviti komentarji:

```
n[mesto]−−;                  // otroku zbijemo eno življenje
if (n[mesto] == 0) {         // če nima več življenj
  System.out.println(mesto); // ga natisnemo
  break;                     // pač je padel in se zlomil
```

(1.4) Tale je imel poleg spremenljivke `b` tudi spremenljivko `(b)`, je pa iz kode težko razbrati, kaj je mislil s tem:

```
int b = 1;
⋮
if (b != (b)) {
  int (b) = 1
```

(1.4) Koristen pogoj:

```
if (y == y − 1)
{
  izbrani = y;
  break;
}
```

(1.4) Komentar na koncu ene od rešitev:

```
# otroci so morali biti precej pametni, da so se domislili te izštevanke -.-
# lažje bi bilo, če bi se držali z an ban
```

(1.4) Tale rešitev ni zadovoljna s tem, da so otroci oštevilčeni od 0 do $n-1$, in jim še dodatno priredi številke od 0 do $n-1$:

```
int P[100];
⋮
while (a != n)
{
  P[a] = a; // otroci se oštevilčijo
  a++;
}
```

Kasneje to tabelo tudi res uporablja, vendar le bere iz nje, nikoli pa je ne spreminja.

(1.4) Tale rešitev se res potrudi, da bi imel `iStevec` vrednost 0:

```
iStevec = 0;
if(iStevec2 > iStevec)
  iStevec = 0;
iStevec = iStevec;
```

(1.4) Rešitev za ljubitelje psihoaktivnih substanc:

```
# ideja je, da preštevam otroke in ugotovim, kdaj so zadeti
list otroci[];
list zadeti_otroci[]; # 420
```

Ta fragment kode je zanimiv tudi s sintaktičnega vidika, saj v python vpeljuje deklaracije spremenljivk v taki obliki, kot jo ponavadi najdemo v C-ju in sorodnih jezikih.

(1.4) Tale tekmovalec je najprej pripravil tabelo z n elementi, takoj nato pa že pozabil, koliko elementov ima ta tabela, in je šel to računat s **sizeof**:

```
unsigned int n, k;
cin >> n;
⋮
unsigned int otroci[n];
// Vsem določimo 2 zivljenji
for (int i = 0; i < sizeof(otroci) / sizeof(otroci[0]); ++i) {
```

Pri tej rešitvi je zanimivo tudi to, kako kombinira elemente dveh jezikov: tabela otroci je variable-length array, kakršni obstajajo v C, ne pa v C++; obenem pa uporablja vhodni tok cin, ki obstaja v C++, ne pa v C.

(1.4) Še en zanimiv pogoj pri neskončni zanki:

```
while (true == true)
```

(1.4) Rešitev za ljubitelje nepotrebnega kompliciranja:

```
otrok = n − (n − 1);
```

(1.5) Današnja mladina je pa res razvajena pri delu s pomnilnikom. Temu človeku se zdi prava malenkost alocirati nekaj terabajtov pomnilnika:

naredimo **int** a[1000000000000] = {0}; // *tabelo, ki ima vse vrednosti postavljene na 0.*
Predpostavljamo, da ne bo vnešenih več kot 10 na 12 podatkov, kar v praktičnem primeru sigurno ne bo.

(1.5) Rešitev z globokim čutom za človeškost:

```
hindex = maxh; # LAHKO BI IZPISAL MAXH, AMPAK JE HINDEX BOLJ ČLOVEŠKO
print(hindex);
```

(1.5) Pri nalogah tipa „opiši postopek“ je vedno nekaj takih rešitev, ki opisujejo, kako bi človek napisal program, ki reši nalogo. Tule je en lep primer z letošnjega tekmovanja:

> Postopek se bi začel z pisanjem podprograma, ki bi določil največje število, npr. 12, in jo shranil v spremenljivko, npr. B.

(1.5) Tole je že skoraj haiku:

> Preberem vse podatke
> si jih zapišem
> vse zapišem v tabelo

(1.5) Rešitev z visokimi pričakovanji do funkcije sort:

> nato bi program preveril, katere št. so vnešene, s funkcijo sort, ki je v **#include** <algoritem>.

(1.5) Rešitev, ki uporabnika nadleguje z vnašanjem nepotrebnih podatkov:

> Vprašamo po številu člankov, ki jih je nek raziskovalec objavil, in po njihovem naslovu;

S temi naslovi seveda kasneje ničesar ne počne.

(1.5) Temeljit razmislek o vhodnih podatkih:

> Imamo število objavljenih člankov ter število njihovih citatov. To ločimo na dva različna pojma:
> — št. člankov,
> — citati

(2.1) Iz ene od rešitev:

> Nastavimo nek int_1 na **true** in dokler je ta int_1 enak **true** do takrat ponovno zažnjujemo naslednjo „funkcijo“: (**while** zanka)

To je ne le cvetka, ampak že skoraj cel šopek: ima spremenljivko po imenu „int_1“, v njej pa hrani **bool**e; telesu zanke pravi „funkcija“; in glagol „zaganjati“ se je prelevil v zelo nepravilnega in dobil obliko „zažnjujemo“...

(2.1) Še en primer rešitve tipa „opiši postopek“, ki v resnici opisuje program. Na koncu odgovora je dodal:

> Da ne bom zgubil točk, vse spremenljivke so integer, program začnem z **begin** in končam z **end**. 1. vrstica programa je **program** sodstvo;

(2.3) Tale bi si privoščil kar 1000 računalnikov, je pa poskusil stvar optimizirati tako, da bi jih postopoma pošiljal v akcijo in tako mogoče odkril virus, še preden zažene vse računalnike:

> Imamo 1000 računalnikov in 24 ur časa. Lotimo se po urah. Najprej prvo uro pregledujemo s 40 računalniki. V primeru, da v eni uri ne najdemo zgoščenke, ki vsebuje virus, dodamo še dodatnih 40 računalnikov za pregledovanje. Ta postopek ponavljamo do 23. ure, kjer bomo uporabljali že 920 računalnikov. Če pa še vedno ne najdemo zgoščenke z virusom, v 24. uri dodamo še zadnjih 80 računalnikov za pregledovanje. [...] Če naletimo na okuženo zgoščenko v zgodnjih urah, bo poraba računalnikov manjša. Če naletimo v poznih urah, bo pa velika.

Nekaj drugih tekmovalcev je poskušalo pri tej nalogi rešitev s 1000 računalniki izboljšati tako, da so opazili, da če na 999 zgoščenkah ni virusa, potem lahko zaključimo, da je virus na tisti preostali, 1000. zgoščenki, ki je torej ni treba preizkusiti. Zato je dovolj že samo 999 računalnikov.

(2.3) Rešitev podnaloge (*b*) za ljubitelje kombinatorike:

> Uporabimo 500 računalnikov in v vsak računalnik vstavimo 2 CD. Nato jih premešamo tako, da lahko s pomočjo kombinatorike ugotovimo, kateri CD je okužen glede na to, kateri računalniki so se sesuli.

To pa je tudi vse, kar je pri tej podnalogi napisal.

(2.3) Nagrado za najboljše izmikanje omejitvam naloge dobi:

> Predpostavimo, da imamo zelo močan računalnik in na njem poženemo 1000 instanc virtualnega sistema. V vsaki instanci virtualnega sistema poženemo po eno zgoščenko in po 1 dnevu pogledamo, kateri virtualni sistem ne deluje več.

Kot tekmovalec tudi sam omeni v nadaljevanju rešitve, je s tem pravzaprav rešil tako podnalogo (a) kot (b).

(2.3) Rešitev za ljubitelje arogance:

> 1. del programa ni vreden mojega časa

Nekakšno rešitev je potem vendarle napisal.

(2.3) Rešitev, ki se z veliko truda in naglice prepriča le o tem, da je vsaj ena zgoščenka res okužena:

> (a) Če želimo virus najti v minimalnem številu dni, zaženemo vse zgoščenke v čim krajšem času na istem računalniku. Med njimi bo tudi okužena zgoščenka, ki bo poskrbela, da se bo računalnik do naslednjega dne sesul.

(2.4) Rešitev s sklicevanjem na izmišljene funkcije:

```
data[s[0]] = solve(buff) # Rešimo enačbo, ki je zapisana v spremenljivki buff
⋮
result = assess(s) # Preverimo, če je leva stran enačbe v s enaka desni strani enačbe v s
```

Funkcij solve in assess pa seveda ni napisal.

(2.4) Eden od tekmovalcev je prišel na naslednjo odlično idejo za rešitev te naloge: program vsaki enoti pripiše neko praštevilo; potem lahko vsako količino predstavimo z nekim (racionalnim) številom tako, da zmnožimo ali delimo ustrezna praštevila. Pravilnost enačbe potem preverimo tako, da vanjo vstavimo števila, ki predstavljajo količine v njej, in preverimo, če je leva stran zdaj enaka desni.

(2.5) Naslednjemu tekmovalcu očitno ni dovolj to, da bi bili žužki heteroseksualni, ampak hoče tudi strogo monogamijo:

> Če je število število žužkov enega spola manjše od števila žužkov drugega spola, potem ne bo moralo priti pri vseh žužkih do heteroseksualne interakcije.

Naslednji tekmovalec pa je šel v nasprotno skrajnost:

> Če so vsi žužki imeli dve ali več interakcij z dvema različnima žužkoma, je možnost, da so vsi heteroseksualci.

(2.5) Rešitev za ljubitelje tavtologij:

> To naredi z "loop" zanko (npr. **for**), ki se ponavlja od 1 do n.

(2.5) Rešitev, ki pretirava z uporabo naprednih podatkovnih struktur:

> nato bi ustvaril slovar spol (python: {}), v katerem bi bil "key" vsaka izmed možnih številk žužkov ($0 \ldots n$), zaenkrat pa bi v paru (value) bil Null.

Po domače povedano, ta slovar bi bil lahko brez škode čisto navadna tabela.

(2.5) Pogoj, ki ne bo pogosto izpolnjen:

```
if (i != j && i == seznam[a] && j == seznam[a])
```

(3.1) Tale rešitev vsakemu zaboju pripiše univerzalni enolični identifikator:

```
import java.util.UUID;
⋮
private static class Zaboj {
   public int teza;
   public UUID uuid = UUID.randomUUID();
   ⋮
}
```

Te UUIDe kasneje tudi res uporablja za primerjanje, če se dve spremenljivki nanašata na isti zaboj:

```
if (b.uuid.equals(a.uuid)) continue;
```

Z zaboji sicer ne počne ničesar takega, da se ne bi dalo čisto istega učinka doseči že s primerjavo referenc, **if** (b == a).

(3.2) Pri tej nalogi je bilo več bleferskih rešitev, ki vedno izpišejo enak odgovor: ena vedno izpiše odgovor iz primera pri besedilu naloge, ena vedno izpiše `0`, ena pa vedno izpiše `-1`. Na srečo so bili naši testni primeri izbrani tako, da so vse tri dobile po 0 točk.

(3.4) Letošnjo nagrado za najgloblje gnezdenje template argumentov dobi:

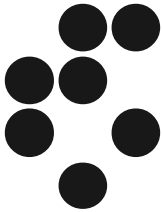
```
priority_queue<pair<int, pair<pair<int, int>, pair<int, int> > > > pq;
```

Za konec pa še ena cvetka iz anket:

> (*Vprašanje v anketi.*) Če imaš kaj vrstnikov, ki se tudi zanimajo za programiranje, pa se tega tekmovanja niso udeležili, kaj bi bilo po tvojem mnenju treba spremeniti, da bi jih prepričali k udeležbi?
>
> (*Odgovor.*) Tepst bi jih bilo treba. Barabe lene!

SODELUJOČE INŠTITUCIJE

Institut Jožef Stefan



Institut je največji javni raziskovalni zavod v Sloveniji s skoraj 800 zaposlenimi, od katerih ima približno polovica doktorat znanosti. Več kot 150 naših doktorjev je habilitiranih na slovenskih univerzah in sodeluje v visokošolskem izobraževalnem procesu. V zadnjih desetih letih je na Institutu opravilo svoja magistrska in doktorska dela več kot 550 raziskovalcev. Institut sodeluje tudi s srednjimi šolami, za katere organizira delovno prakso in jih vključuje v aktivno raziskovalno delo. Glavna raziskovalna področja Instituta so fizika, kemija, molekularna biologija in biotehnologija, informacijske tehnologije, reaktorstvo in energetika ter okolje.

Poslanstvo Instituta je v ustvarjanju, širjenju in prenosu znanja na področju naravoslovnih in tehniških znanosti za blagostanje slovenske družbe in človeštva nasploh. Institut zagotavlja vrhunsko izobrazbo kadrom ter raziskave in razvoj tehnologij na najvišji mednarodni ravni.

Institut namenja veliko pozornost mednarodnemu sodelovanju. Sodeluje z mnogimi uglednimi institucijami po svetu, organizira mednarodne konference, sodeluje na mednarodnih razstavah. Poleg tega pa po najboljših močeh skrbi za mednarodno izmenjavo strokovnjakov. Mnogi raziskovalni dosežki so bili deležni mednarodnih priznanj, veliko sodelavcev IJS pa je mednarodno priznanih znanstvenikov.

Tekmovanje sta podprla naslednja odseka IJS:

CT3 — Center za prenos znanja na področju informacijskih tehnologij

Center za prenos znanja na področju informacijskih tehnologij izvaja izobraževalne, promocijske in infrastrukturne dejavnosti, ki povezujejo raziskovalce in uporabnike njihovih rezultatov. Z uspešnim vključevanjem v evropske raziskovalne projekte se Center širi tudi na raziskovalne in razvojne aktivnosti, predvsem s področja upravljanja z znanjem v tradicionalnih, mrežnih ter virtualnih organizacijah. Center je partner v več EU projektih.

Center razvija in pripravlja skrbno načrtovane izobraževalne dogodke kot so seminarji, delavnice, konference in poletne šole za strokovnjake s področij inteligentne analize podatkov, rudarjenja s podatki, upravljanja z znanjem, mrežnih organizacij, ekologije, medicine, avtomatizacije proizvodnje, poslovnega odločanja in še kaj. Vsi dogodki so namenjeni prenosu osnovnih, dodatnih in vrhunskih specialističnih znanj v podjetja ter raziskovalne in izobraževalne organizacije. V ta namen smo postavili vrsto izobraževalnih portalov, ki ponujajo že za več kot 500 ur posnetih izobraževalnih seminarjev z različnih področij.

Center postaja pomemben dejavnik na področju prenosa in promocije vrhunskih naravoslovno-tehniških znanj. S povezovanjem vrhunskih znanj in dosežkov različnih področij, povezovanjem s centri odličnosti v Evropi in svetu, izkoriščanjem različnih metod in sodobnih tehnologij pri prenosu znanj želimo zgraditi virtualno učečo se skupnost in pripomoči k učinkovitejšemu povezovanju znanosti in industrije ter večji prepoznavnosti domačega znanja v slovenskem, evropskem in širšem okolju.

E3 — Laboratorij za umetno inteligenco
Področje dela Laboratorija za umetno inteligenco so informacijske tehnologije s poudarkom na tehnologijah umetne inteligence. Najpomembnejša področja raziskav in razvoja so: (*a*) analiza podatkov s poudarkom na tekstovnih, spletnih, večpredstavnih in dinamičnih podatkih, (*b*) tehnike za analizo velikih količin podatkov v realnem času, (*c*) vizualizacija kompleksnih podatkov, (*d*) semantične tehnologije, (*e*) jezikovne tehnologije.

Laboratorij za umetno inteligenco posveča posebno pozornost promociji znanosti, posebej med mladimi, kjer v sodelovanju s Centrom za prenos znanja na področju informacijskih tehnologij (CT3) razvija izobraževalni portal VideoLectures.NET in vrsto let organizira tekmovanja ACM v znanju računalništva.

Laboratorij tesno sodeluje s Stanford University, University College London, Mednarodno podiplomsko šolo Jožefa Stefana ter podjetji Quintelligence, Cycorp Europe, LifeNetLive, Modro Oko in Envigence.

*

Fakulteta za matematiko in fiziko
Fakulteta za matematiko in fiziko je članica Univerze v Ljubljani. Sestavljata jo Oddelek za matematiko in Oddelek za fiziko. Izvaja dodiplomske univerzitetne študijske programe matematike, računalništva in informatike ter fizike na različnih smereh od pedagoških do raziskovalnih.

Prav tako izvaja tudi podiplomski specialistični, magistrski in doktorski študij matematike, fizike, mehanike, meteorologije in jedrske tehnike.

Poleg rednega pedagoškega in raziskovalnega dela na fakulteti poteka še vrsta obštudijskih dejavnosti v sodelovanju z različnimi institucijami od Društva matematikov, fizikov in astronomov do Inštituta za matematiko, fiziko in mehaniko ter Instituta Jožef Stefan. Med njimi so tudi tekmovanja iz programiranja, kot sta Programerski izziv in Univerzitetni programerski maraton.

Fakulteta za računalništvo in informatiko
Glavna dejavnost Fakultete za računalništvo in informatiko Univerze v Ljubljani je vzgoja računalniških strokovnjakov različnih profilov. Oblike izobraževanja se razlikujejo med seboj po obsegu, zahtevnosti, načinu izvajanja in številu udeležencev. Poleg rednega izobraževanja skrbi fakulteta še za dopolnilno izobraževanje računalniških strokovnjakov, kot tudi strokovnjakov drugih strok, ki potrebujejo znanje informatike. Prav posebna in zelo osebna pa je vzgoja mladih raziskovalcev, ki se med podiplomskim študijem pod mentorstvom univerzitetnih profesorjev uvajajo v raziskovalno in znanstveno delo.

Fakulteta za elektrotehniko, računalništvo in informatiko

Fakulteta za elektrotehniko, računalništvo in informatiko (FERI) je znanstveno-izobraževalna institucija z izraženim regionalnim, nacionalnim in mednarodnim pomenom. Regionalnost se odraža v tesni povezanosti z industrijo v mestu Maribor in okolici, kjer se zaposluje pretežni del diplomantov dodiplomskih in podiplomskih študijskih programov. Nacionalnega pomena so predvsem inštituti kot sestavni deli FERI ter centri znanja, ki opravljajo prenos temeljnih in aplikativnih znanj v celoten prostor Republike Slovenije. Mednarodni pomen izkazuje fakulteta z vpetostjo v mednarodne raziskovalne tokove s številnimi mednarodnimi projekti, izmenjavo študentov in profesorjev, objavami v uglednih znanstvenih revijah, nastopih na mednarodnih konferencah in organizacijo le-teh.

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Univerze na Primorskem (UP FAMNIT) je prvo generacijo študentov vpisala v študijskem letu 2007/08, pod okriljem UP PEF pa so se že v študijskem letu 2006/07 izvajali podiplomski študijski programi Matematične znanosti in Računalništvo in informatika (magistrska in doktorska programa).

Z ustanovitvijo UP FAMNIT je v letu 2006 je Univerza na Primorskem pridobila svoje naravoslovno uravnoteženje. Sodobne tehnologije v naravoslovju predstavljajo na začetku tretjega tisočletja poseben izziv, saj morajo izpolniti interese hitrega razvoja družbe, kakor tudi skrb za kakovostno ohranjanje naravnega in družbenega ravnovesja. V tem matematična znanja, področje informacijske tehnologije in druga naravoslovna znanja predstavljajo ključ do odgovora pri vprašanjih modeliranja družbeno ekonomskih procesov, njihove logike in zakonitosti racionalnega razmišljanja.

ACM Slovenija

ACM je največje računalniško združenje na svetu s preko 80 000 člani. ACM organizira vplivna srečanja in konference, objavlja izvirne publikacije in vizije razvoja računalništva in informatike.

ACM Slovenija smo ustanovili leta 2001 kot slovensko podružnico ACM. Naš namen je vzdigniti slovensko računalništvo in informatiko korak naprej v bodočnost.

Društvo se ukvarja z:

- Sodelovanjem pri izdaji mednarodno priznane revije Informatica — za doktorande je še posebej zanimiva možnost objaviti 2 strani poročila iz doktorata.
- Urejanjem slovensko-angleškega slovarčka — slovarček je narejen po vzoru Wikipedije, torej lahko vsi vanj vpisujemo svoje predloge za nove termine, glavni uredniki pa pregledujejo korektnost vpisov.
- ACM predavanja sodelujejo s Solomonovimi seminarji.
- Sodelovanjem pri organizaciji študentskih in dijaških tekmovanj iz računalništva.

ACM Slovenija vsako leto oktobra izvede konferenco Informacijska družba in na njej skupščino ACM Slovenija, kjer volimo predstavnike.

IEEE Slovenija

Inštitut inženirjev elektrotehnike in elektronike, znan tudi pod angleško kratico IEEE (Institute of Electrical and Electronics Engineers) je svetovno združenje inženirjev omenjenih strok, ki promovira inženirstvo, ustvarjanje, razvoj, integracijo in pridobivanje znanja na področju elektronskih in informacijskih tehnologij ter znanosti.

REPUBLIKA SLOVENIJA
MINISTRSTVO ZA IZOBRAŽEVANJE, ZNANOST IN ŠPORT

Ministrstvo za izobraževanje, znanost in šport

Ministrstvo za izobraževanje, znanost in šport opravlja upravne in strokovne naloge na področjih predšolske vzgoje, osnovnošolskega izobraževanja, osnovnega glasbenega izobraževanja, nižjega in srednjega poklicnega ter srednjega strokovnega izobraževanja, srednjega splošnega izobraževanja, višjega strokovnega izobraževanja, izobraževanja otrok in mladostnikov s posebnimi potrebami, izobraževanja odraslih, visokošolskega izobraževanja, znanosti, ter športa.

SREBRNI POKROVITELJ

Quintelligence

Obstoječi informacijski sistemi podpirajo predvsem procesni in organizacijski nivo pretoka podatkov in informacij. Biti lastnik informacij in podatkov pa ne pomeni imeti in obvladati znanja in s tem zagotavljati konkurenčne prednosti. Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja. IKT (informacijsko-komunikacijska tehnologija) je postavila temelje za nemoten pretok in hranjenje podatkov in informacij. S primernimi metodami je potrebno na osnovi teh informacij izpeljati ustrezne analize in odločitve. Nivo upravljanja in delovanja se tako seli iz informacijske logistike na mnogo bolj kompleksen in predvsem nedeterminističen nivo razvoja in uporabe metodologij. Tako postajata razvoj in uporaba metod za podporo obvladovanja znanja (knowledge management, KM) vedno pomembnejši segment razvoja.

Podjetje Quintelligence je in bo usmerjeno predvsem v razvoj in izvedbo metod in sistemov za pridobivanje, analizo, hranjenje in prenos znanja. S kombiniranjem delnih — problemsko usmerjenih rešitev, gradimo kompleksen in fleksibilen sistem za podporo KM, ki bo predstavljal osnovo globalnega informacijskega centra znanja.

Obvladovanje znanja je v razumevanju, sledenju, pridobivanju in uporabi novega znanja.


RTK
40
LET
TEKMOVANJ V ZNANJU
RAČUNALNIŠTVA IN INFORMATIKE
V SLOVENIJI


acm